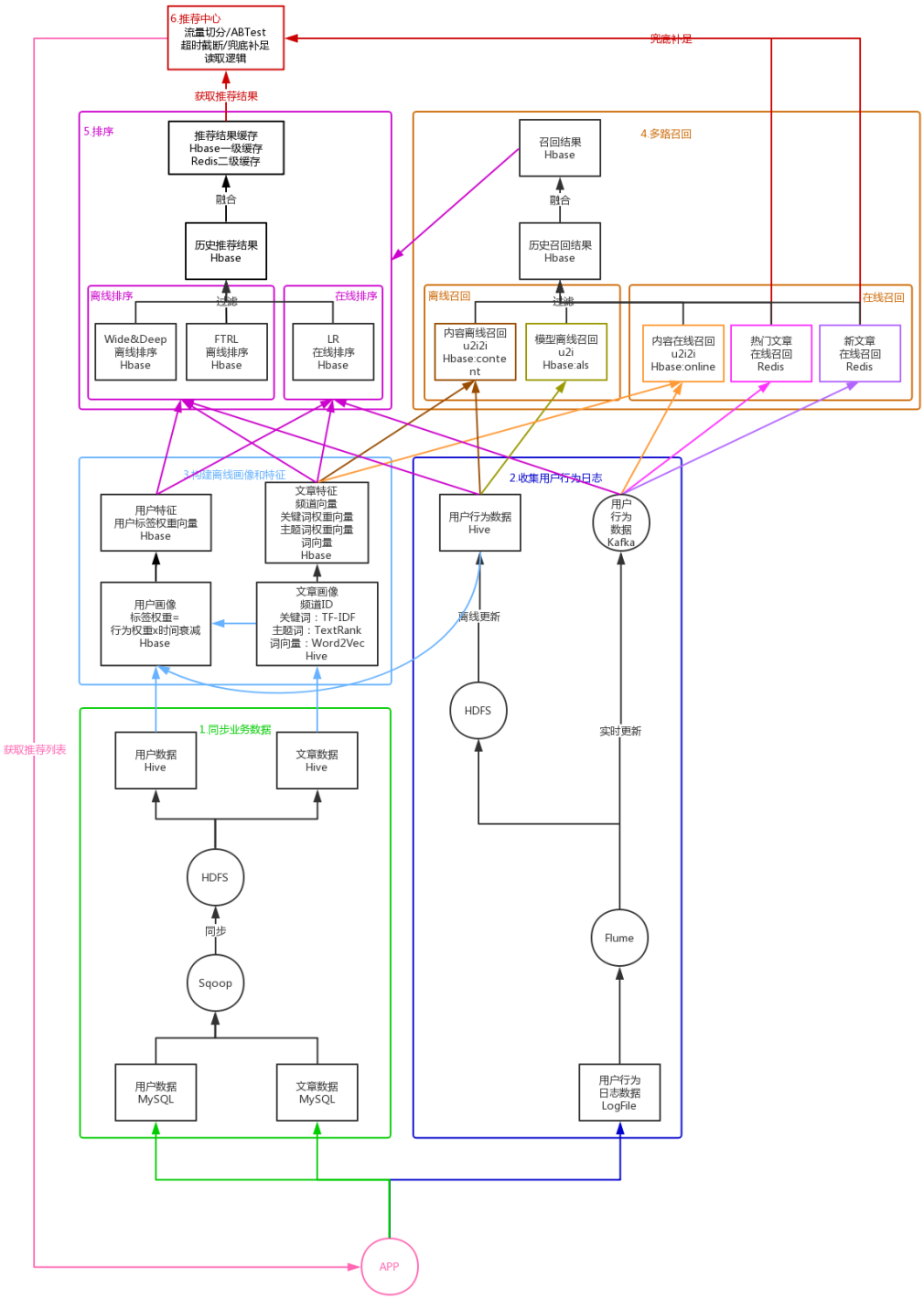
- 电子书：《文章推荐系统》
- 写稿人：小王子特洛伊（https://www.jianshu.com/u/ac833cc5146e ）
- 整理人：Thinkgamer（https://blog.csdn.net/gamer_gyt）
- 特别说明：转载请找作者授权，PDF传播过程中不得去除文前说明和文后介绍
- 更多精彩文章请关注「搜索与推荐Wiki」公众号或者访问博主博客（http://thinkgamer.cn ）

1、推荐流程设计

推荐系统主要解决的是信息过载的问题，目标是从海量物品筛选出不同用户各自喜欢的物品，从而为每个用户提供个性化的推荐。推荐系统往往架设在大规模的业务系统之上，面临着用户的不断增长，物品的不断变化，并且有着全面的推荐评价指标和严格的性能要求（Netflix 的请求时间在 250 ms 以内，今日头条的请求时间在 200ms 以内），所以推荐系统很难一次性地快速计算出用户所喜好的物品，并且同时满足准确度、多样性等评价指标。为了解决如上这些问题，推荐系统通常被设计为三个阶段：召回、排序和调整，如下图所示：

在召回阶段，首先筛选出和用户直接相关或间接相关的物品，将原始数据从万、百万、亿级别缩小到万、千级别；在排序阶段，通常使用二分类算法来预测用户对物品的喜好程度（或者是点击率），然后将物品按照喜好程序从大到小依次排列，筛选出用户最有可能喜欢的物品，这里又将召回数据从万、千级别缩小到千、百级别；最后在调整阶段，需要过滤掉重复推荐的、已经购买或阅读的、已经下线的物品，当召回和排序结果不足时，需要使用热门物品进行补充，最后合并物品基础信息，将包含完整信息的物品推荐列表返回给客户端。

这里以文章推荐系统为例，讲述一下推荐系统的完整流程，如下图所示：


6.推荐中心
流量切分/ABTest
超时截断/兜底补足
读取逻辑
兜底补足
获取推荐结果
5.排序
推荐结果缓存
Hbase一级缓存
Redis二级缓存
融合
历史推荐结果
Hbase
离线排序
过滤
Wide&Deep
离线排序
Hbase
FTRL
离线排序
Hbase
在线排序
LR
在线排序
Hbase
4.多路召回
召回结果
Hbase
融合
历史召回结果
Hbase
离线召回
过滤
内容离线召回
u2i2i
Hbase:conte
nt
模型离线召回
u2i
Hbase:als
在线召回
内容在线召回
u2i2i
Hbase:online
热门文章
在线召回
Redis
新文章
在线召回
Redis
3.构建离线画像和特征
用户特征
用户标签权重向量
Hbase
文章特征
频道向量
关键词权重向量
主题词权重向量
词向量
Hbase
用户画像
标签权重=
行为权重x时间衰减
Hbase
文章画像
频道ID
关键词：TF-IDF
主题词：TextRank
词向量：Word2Vec
Hive
2.收集用户行为日志
用户行为数据
Hive
用户
行为
数据
Kafka
离线更新
HDFS
实时更新
Flume
用户行为
日志数据
LogFile
获取推荐列表
1.同步业务数据
用户数据
Hive
文章数据
Hive
HDFS
同步
Sqoop
用户数据
MySQL
文章数据
MySQL
APP

1. 同步业务数据 为了避免推荐系统的数据读写、计算等对应用产生影响，我们首先要将业务数据从应用数据库 MySQL 同步到推荐系统数据库 Hive 中，这里利用 Sqoop 先将 MySQL 中的业务数据同步到推荐系统的 HDFS 中，再关联到指定的 Hive 表中，这样就可以在推荐系统数据库 Hive 中使用用户数据和文章数据了，并且不会对应用产生任何影响。

2. 收集用户行为数据 除了用户数据和文章数据，我们还需要得到用户对文章的行为数据，比如曝光、点击、阅读、点赞、收藏、分享、评论等。我们的用户行为数据是记录在应用服务器的日志文件中的，所以可以利用 Flume 对应用服务器的日志文件进行监听，一方面将收集到的用户行为数据同步到 HDFS 中，并关联到 Hive 的用户行为表，每天更新一次，以供离线计算使用。另一方面将 Flume 收集到的用户行为数据同步到 Kafka，实时更新，以供在线计算使用。

3. 构建离线画像和特征

- 文章画像由关键词和主题词组成，我们首先读取 Hive 中的文章数据，将文章内容进行分词，根据 TF-IDF 模型计算每个词的权重，将 TF-IDF 权重最高的 K 个词作为关键词，再根据 TextRank 模型计算每个词的权重，将 TextRank 权重最高的 K 个词与 TF-IDF 权重最高的 K 个词的共现词作为主题词，将关键词和主题词存储到 Hive 的文章画像表中。接下来，利用 Word2Vec 模型，计算得到所有关键词的平均向量，作为文章的词向量，存储到 Hive 的文章向量表中，并利用 BucketedRandomProjectionLSH 模型计算得到文章的相似度，将每篇文章相似度最高的 K 篇文章，存储到 Hbase 的文章相似表中。这样我们就得到了每篇文章的画像、词向量以及相似文章列表。

- 构建离线用户画像 我们可以将用户喜欢的文章的主题词作为用户标签，以便后面根据用户标签来推荐符合其偏好的文章。首先读取用户行为数据和文章画像数据，计算在用户产生过行为的所有文章中，每个主题词的权重，不同的行为，权重不同，计算公式为：用户标签权重 =（用户行为分值之和）x 时间衰减，这样就计算得到了用户的标签及标签权重，接着读取用户数据，得到用户基础信息，将用户标签、标签权重及用户基础信息一并存储到 Hbase 的用户画像表中。到这里我们已经通过机器学习算法，基于用户和文章的业务数据得到了用户和文章的画像，但为了后面可以更方便地将数据提供给深度学习模型进行训练，我们还需要将画像数据进一步抽象为特征数据。

- 构建离线文章特征 由于已经有了画像信息，特征构造就变得简单了。读取文章画像数据，将文章权重最高的 K 个关键词的权重作为文章关键词权重向量，将频道 ID、关键词权重向量、词向量作为文章特征存储到 Hbase 的文章特征表。

- 构建离线用户特征 读取用户画像数据，将权重最高的 K 个标签的权重作为用户标签权重向量，将用户标签权重向量作为用户特征存储到 Hbase 的用户特征表。

4. 多路召回

- 基于模型的离线召回 我们可以根据用户的历史点击行为来预测相似用户，并利用相似用户的点击行为来预测对文章的偏好得分，这种召回方式称为 u2u2i。获取用户历史点击行为数据，利用 ALS 模型计算得到用户对文章的偏好得分及文章列表，读取并过滤历史召回结果，防止重复推荐，将过滤后的偏好得分最高的 K 篇文章存入 Hbase 的召回结果表中，key 为 als，表明召回类型为 ALS 模型召回，并记录到 Hbase 的历史召回结果表。

- 基于内容的离线召回 我们可以根据用户的历史点击行为，向用户推荐其以前喜欢的文章的相似文章，这种方式称为 u2i2i。读取用户历史行为数据，获取用户历史发生过点击、阅读、收藏、分享等行为的文章，接着读取文章相似表，获取与发生行为的每篇文章相似度最高的 K 篇文章，然后读取并过滤历史召回结果，防止重复推荐，最后将过滤后的文章存入 Hbase 的召回结果表中，key 为 content，表明召回类型为内容召回，并记录到 Hbase 的历史召回结果表。

- 基于内容的在线召回 和上面一样，还是根据用户的点击行为，向用户推荐其喜欢的文章的相似文章，不过这里是用户实时发生的行为，所以叫做在线召回。读取 Kafka 中的用户实时行为数据，获取用户实时发生点击、阅读、收藏、分享等行为的文章，接着读取文章相似表，获取与发生行为的每篇文章相似度最高的 K 篇文章，然后读取并过滤历史召回结果，防止重复推荐，最后将过滤后的文章存入 Hbase 的召回结果表中，key 为 online，表明召回类型为在线召回，并记录到 Hbase 的历史召回结果表。

- 基于热门文章的在线召回 读取 Kafka 中的用户实时行为数据，获取用户当前发生点击、阅读、收藏、分享等行为的文章，增加这些文章在 Redis 中的热度分数。

- 基于新文章的在线召回 读取 Kafka 中的实时用户行为数据，获取新发布的文章，将其加入到 Redis 中，并设置过期时间。

5. 排序

不同模型的做法大致相同，这里以 LR 模型为例。

- 基于 LR 模型的离线训练 读取 Hive 的用户历史行为数据，并切分为训练集和测试集，根据其中的用户 ID 和文章 ID，读取 Hbase 的用户特征数据和文章特征数据，将二者合并作为训练集的输入特征，将用户对文章是否点击作为训练集的标签，将上一次的模型参数作为 LR 模型的初始化参数，进行点击率预估训练，计算得出 AUC 等评分指标并进行推荐效果分析。

- 基于 LR 模型的在线排序 当推荐中心读取 Hbase 的推荐结果表无数据时，推荐中心将调用在线排序服务来重新获取推荐结果。排序服务首先读取 Hbase 的召回结果作为测试集，读取 Hbase 的用户特征数据和文章特征数据，将二者合并作为测试集的输入特征，使用 LR 模型进行点击率预估，计算得到点击率最高的前 K 个文章，然后读取并过滤历史推荐结果，防止重复推荐，最后将过滤后的文章列表存入 Hbase 的推荐结果表中，key 为 lr，表明排序类型为 LR 排序。

6. 推荐中心

- 流量切分（ABTest） 我们可以根据用户 ID 进行哈希分桶，将流量切分到多个桶，每个桶对应一种排序策略，从而对比不同排序策略在线上环境的效果。

- 推荐数据读取逻辑 优先读取 Redis 和 Hbase 中缓存的推荐结果，若 Redis 和 Hbase 都为空，则调用在线排序服务获得推荐结果。

- 兜底补足（超时截断） 当调用排序服务无结果，或者读取超时的时候，推荐中心会截断当前请求，直接读取 Redis 中的热门文章和新文章作为推荐结果。

- 合并信息 合并物品基础信息，将包含完整信息的物品推荐列表返回给客户端。

2、同步业务数据

在推荐系统架构中，推荐系统的数据库和业务系统的数据库是分离的，这样才能有效避免推荐系统的数据读写、计算等对业务系统的影响，所以同步业务数据往往也是推荐系统要做的第一件事情。通常业务数据是存储在关系型数据库中，比如 MySQL，而推荐系统通常需要使用大数据平台，比如 Hadoop，我们可以利用 Sqoop 将 MySQL 中的业务数据同步到 Hive中，在我们的文章推荐系统中，需要同步的数据包括：

- 用户信息表（user_profile）：包括 user_id | gender | birthday | real_name | create_time | update_time | register_media_time | id_number | id_card_front | id_card_back | id_card_handheld | area | company | career 等字段。
- 用户基础信息表（user_basic）：包括 user_id | mobile | password | user_name | profile_photo | last_login | is_media | article_count | following_count | fans_count | like_count | read_count | introduction | certificate | is_verified | account | email | status 等字段。
- 文章信息表（news_article_basic）：包括 article_id | user_id | channel_id | title | cover | is_advertising | create_time | status | reviewer_id | review_time | delete_time | comment_count | allow_comment | update_time | reject_reason 等字段。
- 文章内容表（news_article_content）：包括 article_id | content | update_time 等字段。
- 频道信息表（news_channel）：包括 channel_id | channel_name | create_time | update_time | sequence | is_visible | is_default 等字段。

首先，在 Hive 中创建业务数据库 toutiao，并创建相关表

```
create database if not exists toutiao comment "user,news
information of mysql" location '/user/hive/warehouse/toutiao.db/';
-- 用户信息表
create table user_profile
(
    user_id             BIGINT comment "userid",
    gender              BOOLEAN comment "gender",
    birthday            STRING comment "birthday",
    real_name           STRING comment "real_name",
    create_time         STRING comment "create_time",
    update_time         STRING comment "update_time",
    register_media_time STRING comment "register_media_time",
    id_number           STRING comment "id_number",
    id_card_front       STRING comment "id_card_front",
    id_card_back        STRING comment "id_card_back",
    id_card_handheld    STRING comment "id_card_handheld",
    area                STRING comment "area",
    company             STRING comment "company",
    career              STRING comment "career"
)
    COMMENT "toutiao user profile"
    row format delimited fields terminated by ',' # 指定分隔符
    LOCATION '/user/hive/warehouse/toutiao.db/user_profile';
-- 用户基础信息表
create table user_basic
```

```
(
    user_id         BIGINT comment "user_id",
    mobile          STRING comment "mobile",
    password        STRING comment "password",
    profile_photo   STRING comment "profile_photo",
    last_login      STRING comment "last_login",
    is_media        BOOLEAN comment "is_media",
    article_count   BIGINT comment "article_count",
    following_count BIGINT comment "following_count",
    fans_count      BIGINT comment "fans_count",
    like_count      BIGINT comment "like_count",
    read_count      BIGINT comment "read_count",
    introduction    STRING comment "introduction",
    certificate     STRING comment "certificate",
    is_verified     BOOLEAN comment "is_verified"
)
    COMMENT "toutiao user basic"
    row format delimited fields terminated by ',' # 指定分隔符
    LOCATION '/user/hive/warehouse/toutiao.db/user_basic';
-- 文章基础信息表
create table news_article_basic
(
    article_id  BIGINT comment "article_id",
    user_id     BIGINT comment "user_id",
    channel_id  BIGINT comment "channel_id",
    title       STRING comment "title",
    status      BIGINT comment "status",
    update_time STRING comment "update_time"
)
    COMMENT "toutiao news_article_basic"
    row format delimited fields terminated by ',' # 指定分隔符
    LOCATION '/user/hive/warehouse/toutiao.db/news_article_basic';
-- 文章频道表
create table news_channel
(
    channel_id   BIGINT comment "channel_id",
    channel_name STRING comment "channel_name",
    create_time  STRING comment "create_time",
    update_time  STRING comment "update_time",
    sequence     BIGINT comment "sequence",
    is_visible   BOOLEAN comment "is_visible",
    is_default   BOOLEAN comment "is_default"
)
    COMMENT "toutiao news_channel"
    row format delimited fields terminated by ',' # 指定分隔符
    LOCATION '/user/hive/warehouse/toutiao.db/news_channel';
-- 文章内容表
create table news_article_content
```

```
(
    article_id BIGINT comment "article_id",
    content    STRING comment "content",
    update_time STRING comment "update_time"
)
    COMMENT "toutiao news_article_content"
    row format delimited fields terminated by ',' # 指定分隔符
    LOCATION
'/user/hive/warehouse/toutiao.db/news_article_content';
```

查看 Sqoop 连接到 MySQL 的数据库列表

```
sqoop list-databases --connect jdbc:mysql://192.168.19.137:3306/ --
username root -P

mysql
sys
toutiao
```

Sqoop 可以指定全量导入和增量导入，通常我们可以先执行一次全量导入，将历史数据全部导入进来，后面再通过定时任务执行增量导入，来保持 MySQL 和 Hive 的数据同步，全量导入不需要提前创建 Hive 表，可以自动创建

```
array=(user_profile user_basic news_article_basic news_channel
news_article_content)

for table_name in ${array[@]};
do
    sqoop import \
        --connect jdbc:mysql://192.168.19.137/toutiao \
        --username root \
        --password password \
        --table $table_name \
        --m 5 \ # 线程数
        --hive-home /root/bigdata/hive \ # hive路径
        --hive-import \ # 导入形式
        --create-hive-table  \ # 自动创建表
        --hive-drop-import-delims \
        --warehouse-dir /user/hive/warehouse/toutiao.db \ # 导入地址
        --hive-table toutiao.$table_name
done
```

增量导入，有 append 和 lastmodified 两种模式

- append：通过指定一个递增的列进行更新，只能追加，不能修改

```
num=0
declare -A check
check=([user_profile]=user_id [user_basic]=user_id
[news_article_basic]=article_id [news_channel]=channel_id
[news_article_content]=channel_id)

for k in ${!check[@]}
do
    sqoop import \
        --connect jdbc:mysql://192.168.19.137/toutiao \
        --username root \
        --password password \
        --table $k \
        --m 4 \
        --hive-home /root/bigdata/hive \ # hive路径
        --hive-import \ # 导入到hive
        --create-hive-table  \ # 自动创建表
        --incremental append \ # 按照id更新
        --check-column ${check[$k]} \ # 指定id列
        --last-value ${num} # 指定最后更新的id
done
```

- lastmodified：按照最后修改时间更新，支持追加和修改

```
time=`date +"%Y-%m-%d" -d "-1day"`
declare -A check
check=([user_profile]=update_time [user_basic]=last_login
[news_article_basic]=update_time [news_channel]=update_time)
declare -A merge
merge=([user_profile]=user_id [user_basic]=user_id
[news_article_basic]=article_id [news_channel]=channel_id)

for k in ${!check[@]}
do
    sqoop import \
        --connect jdbc:mysql://192.168.19.137/toutiao \
        --username root \
        --password password \
        --table $k \
        --m 4 \
        --target-dir /user/hive/warehouse/toutiao.db/$k \ # hive路径
        --incremental lastmodified \ # 按照最后修改时间更新
        --check-column ${check[$k]} \ # 指定时间列
        --merge-key ${merge[$k]} \ # 根据指定列合并重复或修改数据
        --last-value ${time} # 指定最后修改时间
done
```

Sqoop 可以直接导入到 Hive，自动创建 Hive 表，但是 lastmodified 模式不支持
Sqoop 也可以先导入到 HDFS，然后再建立 Hive 表关联，为了使用 lastmodified 模式，通常使用这种方法

```
--target-dir /user/hive/warehouse/toutiao.db/user_profile # 指定导入的HDFS目录
```

若先导入到 HDFS，需要注意 HDFS 默认分隔符为 `,` 而 Hive 默认分隔符为 `/u0001`，所以需要在创建 Hive 表时指定分隔符为 HDFS 分隔符 `,`，若不指定分隔符，查询结果将全部为 NULL

```
row format delimited fields terminated by ',' # 指定分隔符
```

如果 MySQL 数据中存在特殊字符，如 `, \t \n` 都会导致 Hive 读取失败（但不影响导入到 HDFS 中），可以利用 `--query` 参数，在读取时使用 `REPLACE() CHAR() CHR()` 进行字符替换。如果特殊字符过多，比如 news_article_content 表，可以选择全量导入

```
--query 'select article_id, user_id, channel_id,
REPLACE(REPLACE(REPLACE(title, CHAR(13),""),CHAR(10),""), ",", " ")
title, status, update_time from news_article_basic WHERE
$CONDITIONS' \
```

如果 MySQL 数据中存在 tinyint 类型,必须在 `--connet` 中加入 `?tinyInt1isBit=false`，防止 Hive 将 tinyint 类型默认转化为 boolean 类型

```
--connect jdbc:mysql://192.168.19.137/toutiao?tinyInt1isBit=false
```

MySQL 与 Hive 对应类型如下

MySQL	Hive
bigint	bigint
tinyint	boolean
int	int
double	double
bit	boolean
varchar	string
decimal	double
date / timestamp	string

我们可以利用 crontab 来创建定时任务，将更新命令写入脚本 import_incremental.sh，输入 `crontab -e` 打开编辑界面，输入如下内容，即可定时执行数据同步任务

```
# 每隔半小时增量导入一次
*/30 * * * * /root/toutiao_project/scripts/import_incremental.sh
```

crontab 命令格式为 `* * * * * shell`，其中前五个 `*` 分别代表分钟 (0-59)、小时(0-59)、月内的某天(1-31)、月(1-12)、周内的某天(0-7，周日为 0 或 7)， `shell` 表示要执行的命令或脚本，使用方法如下

```
# 每隔5分钟运行一次backupscript脚本
*/5 * * * * /root/backupscript.sh
# 每天的凌晨1点运行backupscript脚本
0 1 * * * /root/backupscript.sh
# 每月的第一个凌晨3:15运行backupscript脚本
15 3 1 * * /root/backupscript.sh
```

crontab 常用命令如下

```
crontab -e      # 修改 crontab 文件，如果文件不存在会自动创建。
crontab -l      # 显示 crontab 文件。
crontab -r      # 删除 crontab 文件。
crontab -ir     # 删除 crontab 文件前提醒用户。

service crond status      # 查看crontab服务状态
service crond start       # 启动服务
service crond stop        # 关闭服务
service crond restart     # 重启服务
service crond reload      # 重新载入配置
```

3、收集用户行为数据

在上一篇文章中，我们完成了业务数据的同步，在推荐系统中另一个必不可少的数据就是用户行为数据，可以说用户行为数据是推荐系统的基石，巧妇难为无米之炊，所以接下来，我们就要将用户的行为数据同步到推荐系统数据库中。

在文章推荐系统中，用户行为包括曝光、点击、停留、收藏、分享等，所以这里我们定义的用户行为数据的字段包括：发生时间（actionTime）、停留时间（readTime）、频道 ID（channelId）、事件名称（action）、用户 ID（userId）、文章 ID（articleId）以及算法 ID（algorithmCombine），这里采用 json 格式，如下所示

```
# 曝光的参数
{"actionTime":"2019-04-10
18:15:35","readTime":"","channelId":0,"param":{"action":
"exposure", "userId": "2", "articleId": "[18577, 14299]",
"algorithmCombine": "C2"}}

# 对文章触发行为的参数
{"actionTime":"2019-04-10
18:15:36","readTime":"","channelId":18,"param":{"action": "click",
"userId": "2", "articleId": "18577", "algorithmCombine": "C2"}}
{"actionTime":"2019-04-10
18:15:38","readTime":"1621","channelId":18,"param":{"action":
"read", "userId": "2", "articleId": "18577", "algorithmCombine":
"C2"}}
{"actionTime":"2019-04-10
18:15:39","readTime":"","channelId":18,"param":{"action": "click",
"userId": "1", "articleId": "14299", "algorithmCombine": "C2"}}
{"actionTime":"2019-04-10
18:15:39","readTime":"","channelId":18,"param":{"action": "click",
"userId": "2", "articleId": "14299", "algorithmCombine": "C2"}}
{"actionTime":"2019-04-10
18:15:41","readTime":"914","channelId":18,"param":{"action":
"read", "userId": "2", "articleId": "14299", "algorithmCombine":
"C2"}}
{"actionTime":"2019-04-10
18:15:47","readTime":"7256","channelId":18,"param":{"action":
"read", "userId": "1", "articleId": "14299", "algorithmCombine":
"C2"}}
```

用户离线行为数据

由于用户行为数据规模庞大，通常是每天更新一次，以供离线计算使用。首先，在 Hive 中创建用户行为数据库 profile 及用户行为表 user_action，设置按照日期进行分区，并匹配 json 格式

```
-- 创建用户行为数据库
create database if not exists profile comment "use action" location
'/user/hive/warehouse/profile.db/';
-- 创建用户行为信息表
create table user_action
(
    actionTime STRING comment "user actions time",
    readTime   STRING comment "user reading time",
    channelId  INT comment "article channel id",
    param map comment "action parameter"
)
    COMMENT "user primitive action"
    PARTITIONED BY (dt STRING) # 按照日期分区
    ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe' # 匹
配json格式
    LOCATION '/user/hive/warehouse/profile.db/user_action';
```

通常用户行为数据被保存在应用服务器的日志文件中，我们可以利用 Flume 监听应用服务器上的日志文件，将用户行为数据收集到 Hive 的 user_action 表对应的 HDFS 目录中，Flume 配置如下

```
a1.sources = s1
a1.sinks = k1
a1.channels = c1

a1.sources.s1.channels= c1
a1.sources.s1.type = exec
a1.sources.s1.command = tail -F /root/logs/userClick.log
a1.sources.s1.interceptors=i1 i2
a1.sources.s1.interceptors.i1.type=regex_filter
a1.sources.s1.interceptors.i1.regex=\\{.*\\}
a1.sources.s1.interceptors.i2.type=timestamp

# c1
a1.channels.c1.type=memory
a1.channels.c1.capacity=30000
a1.channels.c1.transactionCapacity=1000

# k1
a1.sinks.k1.type=hdfs
a1.sinks.k1.channel=c1
a1.sinks.k1.hdfs.path=hdfs://192.168.19.137:9000/user/hive/warehous
e/profile.db/user_action/%Y-%m-%d
a1.sinks.k1.hdfs.useLocalTimeStamp = true
a1.sinks.k1.hdfs.fileType=DataStream
a1.sinks.k1.hdfs.writeFormat=Text
a1.sinks.k1.hdfs.rollInterval=0
a1.sinks.k1.hdfs.rollSize=10240
a1.sinks.k1.hdfs.rollCount=0
a1.sinks.k1.hdfs.idleTimeout=60
```

编写 Flume 启动脚本 collect_click.sh

```
#!/usr/bin/env bash

export JAVA_HOME=/root/bigdata/jdk
export HADOOP_HOME=/root/bigdata/hadoop
export PATH=$PATH:$JAVA_HOME/bin:$HADOOP_HOME/bin

/root/bigdata/flume/bin/flume-ng agent -c /root/bigdata/flume/conf
-f /root/bigdata/flume/conf/collect_click.conf -
Dflume.root.logger=INFO,console -name a1
```

Flume 自动生成目录后，需要手动关联 Hive 分区后才能加载到数据

```
alter table user_action add partition (dt='2019-11-11') location
"/user/hive/warehouse/profile.db/user_action/2011-11-11/"
```

用户实时行为数据

为了提高推荐的实时性，我们也需要收集用户的实时行为数据，以供在线计算使用。这里利用 Flume 将日志收集到 Kafka，在线计算任务可以从 Kafka 读取用户实时行为数据。首先，开启 zookeeper，以守护进程运行

```
/root/bigdata/kafka/bin/zookeeper-server-start.sh -daemon
/root/bigdata/kafka/config/zookeeper.properties
```

开启 Kafka

```
/root/bigdata/kafka/bin/kafka-server-start.sh
/root/bigdata/kafka/config/server.properties

# 开启消息生产者
/root/bigdata/kafka/bin/kafka-console-producer.sh --broker-list
192.168.19.19092 --sync --topic click-trace
# 开启消费者
/root/bigdata/kafka/bin/kafka-console-consumer.sh --bootstrap-
server 192.168.19.137:9092 --topic  click-trace
```

修改 Flume 的日志收集配置文件，添加 c2 和 k2 ，将日志数据收集到 Kafka

```
a1.sources = s1
a1.sinks = k1 k2
a1.channels = c1 c2

a1.sources.s1.channels= c1 c2
a1.sources.s1.type = exec
a1.sources.s1.command = tail -F /root/logs/userClick.log
a1.sources.s1.interceptors=i1 i2
a1.sources.s1.interceptors.i1.type=regex_filter
a1.sources.s1.interceptors.i1.regex=\\{.*\\}
a1.sources.s1.interceptors.i2.type=timestamp

# c1
a1.channels.c1.type=memory
a1.channels.c1.capacity=30000
a1.channels.c1.transactionCapacity=1000

# c2
a1.channels.c2.type=memory
a1.channels.c2.capacity=30000
a1.channels.c2.transactionCapacity=1000

# k1
a1.sinks.k1.type=hdfs
a1.sinks.k1.channel=c1
a1.sinks.k1.hdfs.path=hdfs://192.168.19.137:9000/user/hive/warehous
e/profile.db/user_action/%Y-%m-%d
a1.sinks.k1.hdfs.useLocalTimeStamp = true
a1.sinks.k1.hdfs.fileType=DataStream
a1.sinks.k1.hdfs.writeFormat=Text
a1.sinks.k1.hdfs.rollInterval=0
a1.sinks.k1.hdfs.rollSize=10240
a1.sinks.k1.hdfs.rollCount=0
a1.sinks.k1.hdfs.idleTimeout=60

# k2
a1.sinks.k2.channel=c2
a1.sinks.k2.type=org.apache.flume.supervisorctl
我们可以利用supervisorctl来管理supervisor。sink.kafka.KafkaSink
a1.sinks.k2.kafka.bootstrap.servers=192.168.19.137:9092
a1.sinks.k2.kafka.topic=click-trace
a1.sinks.k2.kafka.batchSize=20
a1.sinks.k2.kafka.producer.requiredAcks=1
```

编写 Kafka 启动脚本 start_kafka.sh

```
#!/usr/bin/env bash
# 启动zookeeper
/root/bigdata/kafka/bin/zookeeper-server-start.sh -daemon
/root/bigdata/kafka/config/zookeeper.properties
# 启动kafka
/root/bigdata/kafka/bin/kafka-server-start.sh
/root/bigdata/kafka/config/server.properties
# 增加topic
/root/bigdata/kafka/bin/kafka-topics.sh --zookeeper
192.168.19.137:2181 --create --replication-factor 1 --topic click-
trace --partitions 1
```

进程管理

我们这里使用 Supervisor 进行进程管理，当进程异常时可以自动重启，Flume 进程配置如下

```
[program:collect-click]
command=/bin/bash /root/toutiao_project/scripts/collect_click.sh
user=root
autorestart=true
redirect_stderr=true
stdout_logfile=/root/logs/collect.log
loglevel=info
stopsignal=KILL
stopasgroup=true
killasgroup=true
```

Kafka 进程配置如下

```
[program:kafka]
command=/bin/bash /root/toutiao_project/scripts/start_kafka.sh
user=root
autorestart=true
redirect_stderr=true
stdout_logfile=/root/logs/kafka.log
loglevel=info
stopsignal=KILL
stopasgroup=true
killasgroup=true
```

启动 Supervisor

```
supervisord -c /etc/supervisord.conf
```

启动 Kafka 消费者，并在应用服务器日志文件中写入日志数据，Kafka 消费者即可收集到实时行为数据

```
# 启动Kafka消费者
/root/bigdata/kafka/bin/kafka-console-consumer.sh --bootstrap-
server 192.168.19.137:9092 --topic  click-trace

# 写入日志数据
echo {\"actionTime\":\"2019-04-10
21:04:39\",\"readTime\":\"\",\"channelId\":18,\"param\":
{\"action\": \"click\", \"userId\": \"2\", \"articleId\":
\"14299\", \"algorithmCombine\": \"C2\"}} >> userClick.log

# 消费者接收到日志数据
{"actionTime":"2019-04-10
21:04:39","readTime":"","channelId":18,"param":{"action": "click",
"userId": "2", "articleId": "14299", "algorithmCombine": "C2"}}
```

Supervisor 常用命令如下

```
supervisorctl

> status              # 查看程序状态
> start apscheduler   # 启动apscheduler单一程序
> stop toutiao:*      # 关闭toutiao组程序
> start toutiao:*     # 启动toutiao组程序
> restart toutiao:*   # 重启toutiao组程序
> update              # 重启配置文件修改过的程序
```

扫一扫关注微信公众号！专注于搜索和推荐系统，尝试使用算法去更好的服务于用户，包括但不局限于机器学习，深度学习，强化学习，自然语言理解，知识图谱，还不定时分享技术，资料，思考等文章！

4、构建离线文章画像

在上述步骤中，我们已经将业务数据和用户行为数据同步到了推荐系统数据库当中，接下来，我们就要对文章数据和用户数据进行分析，构建文章画像和用户画像，本文我们主要讲解如何构建文章画像。文章画像由关键词和主题词组成，我们将每个词的 TF-IDF 权重和 TextRank 权重的乘积作为关键词权重，筛选出权重最高的 K 个词作为关键词，将 TextRank 权重最高的 K 个词与 TF-IDF 权重最高的 K 个词的共现词作为主题词。

首先，在 Hive 中创建文章数据库 article 及相关表，其中表 article_data 用于存储完整的文章信息，表 idf_keywords_values 用于存储关键词和索引信息，表 tfidf_keywords_values 用于存储关键词和 TF-IDF 权重信息，表 textrank_keywords_values 用于存储关键词和 TextRank 权重信息，表 article_profile 用于存储文章画像信息。

```sql
-- 创建文章数据库
create database if not exists article comment "artcile information"
location '/user/hive/warehouse/article.db/';
-- 创建文章信息表
CREATE TABLE article_data
(
    article_id   BIGINT comment "article_id",
    channel_id   INT comment "channel_id",
    channel_name STRING comment "channel_name",
    title        STRING comment "title",
    content      STRING comment "content",
    sentence     STRING comment "sentence"
)
    COMMENT "toutiao news_channel"
    LOCATION '/user/hive/warehouse/article.db/article_data';
-- 创建关键词索引信息表
CREATE TABLE idf_keywords_values
(
    keyword STRING comment "article_id",
    idf     DOUBLE comment "idf",
    index   INT comment "index"
);
-- 创建关键词TF-IDF权重信息表
CREATE TABLE tfidf_keywords_values
(
    article_id INT comment "article_id",
    channel_id INT comment "channel_id",
    keyword    STRING comment "keyword",
    tfidf      DOUBLE comment "tfidf"
);
-- 创建关键词TextRank权重信息表
CREATE TABLE textrank_keywords_values
(
    article_id INT comment "article_id",
    channel_id INT comment "channel_id",
    keyword    STRING comment "keyword",
    textrank   DOUBLE comment "textrank"
);
-- 创建文章画像信息表
CREATE TABLE article_profile
(
    article_id INT comment "article_id",
    channel_id INT comment "channel_id",
    keyword    map comment "keyword",
    topics     array comment "topics"
);
```

计算文章完整信息

为了计算文章画像，需要将文章信息表（news_article_basic）、文章内容表（news_article_content）及频道表（news_channel）进行合并，从而得到完整的文章信息，通常使用 Spark SQL 进行处理。

通过关联表 news_article_basic, news_article_content 和 news_channel 获得文章完整信息，包括 article_id, channel_id, channel_name, title, content，这里获取一个小时内的文章信息。

```
spark.sql("use toutiao")
_now = datetime.today().replace(minute=0, second=0, microsecond=0)
start = datetime.strftime(_now + timedelta(days=0, hours=-1,
minutes=0), "%Y-%m-%d %H:%M:%S")
end = datetime.strftime(_now, "%Y-%m-%d %H:%M:%S")
basic_content = spark.sql(
            "select a.article_id, a.channel_id, a.title, b.content
from news_article_basic a "
            "inner join news_article_content b on
a.article_id=b.article_id where a.review_time >= '{}' "
            "and a.review_time < '{}' and a.status =
2".format(start, end))
basic_content.registerTempTable("temp_article")
channel_basic_content = spark.sql(
            "select t.*, n.channel_name from temp_article t left
join news_channel n on t.channel_id=n.channel_id")
```

`channel_basic_content` 结果如下所示

```
+----------+----------+--------------+--------------------+------------+
|article_id|channel_id|         title|             content|channel_name|
+----------+----------+--------------+--------------------+------------+
|    116636|        18|动态再平衡投资策略历史数据回测|<p>赚钱是个俗气的话题，但又是人...|      python|
+----------+----------+--------------+--------------------+------------+
```

利用 `concat_ws()` 方法，将 channel_name, title, content 这 3 列数据合并为一列 sentence，并将结果写入文章完整信息表 article_data 中

```
import pyspark.sql.functions as F

spark.sql("use article")
sentence_df = channel_basic_content.select("article_id",
"channel_id", "channel_name", "title", "content", \
                                           F.concat_ws(
                                               ",",

channel_basic_content.channel_name,

channel_basic_content.title,

channel_basic_content.content
                                           ).alias("sentence")
                                           )
del basic_content
del channel_basic_content
gc.collect() # 垃圾回收

sentence_df.write.insertInto("article_data")
```

`sentence_df` 结果如下所示，文章完整信息包括 article_id, channel_id, channel_name, title, content, sentence，其中 sentence 为 channel_name, title, content 合并而成的长文本内容

```
+----------+----------+------------+--------------+--------------------+--------------------+
|article_id|channel_id|channel_name|         title|             content|            sentence|
+----------+----------+------------+--------------+--------------------+--------------------+
|    116636|        18|      python|动态再平衡投资策略历史数据回测|<p>赚钱是个俗气的话题，但又是人...|python,动态再平衡投资策略历...|
+----------+----------+------------+--------------+--------------------+--------------------+
```

计算 TF-IDF

前面我们得到了文章的完整内容信息，接下来，我们要先对文章进行分词，然后计算每个词的 TF-IDF 权重，将 TF-IDF 权重最高的 K 个词作为文章的关键词。首先，读取文章信息

```
spark.sql("use article")
article_dataframe = spark.sql("select * from article_data")
```

利用 `mapPartitions()` 方法，对每篇文章进行分词，这里使用的是 `jieba` 分词器

```
words_df =
article_dataframe.rdd.mapPartitions(segmentation).toDF(["article_id
", "channel_id", "words"])

def segmentation(partition):
    import os
    import re
    import jieba
    import jieba.analyse
    import jieba.posseg as pseg
    import codecs

    abspath = "/root/words"

    # 结巴加载用户词典
    userdict_path = os.path.join(abspath, "ITKeywords.txt")
    jieba.load_userdict(userdict_path)

    # 停用词文本
    stopwords_path = os.path.join(abspath, "stopwords.txt")

    def get_stopwords_list():
        """返回stopwords列表"""
        stopwords_list = [i.strip() for i in
codecs.open(stopwords_path).readlines()]
        return stopwords_list

    # 所有的停用词列表
    stopwords_list = get_stopwords_list()

    # 分词
    def cut_sentence(sentence):
        """对切割之后的词语进行过滤，去除停用词，保留名词，英文和自定义词库中的
词，长度大于2的词"""
        seg_list = pseg.lcut(sentence)
        seg_list = [i for i in seg_list if i.flag not in
stopwords_list]
        filtered_words_list = []
        for seg in seg_list:
            if len(seg.word) <= 1:
                continue
            elif seg.flag == "eng":
                if len(seg.word) <= 2:
                    continue
                else:
                    filtered_words_list.append(seg.word)
            elif seg.flag.startswith("n"):
```

```
                filtered_words_list.append(seg.word)
            elif seg.flag in ["x", "eng"]:  # 是自定一个词语或者是英文单
词
                filtered_words_list.append(seg.word)
        return filtered_words_list

    for row in partition:
        sentence = re.sub("<.*?>", "", row.sentence)    # 替换掉标签
数据
        words = cut_sentence(sentence)
        yield row.article_id, row.channel_id, words
```

`words_df` 结果如下所示，words 为将 sentence 分词后的单词列表

```
+----------+----------+--------------------+
|article_id|channel_id|               words|
+----------+----------+--------------------+
|         1|        17|[Vue, props, 用法, ...|
|         2|        17|[vue, 响应式, 原理, mo...|
|         3|        17|[JavaScript, 浅拷贝,...|
|         4|        17|[vue2, vuex, elem...|
|         5|        17|[immutability, Re...|
|         6|        17|[node, npm, cnpm,...|
|         7|        17|[Web, 工程师, 以太坊, 入...|
|         8|        17|[Web, pa, api, we...|
|         9|        17|[vue, 中用, 数据驱动, 视...|
|        10|        17|[程序, WebSocket, 长...|
+----------+----------+--------------------+
```

使用分词结果对词频统计模型（CountVectorizer）进行词频统计训练，并将 CountVectorizer 模型保存到 HDFS 中

```
from pyspark.ml.feature import CountVectorizer
# vocabSize是总词汇的大小，minDF是文本中出现的最少次数
cv = CountVectorizer(inputCol="words", outputCol="countFeatures",
vocabSize=200*10000, minDF=1.0)
# 训练词频统计模型
cv_model = cv.fit(words_df)
cv_model.write().overwrite().save("hdfs://hadoop-
master:9000/headlines/models/CV.model")
```

加载 CountVectorizer 模型，计算词频向量

```
from pyspark.ml.feature import CountVectorizerModel
cv_model = CountVectorizerModel.load("hdfs://hadoop-
master:9000/headlines/models/CV.model")
# 得出词频向量结果
cv_result = cv_model.transform(words_df)
```

`cv_result` 结果如下所示，countFeatures 为词频向量，如 (986, [2, 4, ...], [3.0, 5.0, ...]) 表示总词汇的大小为 986 个，索引为 2 和 4 的词在某篇文章中分别出现 3 次和 5 次，

```
+----------+----------+--------------------+--------------------+
|article_id|channel_id|               words|       countFeatures|
+----------+----------+--------------------+--------------------+
|      4273|        15|[javascript, reac...|(986,[2,4,9,10,11...|
|      4274|        19|[java, java, 笔记, ...|(986,[0,1,8,16,17...|
|      4275|        19|[java, 传统, 方式, 类继...|(986,[1,2,8,16,18...|
|      4276|        15|[javascript, Vue,...|(986,[1,2,4,6,8,1...|
|      4278|        15|[javascript, 作用域链...|(986,[2,3,10,18,2...|
|      4279|        19|[java, springboot...|(986,[1,8,11,23,2...|
|      4280|        19|[java, Jedis, 工具类...|(986,[1,2,4,5,7,8...|
|      4281|        19|[java, java, 记录, ...|(986,[2,16,23,32,...|
|      4282|        15|[javascript, VueS...|(986,[2,4,6,10,16...|
|      4283|        15|[javascript, 体积, ...|(986,[2,3,4,10,11...|
+----------+----------+--------------------+--------------------+
```

得到词频向量后，再利用逆文本频率模型（ IDF ），根据词频向量进行 IDF 统计训练，并将 IDF 模型保存到 HDFS

```
from pyspark.ml.feature import IDF
idf = IDF(inputCol="countFeatures", outputCol="idfFeatures")
idf_model = idf.fit(cv_result)
idf_model.write().overwrite().save("hdfs://hadoop-
master:9000/headlines/models/IDF.model")
```

我们已经分别计算了文章信息中每个词的 TF 和 IDF，这时就可以加载 CountVectorizer 模型和 IDF 模型，计算每个词的 TF-IDF

```
from pyspark.ml.feature import CountVectorizerModel
cv_model = CountVectorizerModel.load("hdfs://hadoop-
master:9000/headlines/models/countVectorizerOfArticleWords.model")
from pyspark.ml.feature import IDFModel
idf_model = IDFModel.load("hdfs://hadoop-
master:9000/headlines/models/IDFOfArticleWords.model")

cv_result = cv_model.transform(words_df)
tfidf_result = idf_model.transform(cv_result)
```

`tfidf_result` 结果如下所示，idfFeatures 为 TF-IDF 权重向量，如 (986, [2, 4, ...], [0.3, 0.5, ...]) 表示总词汇的大小为 986 个，索引为 2 和 4 的词在某篇文章中的 TF-IDF 值分别为 0.3 和 0.5

```
+----------+----------+--------------------+--------------------+--------------------+
|article_id|channel_id|               words|       countFeatures|         idfFeatures|
+----------+----------+--------------------+--------------------+--------------------+
|      4273|        15|[javascript, reac...|(986,[2,4,9,10,11...|(986,[2,4,9,10,11...|
|      4274|        19|[java, java, 笔记, ...|(986,[0,1,8,16,17...|(986,[0,1,8,16,17...|
|      4275|        19|[java, 传统, 方式, 类继...|(986,[1,2,8,16,18...|(986,[1,2,8,16,18...|
|      4276|        15|[javascript, Vue,...|(986,[1,2,4,6,8,1...|(986,[1,2,4,6,8,1...|
|      4278|        15|[javascript, 作用域链...|(986,[2,3,10,18,2...|(986,[2,3,10,18,2...|
|      4279|        19|[java, springboot...|(986,[1,8,11,23,2...|(986,[1,8,11,23,2...|
|      4280|        19|[java, Jedis, 工具类...|(986,[1,2,4,5,7,8...|(986,[1,2,4,5,7,8...|
|      4281|        19|[java, java, 记录, ...|(986,[2,16,23,32,...|(986,[2,16,23,32,...|
|      4282|        15|[javascript, VueS...|(986,[2,4,6,10,16...|(986,[2,4,6,10,16...|
|      4283|        15|[javascript, 体积, ...|(986,[2,3,4,10,11...|(986,[2,3,4,10,11...|
+----------+----------+--------------------+--------------------+--------------------+
```

对文章的每个词都根据 TF-IDF 权重排序，保留 TF-IDF 权重最高的前 K 个词作为关键词

```
def sort_by_tfidf(partition):
    TOPK = 20
    for row in partition:
        # 找到索引与IDF值并进行排序
        _dict = list(zip(row.idfFeatures.indices,
row.idfFeatures.values))
        _dict = sorted(_dict, key=lambda x: x[1], reverse=True)
        result = _dict[:TOPK]
        for word_index, tfidf in result:
            yield row.article_id, row.channel_id, int(word_index),
round(float(tfidf), 4)

keywords_by_tfidf =
tfidf_result.rdd.mapPartitions(sort_by_tfidf).toDF(["article_id",
"channel_id", "index", "weights"])
```

`keywords_by_tfidf` 结果如下所示，每篇文章保留了权重最高的 K 个单词，index 为单词索引，weights 为对应单词的 TF-IDF 权重

```
+----------+----------+-----+-------+
|article_id|channel_id|index|weights|
+----------+----------+-----+-------+
|      4273|        15|    9|66.4852|
|      4273|        15|   12|44.1756|
|      4273|        15|   19|37.5045|
|      4273|        15|   36| 27.276|
|      4273|        15|   37|25.5712|
|      4273|        15|   45|23.8665|
|      4273|        15|   57|18.7522|
|      4273|        15|   35|15.5914|
|      4273|        15|   79|15.3427|
|      4273|        15|   22|14.1624|
|      4273|        15|   87| 13.638|
|      4273|        15|   88| 13.638|
|      4273|        15|   89| 13.638|
|      4273|        15|   90| 13.638|
|      4273|        15|   24|12.6153|
|      4273|        15|   99|11.9332|
|      4273|        15|   64|11.6935|
|      4273|        15|   63|10.3943|
|      4273|        15|   86|10.3943|
|      4273|        15|  123|10.2285|
+----------+----------+-----+-------+
```

接下来，我们需要知道每个词的对应的 TF-IDF 值，可以利用 `zip()` 方法，将所有文章中的每个词及其 TF-IDF 权重组成字典，再加入索引列，由此得到每个词对应的 TF-IDF 值，将该结果保存到 idf_keywords_values 表

```
keywords_list_with_idf = list(zip(cv_model.vocabulary,
idf_model.idf.toArray()))
def append_index(data):
    for index in range(len(data)):
        data[index] = list(data[index]) # 将元组转为list
        data[index].append(index)       # 加入索引
        data[index][1] = float(data[index][1])
append_index(keywords_list_with_idf)
sc = spark.sparkContext
rdd = sc.parallelize(keywords_list_with_idf) # 创建rdd
idf_keywords = rdd.toDF(["keywords", "idf", "index"])

idf_keywords.write.insertInto('idf_keywords_values')
```

`idf_keywords` 结果如下所示，包含了所有单词的名称、TF-IDF 权重及索引

```
pa      0.6651385256756351      1
ul      0.8070591229443697      2
代码    0.7368239176481552      3
方法    0.7506253985501485      4
数据    0.9375297590538404      5
```

通过 index 列，将 `keywords_by_tfidf` 与表 idf_keywords_values 进行连接，选取文章 ID、频道 ID、关键词、TF-IDF 权重作为结果，并保存到 TF-IDF 关键词表 tfidf_keywords_values

```
keywords_index = spark.sql("select keyword, index idx from
idf_keywords_values")
keywords_result = keywords_by_tfidf.join(keywords_index,
keywords_index.idx ==
keywords_by_tfidf.index).select(["article_id", "channel_id",
"keyword", "weights"])
keywords_result.write.insertInto("tfidf_keywords_values")
```

`keywords_result` 结果如下所示，keyword 和 weights 即为所有词在每个文章中的 TF-IDF 权重

```
+----------+----------+-------+--------+
|article_id|channel_id|keyword| weights|
+----------+----------+-------+--------+
|         4|        17|  https|  5.1971|
|         6|        17|  https|  5.1971|
|         8|        17|  https|  2.5986|
|         1|        17|   人工智能|   9.095|
|         2|        17|   人工智能| 74.0591|
|         5|        17|   人工智能| 12.9928|
|         7|        17|   人工智能|  5.1971|
```

计算 TextRank

前面我们已经计算好了每个词的 TF-IDF 权重，为了计算关键词，还需要得到每个词的 TextRank 权重，接下来，还是先读取文章完整信息

```
spark.sql("use article")
article_dataframe = spark.sql("select * from article_data")
```

对文章 sentence 列的内容进行分词，计算每个词的 TextRank 权重，并将每篇文章 TextRank 权重最高的 K 个词保存到 TextRank 结果表 textrank_keywords_values

```
textrank_keywords_df =
article_dataframe.rdd.mapPartitions(textrank).toDF(["article_id",
"channel_id", "keyword", "textrank"])

textrank_keywords_df.write.insertInto("textrank_keywords_values")
```

TextRank 计算细节：分词后只保留指定词性的词，滑动截取长度为 K 的窗口，计算窗口内的各个词的投票数

```
def textrank(partition):
    import os
    import jieba
    import jieba.analyse
    import jieba.posseg as pseg
    import codecs

    abspath = "/root/words"

    # 结巴加载用户词典
    userDict_path = os.path.join(abspath, "ITKeywords.txt")
    jieba.load_userdict(userDict_path)

    # 停用词文本
    stopwords_path = os.path.join(abspath, "stopwords.txt")

    def get_stopwords_list():
        """返回stopwords列表"""
        stopwords_list = [i.strip() for i in
codecs.open(stopwords_path).readlines()]
        return stopwords_list

    # 所有的停用词列表
    stopwords_list = get_stopwords_list()

    class TextRank(jieba.analyse.TextRank):
        def __init__(self, window=20, word_min_len=2):
            super(TextRank, self).__init__()
            self.span = window  # 窗口大小
            self.word_min_len = word_min_len  # 单词的最小长度
            # 要保留的词性，根据jieba github ，具体参见
https://github.com/baidu/lac
```

```
            self.pos_filt = frozenset(
                ('n', 'x', 'eng', 'f', 's', 't', 'nr', 'ns', 'nt',
"nw", "nz", "PER", "LOC", "ORG"))

        def pairfilter(self, wp):
            """过滤条件，返回True或者False"""

            if wp.flag == "eng":
                if len(wp.word) <= 2:
                    return False

            if wp.flag in self.pos_filt and len(wp.word.strip()) >=
self.word_min_len \
                    and wp.word.lower() not in stopwords_list:
                return True
    # TextRank过滤窗口大小为5，单词最小为2
    textrank_model = TextRank(window=5, word_min_len=2)
    allowPOS = ('n', "x", 'eng', 'nr', 'ns', 'nt', "nw", "nz", "c")

    for row in partition:
        tags = textrank_model.textrank(row.sentence, topK=20,
withWeight=True, allowPOS=allowPOS, withFlag=False)
        for tag in tags:
            yield row.article_id, row.channel_id, tag[0], tag[1]
```

`textrank_keywords_df` 结果如下所示，keyword 和 textrank 即为每个单词在文章中的 TextRank 权重

```
+----------+----------+----------+------------------+
|article_id|channel_id|   keyword|          textrank|
+----------+----------+----------+------------------+
|      4273|        15|        pa|               1.0|
|      4273|        15|     class|  0.826167020155012|
|      4273|        15|      hljs| 0.7374857053060796|
|      4273|        15|        组件|0.44124935366123624|
|      4273|        15|     Hooks|0.33877451789133467|
|      4273|        15|        函数| 0.2194471428320325|
```

画像计算

我们计算出 TF-IDF 和 TextRank 后，就可以计算关键词和主题词了，读取 TF-IDF 权重

```
idf_keywords_values = oa.spark.sql("select * from
idf_keywords_values")
```

读取 TextRank 权重

```
textrank_keywords_values = oa.spark.sql("select * from
textrank_keywords_values")
```

通过 `keyword` 关联 TF-IDF 权重和 TextRank 权重

```
keywords_res = textrank_keywords_values.join(idf_keywords_values,
on=['keyword'], how='left')
```

计算 TF-IDF 权重和 TextRank 权重的乘积作为关键词权重

```
keywords_weights = keywords_res.withColumn('weights',
keywords_res.textrank * keywords_res.idf).select(["article_id",
"channel_id", "keyword", "weights"])
```

`keywords_weights` 结果如下所示

```
+----------+----------+---------+-------------------+
|article_id|channel_id|  keyword|            weights|
+----------+----------+---------+-------------------+
|         2|        17|    input| 0.7747667171945103|
|         1|        17|childNode|  1.543308965155555|
|        10|        17|      amp| 0.9526597294643435|
|         3|        17|      jpg| 1.8083185215812947|
```

这里，我们需要将相同文章的词都合并到一条记录中，将 `keywords_weights` 按照 article_id 分组，并利用 `collect_list()` 方法，分别将关键词和权重合并为列表

```
keywords_weights.registerTempTable('temp')

keywords_weights = spark.sql("select article_id, min(channel_id)
channel_id, collect_list(keyword) keywords, collect_list(weights)
weights from temp group by article_id")`
```

`keywords_weights` 结果如下所示，keywords 为每篇文章的关键词列表，weights 为关键词对应的权重列表

```
+----------+----------+--------------------+--------------------+
|article_id|channel_id|            keywords|             weights|
+----------+----------+--------------------+--------------------+
|      4273|        15|[hljs, 逻辑, compon...|[3.69426241356356...|
|      4278|        15|[imageView2, code...|[7.40444917507044...|
|      4275|        19|[Thread, 类继承, cur...|[0.75086108179368...|
|      4281|        19|[调度, 运行时, 中断, 环境,...|[0.94025458334785...|
|      4279|        19|[utf, code, BINDI...|[0.48772006807450...|
|      4283|        15|[hljs, imageView2...|[2.83648176481845...|
|      4282|        15|[imageView2, 文件, ...|[8.31378017714117...|
|      4276|        15|[hljs, code, noop...|[4.01098232733758...|
|      4280|        19|[.h, beans, nnota...|[0.18233072091682...|
|      4274|        19|[Enumeration, Vec...|[2.27602837284537...|
+----------+----------+--------------------+--------------------+
```

为了方便查询，我们需要将关键词和权重合并为一列，并存储为 map 类型，这里利用 `dict()` 和 `zip()` 方法，将每个关键词及其权重组合成字典

```
def to_map(row):
    return row.article_id, row.channel_id, dict(zip(row.keywords,
row.weights))

article_keywords =
keywords_weights.rdd.map(to_map).toDF(['article_id', 'channel_id',
'keywords'])
```

`article_keywords` 结果如下所示，keywords 即为每篇文章的关键词和对应权重

```
+----------+----------+--------------------+
|article_id|channel_id|            keywords|
+----------+----------+--------------------+
|      4273|        15|Map(函数 -> 0.31045...|
|      4278|        15|Map(imageView2 ->...|
|      4275|        19|Map(jvm -> 0.4719...|
|      4281|        19|Map(线程 -> 1.35536...|
|      4279|        19|Map(business -> 0...|
|      4283|        15|Map(imageView2 ->...|
|      4282|        15|Map(imageView2 ->...|
|      4276|        15|Map(viewplus -> 1...|
|      4280|        19|Map(Jedis -> 0.68...|
|      4274|        19|Map(pre -> 0.6368...|
+----------+----------+--------------------+
```

前面我们计算完了关键词，接下来我们将 TF-IDF 和 TextRank 的共现词作为主题词，将 TF-IDF 权重表 tfidf_keywords_values 和 TextRank 权重表 textrank_keywords_values 进行关联，并利用 `collect_set()` 对结果进行去重，即可得到 TF-IDF 和 TextRank 的共现词，即主题词

```
topic_sql = """
                select t.article_id article_id2,
collect_set(t.keyword) topics from tfidf_keywords_values t
                inner join
                textrank_keywords_values r
                where t.keyword=r.keyword
                group by article_id2
                """
article_topics = spark.sql(topic_sql)
```

`article_topics` 结果如下所示，topics 即为每篇文章的主题词列表

```
+----------+--------------------+
|article_id2|              topics|
+----------+--------------------+
|       148|[transform, solid...|
|       463|[clone, 按钮, 空格键, ...|
|       471|[font, DOCTYPE, m...|
|       496|[t02, lock, 线程, n...|
|       833|[modal, close, 属性...|
|      1088|[外边距, 宽度, 内边距, 像素...|
|      1238|[php, 服务员, 语言, 仓库...|
|      1342|[速度, 初速度, canvas,...|
|      1580|[vue, filename, r...|
|      1591|[内容, keywords, li...|
|      1645|[圆角, width, borde...|
|      1829|[mirrors, GCC, gn...|
|      1959|[GNU, openjdk, In...|
|      2122|[样式, CSS+DIV, 背景颜...|
|      2142|[tuple, fromkeys,...|
|      2366|[weights, 大话, &#,...|
|      2659|[pic, 定义, 绝对地址, 超...|
|      2866|[和子, 企业开发, class,...|
|      3175|[stretch, transfo...|
|      3749|[conpiler, callAs...|
+----------+--------------------+
```

最后，将主题词结果和关键词结果合并，即为文章画像，保存到表 `article_profile`

```
article_profile = article_keywords.join(article_topics,
article_keywords.article_id==article_topics.article_id2).select(["a
rticle_id", "channel_id", "keywords", "topics"])

article_profile.write.insertInto("article_profile")
```

文章画像数据查询测试

```
hive> select * from article_profile limit 1;
OK
26      17      {"策
略":0.3973770571351729,"jpg":0.9806348975390871,"用
户":1.2794959063944176,"strong":1.6488457985625076,"文
件":0.28144603583387057,"逻辑":0.45256526469610714,"形
式":0.4123994242601279,"全
自":0.9594604850547191,"h2":0.6244481634710125,"版
本":0.44280276959510817,"Adobe":0.8553618185108718,"安
装":0.8305037437573172,"检查更新":1.8088946300014435,"产
品":0.774842382276899,"下载页":1.4256311032544344,"过
程":0.19827163395829256,"json":0.6423301791599972,"方
式":0.582762869780791,"退出应
用":1.2338671268242603,"Setup":1.004399549339134}   ["Electron","全
自动","产品","版本号","安装包","检查更新","方案","版本","退出应用","逻
辑","安装过程","方式","定性","新版本","Setup","静默","用户"]
Time taken: 0.322 seconds, Fetched: 1 row(s)
```

Apscheduler 定时更新

定义离线更新文章画像的方法，首先合并最近一个小时的文章信息，接着计算每个词的 TF-IDF 和 TextRank 权重，并根据 TF-IDF 和 TextRank 权重计算得出文章关键词和主题词，最后将文章画像信息保存到 Hive

```
def update_article_profile():
    """
    定时更新文章画像
    :return:
    """
    ua = UpdateArticle()
    # 合并文章信息
    sentence_df = ua.merge_article_data()
    if sentence_df.rdd.collect():
        textrank_keywords_df, keywordsIndex =
ua.generate_article_label()
        ua.get_article_profile(textrank_keywords_df, keywordsIndex)
```

利用 Apscheduler 添加定时更新文章画像任务，设定每隔 1 个小时更新一次

```
from apscheduler.schedulers.blocking import BlockingScheduler
from apscheduler.executors.pool import ProcessPoolExecutor

# 创建scheduler,多进程执行
executors = {
    'default': ProcessPoolExecutor(3)
}

scheduler = BlockingScheduler(executors=executors)

# 添加一个定时更新文章画像的任务,每隔1个小时运行一次
scheduler.add_job(update_article_profile, trigger='interval',
hours=1)

scheduler.start()
```

利用 Supervisor 进行进程管理，配置文件如下

```
[program:offline]
environment=JAVA_HOME=/root/bigdata/jdk,SPARK_HOME=/root/bigdata/sp
ark,HADOOP_HOME=/root/bigdata/hadoop,PYSPARK_PYTHON=/miniconda2/env
s/reco_sys/bin/python,PYSPARK_DRIVER_PYTHON=/miniconda2/envs/reco_s
ys/bin/python
command=/miniconda2/envs/reco_sys/bin/python
/root/toutiao_project/scheduler/main.py
directory=/root/toutiao_project/scheduler
user=root
autorestart=true
redirect_stderr=true
stdout_logfile=/root/logs/offlinesuper.log
loglevel=info
stopsignal=KILL
stopasgroup=true
killasgroup=true
```

5、计算文章相似度

在上篇文章中，我们已经完成了离线文章画像的构建，接下来，我们要为相似文章推荐做准备，那就是计算文章之间的相似度。首先，我们要计算出文章的词向量，然后利用文章的词向量来计算文章的相似度。

计算文章词向量

我们可以通过大量的历史文章数据，训练文章中每个词的词向量，由于文章数据过多，通常是分频道进行词向量训练，即每个频道训练一个词向量模型，我们包括的频道如下所示

```
channel_info = {
            1: "html",
            2: "开发者资讯",
            3: "ios",
            4: "c++",
            5: "android",
            6: "css",
            7: "数据库",
            8: "区块链",
            9: "go",
            10: "产品",
            11: "后端",
            12: "linux",
            13: "人工智能",
            14: "php",
            15: "javascript",
            16: "架构",
            17: "前端",
            18: "python",
            19: "java",
            20: "算法",
            21: "面试",
            22: "科技动态",
            23: "js",
            24: "设计",
            25: "数码产品",
        }
```

接下来，分别对各自频道内的文章进行分词处理，这里先选取 18 号频道内的所有文章，进行分词处理

```
spark.sql("use article")
article_data = spark.sql("select * from article_data where
channel_id=18")
words_df =
article_data.rdd.mapPartitions(segmentation).toDF(['article_id',
'channel_id', 'words'])

def segmentation(partition):
    import os
    import re
    import jieba
    import jieba.analyse
    import jieba.posseg as pseg
```

```
    import codecs

    abspath = "/root/words"

    # 结巴加载用户词典
    userDict_path = os.path.join(abspath, "ITKeywords.txt")
    jieba.load_userdict(userDict_path)

    # 停用词文本
    stopwords_path = os.path.join(abspath, "stopwords.txt")

    def get_stopwords_list():
        """返回stopwords列表"""
        stopwords_list = [i.strip() for i in
codecs.open(stopwords_path).readlines()]
        return stopwords_list

    # 所有的停用词列表
    stopwords_list = get_stopwords_list()

    # 分词
    def cut_sentence(sentence):
        """对切割之后的词语进行过滤，去除停用词，保留名词，英文和自定义词库中的
词，长度大于2的词"""
        # eg:[pair('今天', 't'), pair('有', 'd'), pair('雾', 'n'),
pair('霾', 'g')]
        seg_list = pseg.lcut(sentence)
        seg_list = [i for i in seg_list if i.flag not in
stopwords_list]
        filtered_words_list = []
        for seg in seg_list:
            if len(seg.word) <= 1:
                continue
            elif seg.flag == "eng":
                if len(seg.word) <= 2:
                    continue
                else:
                    filtered_words_list.append(seg.word)
            elif seg.flag.startswith("n"):
                filtered_words_list.append(seg.word)
            elif seg.flag in ["x", "eng"]:  # 是自定一个词语或者是英文单
词
                filtered_words_list.append(seg.word)
        return filtered_words_list

    for row in partition:
        sentence = re.sub("<.*?>", "", row.sentence)    # 替换掉标签
```

```
数据
        words = cut_sentence(sentence)
        yield row.article_id, row.channel_id, words
```

`words_df` 结果如下所示，words 为分词后的词语列表

```
+----------+----------+--------------------+
|article_id|channel_id|               words|
+----------+----------+--------------------+
|        18|        17|[web, pa, react, ...|
+----------+----------+--------------------+
```

接着，使用分词后的所有词语，对 Word2Vec 模型进行训练并将模型保存到 HDFS，其中 vectorSize 为词向量的长度，minCount 为词语的最小出现次数，windowSize 为训练窗口的大小，inputCol 为输入的列名，outputCol 为输出的列名

```
from pyspark.ml.feature import Word2Vec

w2v_model = Word2Vec(vectorSize=100, inputCol='words',
outputCol='vector', minCount=3)
model = w2v_model.fit(words_df)
model.save("hdfs://hadoop-
master:9000/headlines/models/word2vec_model/channel_18_python.word2
vec")
```

加载训练好的 Word2Vec 模型

```
from pyspark.ml.feature import Word2VecModel

w2v_model = Word2VecModel.load("hdfs://hadoop-
master:9000/headlines/models/word2vec_model/channel_18_python.word2
vec")
vectors = w2v_model.getVectors()
```

`vectors` 结果如下所示，其中 vector 是训练后的每个词的 100 维词向量，是 vector 类型格式的，如 [0.2 -0.05 -0.1 ...]

```
+-----------------+--------------------+
|             word|              vector|
+-----------------+--------------------+
|               广义|[0.28907623887062...|
|               钟爱|[-0.0529650673270...|
|c1c3387c24028915fc|[0.08250344544649...|
|          failCnt0|[-0.0034321683924...|
|       freeman1974|[0.01132440567016...|
|               伙伴|[-0.1075697541236...|
|  testStationarity|[0.09605087339878...|
|               箭头|[0.08957882970571...|
|        fieldsfrom|[-0.0121747571974...|
|      RoundrobinLB|[-0.0941602289676...|
|              COCO|[-0.2620599269866...|
|              拜拜|[-0.0820834264159...|
|          quotient|[0.08328679203987...|
|              货币|[-0.0276580695062...|
|              人物|[-0.1581292450428...|
|             wsy|[0.09347543865442...|
|           serious|[0.01220745686441...|
|              跨进程|[-0.0330988727509...|
|        fromParams|[0.00353708816692...|
|        MongoDB数据库|[-0.1521783322095...|
+-----------------+--------------------+
```

这里，我们计算出了所有词语的词向量，接下来，还要得到关键词的词向量，因为我们需要通过关键词的词向量来计算文章的词向量。那么，首先通过读取频道内的文章画像来得到关键词（实际场景应该只读取新增文章画像）

```
article_profile = spark.sql("select * from article_profile where
channel_id=18")
```

在文章画像表中，关键词和权重是存储在同一列的，我们可以利用 `LATERAL VIEW explode()` 方法，将 map 类型的 keywords 列中的关键词和权重转换成单独的两列数据

```
article_profile.registerTempTable('profile')
keyword_weight = spark.sql("select article_id, channel_id, keyword,
weight from profile LATERAL VIEW explode(keywords) AS keyword,
weight")
```

`keyword_weight` 结果如下所示，keyword 为关键词，weight 为对应的权重

```
+----------+----------+--------+-------------------+
|article_id|channel_id| keyword|             weight|
+----------+----------+--------+-------------------+
|     13098|        18|    repr| 0.6326590117716192|
|     13098|        18|      __| 2.5401122038114203|
|     13098|        18|      属性|0.23645924932468856|
|     13098|        18|     pre| 0.6040062287555379|
|     13098|        18|    code| 0.9531379029975557|
|     13098|        18|     def| 0.5063435861497416|
|     13098|        18|   color| 1.1337936117177925|
|     13098|        18|      定义| 0.1554380122061322|
|     13098|        18| Student| 0.5033771372284416|
|     13098|        18|getPrice| 0.7404427038950527|
|     13098|        18|      方法|0.08080845613717194|
|     13098|        18|     div| 0.3434819820586186|
|     13098|        18|     str|0.35999033790156054|
|     13098|        18|      pa| 0.6651385256756351|
|     13098|        18|   slots| 0.6992789472129189|
|     13098|        18| cnblogs|0.33926586102013295|
|     13098|        18|      函数|0.15015578405898256|
|     13098|        18|   style| 2.4777013955852873|
|     13098|        18|      &#| 0.4911011561534254|
|     13098|        18|   class|0.28891320463243075|
+----------+----------+--------+-------------------+
```

这时就可以利用关键词 keyword 列，将文章关键词 `keyword_weight` 与词向量结果 `vectors` 进行内连接，从而得到每个关键词的词向量

```
keywords_vector = keyword_weight.join(vectors,
vectors.word==keyword_weight.keyword, 'inner')
```

`keywords_vector` 结果如下所示，vector 即对应关键词的 100 维词向量

```
+----------+----------+-------+------------------+------+--------------------+
|article_id|channel_id|keyword|            weight|  word|              vector|
+----------+----------+-------+------------------+------+--------------------+
|     12936|        18| strong|  4.705249850191452|strong|[0.06607607007026...|
|     12936|        18| python| 1.5221131438694515|python|[-0.0696719884872...|
|     12936|        18|   bool|  1.698618363235006|  bool|[0.08196849375963...|
|     12936|        18|   编程语言| 1.0816836469752635|  编程语言|[-0.1638689935207...|
|     12936|        18|   True| 1.1097628840375606|  True|[0.24163006246089...|
|     12936|        18|     遗漏|   1.63385196709268|    遗漏|[-0.2000092417001...|
|     12936|        18|   elif| 2.0815467371010805|  elif|[0.23405943810939...|
|     12936|        18|    int|   0.69289212656367|   int|[-0.0279460791498...|
|     12936|        18|    str| 1.1603775123519366|   str|[-0.0372486524283...|
|     12936|        18|     pa|0.36743860272647505|    pa|[0.12457559257745...|
|     12936|        18|     单词|  1.406335008823204|    单词|[-0.0355367287993...|
|     12936|        18|     数据| 0.2990961325092939|    数据|[0.14358098804950...|
|     12936|        18|     过程| 0.4050493263937482|    过程|[-0.0100963125005...|
|     12936|        18|  if语句|  1.957626628186806|  if语句|[0.21955621242523...|
|     12936|        18|     大写| 1.7154369266840954|    大写|[0.03721550852060...|
|     12936|        18|  style| 1.6059537337711527| style|[0.25782978534698...|
|     12936|        18|   命名规范| 1.9782233064408097|  命名规范|[0.06696650385856...|
|     12936|        18|     字母| 1.2148323436339508|    字母|[0.24973358213901...|
|     12936|        18|     重点| 0.9063778617582984|    重点|[0.06429161876440...|
|     12936|        18|   font| 1.7792174100662055|  font|[0.66760659217834...|
+----------+----------+-------+------------------+------+--------------------+
```

接下来，将文章每个关键词的词向量加入权重信息，这里使每个关键词的词向量 = 关键词的权重 x 关键词的词向量，即 weight_vector = weight x vector，注意这里的 `vector` 为 vector 类型，所以 weight x vector 是权重和向量的每个元素相乘，向量的长度保持不变

```
def compute_vector(row):
    return row.article_id, row.channel_id, row.keyword, row.weight
* row.vector

article_keyword_vectors =
keywords_vector.rdd.map(compute_vector).toDF(["article_id",
"channel_id", "keyword", "weightingVector"])
```

`article_keyword_vectors` 结果如下所示，weightingVector 即为加入权重信息后的关键词的词向量

```
+----------+----------+--------+--------------------+
|article_id|channel_id| keyword|     weightingVector|
+----------+----------+--------+--------------------+
|     13098|        18|    repr|[0.13308673769053...|
|     13098|        18|      __|[0.03018926765933...|
|     13098|        18|      属性|[-0.0191257052454...|
|     13098|        18|     pre|[0.34502730264365...|
|     13098|        18|    code|[0.34601303844503...|
|     13098|        18|     def|[0.07761504849378...|
|     13098|        18|   color|[0.61312345712161...|
|     13098|        18|      定义|[-0.0010762159703...|
|     13098|        18| Student|[0.09441257176805...|
|     13098|        18|getPrice|[-0.0847735848446...|
|     13098|        18|      方法|[-0.0048283701284...|
|     13098|        18|     div|[0.04037546778136...|
|     13098|        18|     str|[-0.0134091549740...|
|     13098|        18|      pa|[0.08286002598213...|
|     13098|        18|   slots|[-0.2270685226558...|
|     13098|        18| cnblogs|[0.02879135118511...|
|     13098|        18|      函数|[0.01213366982724...|
|     13098|        18|   style|[0.63882521897767...|
|     13098|        18|      &#|[-0.0073760822316...|
|     13098|        18|   class|[-0.0168053401172...|
+----------+----------+--------+--------------------+
```

再将上面的结果按照 article_id 进行分组，利用 `collect_set()` 方法，将一篇文章内所有关键词的词向量合并为一个列表

```
article_keyword_vectors.registerTempTable('temptable')
article_keyword_vectors = spark.sql("select article_id,
min(channel_id) channel_id, collect_set(weightingVector) vectors
from temptable group by article_id")
```

`article_keyword_vectors` 结果如下所示，vectors 即为文章内所有关键词向量的列表，如 [[0.6 0.2 ...], [0.1 -0.07 ...], ...]

```
+----------+----------+--------------------+
|article_id|channel_id|             vectors|
+----------+----------+--------------------+
|     13098|        18|[[0.6131234571216...|
|     13248|        18|[[2.5336782538801...|
|     13401|        18|[[0.1199601801400...|
|     13723|        18|[[0.0767033252676...|
|     14719|        18|[[-0.091019116457...|
|     14846|        18|[[-0.069822823712...|
|     15173|        18|[[-0.359728582063...|
|     15194|        18|[[-0.006211698526...|
|     15237|        18|[[-0.023426829636...|
|     15322|        18|[[0.0780172035455...|
+----------+----------+--------------------+
```

接下来，利用上面得出的二维列表，计算每篇文章内所有关键词的词向量的平均值，作为文章的词向量。注意，这里的 `vectors` 是包含多个词向量的列表，词向量列表的平均值等于其中每个词向量的对应元素相加再除以词向量的个数

```
def compute_avg_vectors(row):
    x = 0
    for i in row.vectors:
        x += i
    # 求平均值
    return row.article_id, row.channel_id, x / len(row.vectors)

article_vector =
article_keyword_vectors.rdd.map(compute_avg_vectors).toDF(['article
_id', 'channel_id', 'vector'])
```

`article_vector` 结果如下所示

```
+----------+----------+--------------------+
|article_id|channel_id|              vector|
+----------+----------+--------------------+
|     12936|        18|[0.15785672885486...|
|     13206|        18|[0.36658417091938...|
|     14029|        18|[0.08382564595017...|
|     14259|        18|[-0.1518681660977...|
|     14805|        18|[0.11028526511434...|
|     15921|        18|[0.10679691438887...|
|     17370|        18|[0.08871408187970...|
|     17595|        18|[0.21126698350251...|
|     18026|        18|[0.32436757418235...|
|     18117|        18|[-0.1335141347559...|
+----------+----------+--------------------+
```

此时，`article_vector` 中的 `vector` 列还是 vector 类型，而 Hive 不支持该数据类型，所以需要将 vector 类型转成 array 类型（list）

```
def to_list(row):
    return row.article_id, row.channel_id, [float(i) for i in
row.vector.toArray()]

article_vector =
article_vector.rdd.map(to_list).toDF(['article_id', 'channel_id',
'vector'])
```

在 Hive 中创建文章词向量表 article_vector

```
CREATE TABLE article_vector
(
    article_id INT comment "article_id",
    channel_id INT comment "channel_id",
    articlevector ARRAY comment "keyword"
);
```

最后，将 18 号频道内的所有文章的词向量存储到 Hive 的文章词向量表 article_vector 中

```
article_vector.write.insertInto("article_vector")
```

这样，我们就计算出了 18 号频道下每篇文章的词向量，在实际场景中，我们还要分别计算出其他所有频道下每篇文章的词向量。

计算文章相似度

前面我们计算出了文章的词向量，接下来就可以根据文章的词向量来计算文章的相似度了。通常我们会有几百万、几千万甚至上亿规模的文章数据，为了优化计算性能，我们可以只计算每个频道内文章之间的相似度，因为通常只有相同频道的文章关联性较高，而不同频道之间的文章通常关联性较低。在每个频道内，我们还可以用聚类或局部敏感哈希对文章进行分桶，将文章相似度的计算限制在更小的范围，只计算相同分类内或相同桶内的文章相似度。

- 聚类（Clustering），对每个频道内的文章进行聚类，可以使用 KMeans 算法，需要提前设定好类别个数 K，聚类算法的时间复杂度并不小，也可以使用一些优化的聚类算法，比如二分聚类、层次聚类等。但通常聚类算法也比较耗时，所以通常被使用更多的是局部敏感哈希。

Spark 的 BisectingKMeans 模型训练代码示例

```
from pyspark.ml.clustering import BisectingKMeans

bkmeans = BisectingKMeans(k=100, minDivisibleClusterSize=50,
featuresCol="articlevector", predictionCol='group')
bkmeans_model = bkmeans.fit(article_vector)
bkmeans_model.save("hdfs://hadoop-
master:9000/headlines/models/articleBisKmeans/channel_%d_%s.bkmeans
" % (channel_id, channel))
```

- 局部敏感哈希 LSH（Locality Sensitive Hashing），LSH 算法是基于一个假设，如果两个文本在原有的数据空间是相似的，那么经过哈希函数转换以后，它们仍然具有很高的相似度，即越相似的文本在哈希之后，落到相同的桶内的概率就越高。所以，我们只需要将目标文章进行哈希映射并得到其桶号，然后取出该桶内的所有文章，再进行线性匹配即可查找到与目标文章相邻的文章。其实 LSH 并不能保证一定能够查找到与目标文章最相邻的文章，而是在减少需要匹配的文章个数的同时，保证查找到最近邻的文章的概率很大。

下面我们将使用 LSH 模型来计算文章相似度，首先，读取 18 号频道内所有文章的 ID 和词向量作为训练集

```
article_vector = spark.sql("select article_id, articlevector from
article_vector where channel_id=18")
train = articlevector.select(['article_id', 'articlevector'])
```

文章词向量表中的词向量是被存储为 array 类型的，我们利用 Spark 的 `Vectors.dense()` 方法，将 array 类型（list）转为 vector 类型

```
from pyspark.ml.linalg import Vectors

def list_to_vector(row):
    return row.article_id, Vectors.dense(row.articlevector)

train = train.rdd.map(list_to_vector).toDF(['article_id',
'articlevector'])
```

使用训练集 `train` 对 Spark 的 `BucketedRandomProjectionLSH` 模型进行训练，其中 inputCol 为输入特征列，outputCol 为输出特征列，numHashTables 为哈希表数量，bucketLength 为桶的数量，数量越多，相同数据进入到同一个桶的概率就越高

```
from pyspark.ml.feature import BucketedRandomProjectionLSH

brp = BucketedRandomProjectionLSH(inputCol='articlevector',
outputCol='hashes', numHashTables=4.0, bucketLength=10.0)
model = brp.fit(train)
```

训练好模型后，调用 `approxSimilarityJoin()` 方法即可计算数据之间的相似度，如 `model.approxSimilarityJoin(df1, df2, 2.0, distCol='EuclideanDistance')` 就是利用欧几里得距离作为相似度，计算在 df1 与 df2 每条数据的相似度，这里我们计算训练集中所有文章之间的相似度

```
similar = model.approxSimilarityJoin(train, train, 2.0,
distCol='EuclideanDistance')
```

`similar` 结果如下所示，EuclideanDistance 就是两篇文章的欧几里得距离，即相似度

```
+--------------------+--------------------+------------------+
|            datasetA|            datasetB| EuclideanDistance|
+--------------------+--------------------+------------------+
|[17595,[0.2112669...|[17595,[0.2112669...|               0.0|
|[12936,[0.1578567...|[17370,[0.0887140...| 1.361573657055773|
|[15921,[0.1067969...|[14805,[0.1102852...|0.7717752375745968|
|[15921,[0.1067969...|[12936,[0.1578567...|1.5205099193938123|
|[15921,[0.1067969...|[14259,[-0.151868...|  1.86883894754822|
|[13206,[0.3665841...|[13206,[0.3665841...|               0.0|
|[14029,[0.0838256...|[17595,[0.2112669...|1.9271039464531845|
|[12936,[0.1578567...|[12936,[0.1578567...|               0.0|
|[12936,[0.1578567...|[18117,[-0.133514...|1.8511391009238651|
|[18117,[-0.133514...|[18117,[-0.133514...|               0.0|
|[18117,[-0.133514...|[14805,[0.1102852...|1.4227262480154705|
|[17370,[0.0887140...|[15921,[0.1067969...|0.8802763327501671|
|[14029,[0.0838256...|[14029,[0.0838256...|               0.0|
|[17595,[0.2112669...|[14029,[0.0838256...|1.9271039464531845|
|[14805,[0.1102852...|[12936,[0.1578567...|1.3378357895763942|
|[14805,[0.1102852...|[14029,[0.0838256...|1.8215851677420467|
|[17370,[0.0887140...|[14029,[0.0838256...|1.9964285103159445|
|[14259,[-0.151868...|[17370,[0.0887140...|1.8488428360491858|
|[14805,[0.1102852...|[18117,[-0.133514...|1.4227262480154705|
|[18117,[-0.133514...|[17370,[0.0887140...|1.5285416639087503|
+--------------------+--------------------+------------------+
```

在后面的推荐流程中，会经常查询文章相似度，所以出于性能考虑，我们选择将文章相似度结果存储到 Hbase 中。首先创建文章相似度表

```
create 'article_similar', 'similar'
```

然后存储文章相似度结果

```
def save_hbase(partition):
    import happybase
    pool = happybase.ConnectionPool(size=3, host='hadoop-master')

    with pool.connection() as conn:
        # 建立表连接
        table = conn.table('article_similar')
        for row in partition:
            if row.datasetA.article_id != row.datasetB.article_id:
                table.put(str(row.datasetA.article_id).encode(),
{"similar:{}".format(row.datasetB.article_id).encode(): b'%0.4f' %
(row.EuclideanDistance)})

        # 手动关闭所有的连接
        conn.close()

similar.foreachPartition(save_hbase)
```

Apscheduler 定时更新

将文章相似度计算加入到文章画像更新方法中，首先合并最近一个小时的文章完整信息，接着计算 TF-IDF 和 TextRank 权重，并根据 TF-IDF 和 TextRank 权重计算得出关键词和主题词，最后计算文章的词向量及文章的相似度

```
def update_article_profile():
    """
    定时更新文章画像及文章相似度
    :return:
    """
    ua = UpdateArticle()
    sentence_df = ua.merge_article_data()
    if sentence_df.rdd.collect():
        textrank_keywords_df, keywordsIndex =
ua.generate_article_label()
        article_profile =
ua.get_article_profile(textrank_keywords_df, keywordsIndex)
        ua.compute_article_similar(article_profile)
```

扫一扫关注微信公众号！专注于搜索和推荐系统，尝试使用算法去更好的服务于用户，包括但不局限于机器学习，深度学习，强化学习，自然语言理解，知识图谱，还不定时分享技术，资料，思考等文章！

6、构建离线用户画像

前面我们完成了文章画像的构建以及文章相似度的计算，接下来，我们就要实现用户画像的构建了。用户画像往往是大型网站的重要模块，基于用户画像不仅可以实现个性化推荐，还可以实现用户分群、精准推送、精准营销以及用户行为预测、商业化转化分析等，为商业决策提供数据支持。通常用户画像包括用户属性信息（性别、年龄、出生日期等）、用户行为信息（浏览、收藏、点赞等）以及环境信息（时间、地理位置等）。

处理用户行为数据

在数据准备阶段，我们通过 Flume 已经可以将用户行为数据收集到 Hive 的 user_action 表的 HDFS 路径中，先来看一下这些数据长什么样子，我们读取当天的用户行为数据，注意读取之前要先关联分区

```
_day = time.strftime("%Y-%m-%d", time.localtime())
_locations = '/user/hive/warehouse/profile.db/user_action/' + _day
if fs.exists(_locations):
    # 如果有该文件直接关联，捕获关联重复异常
    try:
        self.spark.sql("alter table user_action add partition
(dt='%s') location '%s'" % (_day, _locations))
    except Exception as e:
        pass

    self.spark.sql("use profile")
    user_action = self.spark.sql("select actionTime, readTime,
channelId, param.articleId, param.algorithmCombine, param.action,
param.userId from user_action where dt>=" + _day)
```

`user_action` 结果如下所示

actionTime	readTime	channelId	articleId	algorithmCombine	action	userId
2019-04-09 07:13:25		0	[15716, 19171, 13...	C2	exposure	1103195673450250240
2019-04-09 07:13:49		18	19171	C2	click	1103195673450250240
2019-04-09 07:14:02	12238	18	19171	C2	read	1103195673450250240
2019-04-09 07:14:04		18	13797	C2	click	1103195673450250240
2019-04-09 07:14:27	21820	18	13797	C2	read	1103195673450250240
2019-04-09 07:13:25		0	[15716, 19171, 13...	C2	exposure	1103195673450250240
2019-04-09 07:13:49		18	19171	C2	click	1103195673450250240
2019-04-09 07:14:02	12238	18	19171	C2	read	1103195673450250240
2019-04-09 07:14:04		18	13797	C2	click	1103195673450250240
2019-04-09 07:14:27	21820	18	13797	C2	read	1103195673450250240
2019-04-09 07:13:25		0	[15716, 19171, 13...	C2	exposure	1103195673450250240
2019-04-09 07:13:49		18	19171	C2	click	1103195673450250240
2019-04-09 07:14:02	12238	18	19171	C2	read	1103195673450250240
2019-04-09 07:14:04		18	13797	C2	click	1103195673450250240
2019-04-09 07:14:27	21820	18	13797	C2	read	1103195673450250240
2019-04-09 07:13:25		0	[15716, 19171, 13...	C2	exposure	1103195673450250240
2019-04-09 07:13:49		18	19171	C2	click	1103195673450250240
2019-04-09 07:14:02	12238	18	19171	C2	read	1103195673450250240
2019-04-09 07:14:04		18	13797	C2	click	1103195673450250240
2019-04-09 07:14:27	21820	18	13797	C2	read	1103195673450250240

可以发现，上面的一条记录代表用户对文章的一次行为，但通常我们需要查询某个用户对某篇文章的所有行为，所以，我们要将这里用户对文章的多条行为数据合并为一条，其中包括用户对文章的所有行为。我们需要新建一个 Hive 表 user_article_basic，这张表包括了用户 ID、文章 ID、是否曝光、是否点击、阅读时间等等，随后我们将处理好的用户行为数据存储到此表中

```
create table user_article_basic
(
    user_id     BIGINT comment "userID",
    action_time STRING comment "user actions time",
    article_id  BIGINT comment "articleid",
    channel_id  INT comment "channel_id",
    shared      BOOLEAN comment "is shared",
    clicked     BOOLEAN comment "is clicked",
    collected   BOOLEAN comment "is collected",
    exposure    BOOLEAN comment "is exposured",
    read_time   STRING comment "reading time"
)
    COMMENT "user_article_basic"
    CLUSTERED by (user_id) into 2 buckets
    STORED as textfile
    LOCATION '/user/hive/warehouse/profile.db/user_article_basic';
```

遍历每一条原始用户行为数据，判断用户对文章的行为，在 user_action_basic 中将该用户与该文章对应的行为设置为 True

```
if user_action.collect():
    def _generate(row):
        _list = []
        if row.action == 'exposure':
            for article_id in eval(row.articleId):
                # ["user_id", "action_time","article_id",
"channel_id", "shared", "clicked", "collected", "exposure",
"read_time"]
                _list.append(
                    [row.userId, row.actionTime, article_id,
row.channelId, False, False, False, True, row.readTime])
            return _list
        else:
            class Temp(object):
                shared = False
                clicked = False
                collected = False
                read_time = ""

            _tp = Temp()
            if row.action == 'click':
                _tp.clicked = True
            elif row.action == 'share':
                _tp.shared = True
            elif row.action == 'collect':
                _tp.collected = True
            elif row.action == 'read':
                _tp.clicked = True

            _list.append(
                [row.userId, row.actionTime, int(row.articleId),
row.channelId, _tp.shared, _tp.clicked, _tp.collected,
                 True, row.readTime])
            return _list

    user_action_basic = user_action.rdd.flatMap(_generate)
    user_action_basic = user_action_basic.toDF(
        ["user_id", "action_time", "article_id", "channel_id",
"shared", "clicked", "collected", "exposure",
         "read_time"])
```

`user_action_basic` 结果如下所示，这里的一条记录包括了某个用户对某篇文章的所有行为

user_id	action_time	article_id	channel_id	shared	clicked	collected	exposure	read_time
1103195673450250240	2019-04-09 07:13:25	15716	0	false	false	false	true	
1103195673450250240	2019-04-09 07:13:25	19171	0	false	false	false	true	
1103195673450250240	2019-04-09 07:13:25	13797	0	false	false	false	true	
1103195673450250240	2019-04-09 07:13:25	17511	0	false	false	false	true	
1103195673450250240	2019-04-09 07:13:25	18795	0	false	false	false	true	
1103195673450250240	2019-04-09 07:13:25	18156	0	false	false	false	true	
1103195673450250240	2019-04-09 07:13:25	43885	0	false	false	false	true	
1103195673450250240	2019-04-09 07:13:25	13167	0	false	false	false	true	
1103195673450250240	2019-04-09 07:13:25	13039	0	false	false	false	true	
1103195673450250240	2019-04-09 07:13:25	18038	0	false	false	false	true	
1103195673450250240	2019-04-09 07:13:49	19171	18	false	true	false	true	
1103195673450250240	2019-04-09 07:14:02	19171	18	false	true	false	true	12238
1103195673450250240	2019-04-09 07:14:04	13797	18	false	true	false	true	
1103195673450250240	2019-04-09 07:14:27	13797	18	false	true	false	true	21820
1103195673450250240	2019-04-09 07:13:25	15716	0	false	false	false	true	
1103195673450250240	2019-04-09 07:13:25	19171	0	false	false	false	true	
1103195673450250240	2019-04-09 07:13:25	13797	0	false	false	false	true	
1103195673450250240	2019-04-09 07:13:25	17511	0	false	false	false	true	
1103195673450250240	2019-04-09 07:13:25	18795	0	false	false	false	true	
1103195673450250240	2019-04-09 07:13:25	18156	0	false	false	false	true	

由于 Hive 目前还不支持 pyspark 的原子性操作，所以 user_article_basic 表的用户行为数据只能全量更新（实际场景中可以选择其他语言或数据库实现）。这里，我们需要将当天的用户行为与 user_action_basic 的历史用户行为进行合并

```
old_data = uup.spark.sql("select * from user_article_basic")
new_data = old_data.unionAll(user_action_basic)
```

合并后又会产生一个新的问题，那就是用户 ID 和文章 ID 可能重复，因为今天某个用户对某篇文章的记录可能在历史数据中也存在，而 `unionAll()` 方法并没有去重，这里我们可以按照用户 ID 和文章 ID 进行分组，利用 `max()` 方法得到 action_time, channel_id, shared, clicked, collected, exposure, read_time 即可，去重后直接存储到 user_article_basic 表中

```
new_data.registerTempTable("temptable")

self.spark.sql('''insert overwrite table user_article_basic select
user_id, max(action_time) as action_time,
        article_id, max(channel_id) as channel_id, max(shared) as
shared, max(clicked) as clicked,
        max(collected) as collected, max(exposure) as exposure,
max(read_time) as read_time from temptable
        group by user_id, article_id''')
```

表 user_article_basic 结果如下所示

```
+------------------+-------------------+------------------+----------+------+-------+---------+--------+---------+
|           user_id|        action_time|        article_id|channel_id|shared|clicked|collected|exposure|read_time|
+------------------+-------------------+------------------+----------+------+-------+---------+--------+---------+
|1114863735962337280|2019-04-07 20:13:23|1112608068731928576|         0| false|  false|    false|    true|         |
|1114863735962337280|2019-04-07 20:13:23|1112593242529988608|         0| false|  false|    false|    true|         |
|1114863735962337280|2019-04-07 20:13:23|1112566345800613888|         0| false|  false|    false|    true|         |
|1114863735962337280|2019-04-07 20:13:23|1112593324574769152|         0| false|  false|    false|    true|         |
|1114863735962337280|2019-04-07 20:13:23|1112592065390182400|         0| false|  false|    false|    true|         |
|1114863735962337280|2019-04-07 20:13:23|            141440|         0| false|  false|    false|    true|         |
|1114863735962337280|2019-04-07 20:13:23|1112525856586072064|         0| false|  false|    false|    true|         |
|1114863735962337280|2019-04-07 20:13:23|1109326351522856960|         0| false|  false|    false|    true|         |
|1114863735962337280|2019-04-07 20:13:23|1108924834420621312|         0| false|  false|    false|    true|         |
|1114863741448486912|2019-04-07 20:13:24|1112608068731928576|         0| false|  false|    false|    true|         |
|1114863741448486912|2019-04-07 20:13:24|1112593242529988608|         0| false|  false|    false|    true|         |
|1114863741448486912|2019-04-07 20:13:24|1112566345800613888|         0| false|  false|    false|    true|         |
|1114863741448486912|2019-04-07 20:13:24|1112593324574769152|         0| false|  false|    false|    true|         |
|1114863741448486912|2019-04-07 20:13:24|1112592065390182400|         0| false|  false|    false|    true|         |
|1114863741448486912|2019-04-07 20:13:24|            141440|         0| false|  false|    false|    true|         |
|1114863741448486912|2019-04-07 20:13:24|1112525856586072064|         0| false|  false|    false|    true|         |
|1114863741448486912|2019-04-07 20:13:24|1109326351522856960|         0| false|  false|    false|    true|         |
|1114863741448486912|2019-04-07 20:13:24|1108924834420621312|         0| false|  false|    false|    true|         |
|1114863748553637888|2019-04-07 20:13:26|1112608068731928576|         0| false|  false|    false|    true|         |
|1114863748553637888|2019-04-07 20:13:26|1112593242529988608|         0| false|  false|    false|    true|         |
+------------------+-------------------+------------------+----------+------+-------+---------+--------+---------+
```

计算用户画像

我们选择将用户画像存储在 Hbase 中，因为 Hbase 支持原子性操作和快速读取，并且 Hive 也可以通过创建外部表关联到 Hbase，进行离线分析，如果要删除 Hive 外部表的话，对 Hbase 也没有影响。首先，在 Hbase 中创建用户画像表

```
create 'user_profile', 'basic','partial','env'
```

在 Hive 中创建 Hbase 外部表，注意字段类型设置为 map

```
create external table user_profile_hbase
(
    user_id         STRING comment "userID",
    information     MAP<STRING, DOUBLE> comment "user basic
information",
    article_partial MAP<STRING, DOUBLE> comment "article partial",
    env             MAP<STRING, INT> comment "user env"
)
    COMMENT "user profile table"
    STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
        WITH SERDEPROPERTIES ("hbase.columns.mapping" =
":key,basic:,partial:,env:")
    TBLPROPERTIES ("hbase.table.name" = "user_profile");
```

创建外部表之后，还需要导入一些依赖包

```
cp -r /root/bigdata/hbase/lib/hbase-*.jar /root/bigdata/spark/jars/
cp -r /root/bigdata/hive/lib/h*.jar /root/bigdata/spark/jars/
```

接下来，读取处理好的用户行为数据，由于日志中的 channel_id 有可能是来自于推荐频道（0），而不是文章真实的频道，所以这里要将 channel_id 列删除

```
spark.sql("use profile")
user_article_basic = spark.sql("select * from
user_article_basic").drop('channel_id')
```

通过文章 ID，将用户行为数据与文章画像数据进行连接，从而得到文章频道 ID 和文章主题词

```
spark.sql('use article')
article_topic = spark.sql("select article_id, channel_id, topics
from article_profile")
user_article_topic = user_article_basic.join(article_topic,
how='left', on=['article_id'])
```

`user_article_topic` 结果如下图所示，其中 topics 列即为文章主题词列表，如 ['补码', '字符串', '李白', ...]

article_id	user_id	action_time	shared	clicked	collected	exposure	read_time	channel_id	topics
13401	10	2019-03-06 10:06:12	false	false	false	true		18	[补码, 字符串, 李白, typ...]
13401	1114864237131333632	2019-04-09 16:39:51	false	false	false	true		18	[补码, 字符串, 李白, typ...]
13401	1106396183141548032	2019-03-28 10:58:20	false	false	false	true		18	[补码, 字符串, 李白, typ...]
13401	1109994594201763840	2019-03-26 15:03:58	false	false	false	true		18	[补码, 字符串, 李白, typ...]
14805	1105045287866466304	2019-03-11 18:15:48	false	false	false	true		18	[占位符, Code, sep, ...]
14805	1114865875103514624	2019-04-09 16:44:09	false	false	false	true		18	[占位符, Code, sep, ...]
14805	1	2019-03-05 17:34:03	false	false	false	true		18	[占位符, Code, sep, ...]
14805	1113004557979353088	2019-04-04 08:31:44	false	false	false	true		18	[占位符, Code, sep, ...]
14805	1115534909935452160	2019-04-09 16:46:37	false	false	false	true		18	[占位符, Code, sep, ...]
14805	1115089292662669312	2019-04-09 16:39:49	false	false	false	true		18	[占位符, Code, sep, ...]
14805	1114864434305564672	2019-04-09 16:42:37	false	false	false	true		18	[占位符, Code, sep, ...]
14805	1112715153402494976	2019-04-01 21:56:48	false	false	false	true		18	[占位符, Code, sep, ...]
14805	1114863998437687296	2019-04-09 16:41:09	false	false	false	true		18	[占位符, Code, sep, ...]
14805	1111524501104885760	2019-03-29 15:05:28	false	false	false	true		18	[占位符, Code, sep, ...]
14805	1114864237131333632	2019-04-09 16:34:48	false	false	false	true		18	[占位符, Code, sep, ...]
14805	1111189494544990208	2019-03-28 16:57:45	false	false	false	true		18	[占位符, Code, sep, ...]
14805	1106476833370537984	2019-03-15 16:48:08	false	false	false	true		18	[占位符, Code, sep, ...]
14805	1114864474352779264	2019-04-07 20:19:42	false	true	false	true	32360	18	[占位符, Code, sep, ...]
14805	1109995683777085440	2019-03-25 09:53:07	false	false	false	true		18	[占位符, Code, sep, ...]
14805	1113053603926376448	2019-04-02 20:23:35	false	false	false	true		18	[占位符, Code, sep, ...]

接下来，我们需要计算每一个主题词对于用户的权重，所以需要将 topics 列中的每个主题词都拆分为单独的一条记录。可以利用 Spark 的 `explode()` 方法，达到类似“爆炸”的效果

```
import pyspark.sql.functions as F

user_article_topic = user_topic.withColumn('topic',
F.explode('topics')).drop('topics')
```

`user_article_topic` 如下图所示

```
+----------+------------------+-------------------+------+-------+---------+--------+---------+----------+--------+
|article_id|           user_id|        action_time|shared|clicked|collected|exposure|read_time|channel_id|   topic|
+----------+------------------+-------------------+------+-------+---------+--------+---------+----------+--------+
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|      补码|
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|    字符串|
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|      李白|
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|    type|
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|      元素|
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|    删除元素|
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|      负数|
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|      基数|
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|     tp2|
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|    数据类型|
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|     二进制|
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|xiaoming|
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|      大写|
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|      示例|
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|      字典|
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|    八进制|
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|      元组|
|     13401|                10|2019-03-06 10:06:12| false|  false|    false|    true|         |        18|   print|
|     13401|1114864237131333632|2019-04-09 16:39:51| false|  false|    false|    true|         |        18|      补码|
|     13401|1114864237131333632|2019-04-09 16:39:51| false|  false|    false|    true|         |        18|    字符串|
+----------+------------------+-------------------+------+-------+---------+--------+---------+----------+--------+
```

我们通过用户对哪些文章发生了行为以及该文章有哪些主题词，计算出了用户对哪些主题词发生了行为。这样，我们就可以根据用户对主题词的行为来计算主题词对用户的权重，并且将这些主题词作为用户的标签。那么，用户标签权重的计算公式为：用户标签权重 =（用户行为分值之和）x 时间衰减。其中，时间衰减公式为：时间衰减系数 = 1 / (log(t) + 1)，其中 t 为发生行为的时间距离当前时间的大小

不同的用户行为对应不同的权重，如下所示

用户行为	分值
阅读时间(<1000)	1
阅读时间(>=1000)	2
收藏	2
分享	3
点击	5

计算用户标签及权重，并存储到 Hbase 中 user_profile 表的 partial 列族中。注意，这里我们将频道 ID 和标签一起作为 partial 列族的键存储，这样我们就方便查询不同频道的标签及权重了

```
def compute_user_label_weights(partitions):
    """ 计算用户标签权重
    """
    action_weight = {
        "read_min": 1,
        "read_middle": 2,
        "collect": 2,
        "share": 3,
        "click": 5
    }

    from datetime import datetime
    import numpy as np

    # 循环处理每个用户对应的每个主题词
    for row in partitions:
        # 计算时间衰减系数
        t = datetime.now() - datetime.strptime(row.action_time,
'%Y-%m-%d %H:%M:%S')
        alpha = 1 / (np.log(t.days + 1) + 1)

        if row.read_time  == '':
            read_t = 0
        else:
            read_t = int(row.read_time)

        # 计算阅读时间的行为分数
        read_score = action_weight['read_middle'] if read_t > 1000
else action_weight['read_min']

        # 计算各种行为的权重和并乘以时间衰减系数
        weights = alpha * (row.shared * action_weight['share'] +
row.clicked * action_weight['click'] +
                           row.collected * action_weight['collect']
+ read_score)

        # 更新到user_profilehbase表
        with pool.connection() as conn:
            table = conn.table('user_profile')
            table.put('user:{}'.format(row.user_id).encode(),
                      {'partial:{}:{}'.format(row.channel_id,
row.topic).encode(): json.dumps(
                           weights).encode()})
            conn.close()

user_topic.foreachPartition(compute_user_label_weights)
```

在 Hive 中查询用户标签及权重

```
hive> select * from user_profile_hbase limit 1;
OK
user:1  {"birthday":0.0,"gender":null}
{"18:##":0.25704484358604845,"18:&#":0.25704484358604845,"18:+++":0
.23934588700996243,"18:+++++":0.23934588700996243,"18:AAA":0.274796
4402379244,"18:Animal":0.2747964402379244,"18:Author":0.27479644023
79244,"18:BASE":0.23934588700996243,"18:BBQ":0.23934588700996243,"1
8:Blueprint":1.6487786414275463,"18:Code":0.23934588700996243,"18:D
IR......
```

接下来，要将用户属性信息加入到用户画像中。读取用户基础信息，存储到用户画像表的 basic 列族即可

```
def update_user_info():
    """
    更新用户画像的属性信息
    :return:
    """
    spark.sql("use toutiao")
    user_basic = spark.sql("select user_id, gender, birthday from
user_profile")

    def udapte_user_basic(partition):

        import happybase
        #  用于读取hbase缓存结果配置
        pool = happybase.ConnectionPool(size=10,
host='172.17.0.134', port=9090)
        for row in partition:
            from datetime import date
            age = 0
            if row.birthday != 'null':
                born = datetime.strptime(row.birthday, '%Y-%m-%d')
                today = date.today()
                age = today.year - born.year - ((today.month,
today.day) < (born.month, born.day))

            with pool.connection() as conn:
                table = conn.table('user_profile')
                table.put('user:{}'.format(row.user_id).encode(),
                          {'basic:gender'.encode():
json.dumps(row.gender).encode()})
                table.put('user:{}'.format(row.user_id).encode(),
                          {'basic:birthday'.encode():
json.dumps(age).encode()})
                conn.close()

    user_basic.foreachPartition(udapte_user_basic)
```

到这里，我们的用户画像就计算完成了。

Apscheduler 定时更新

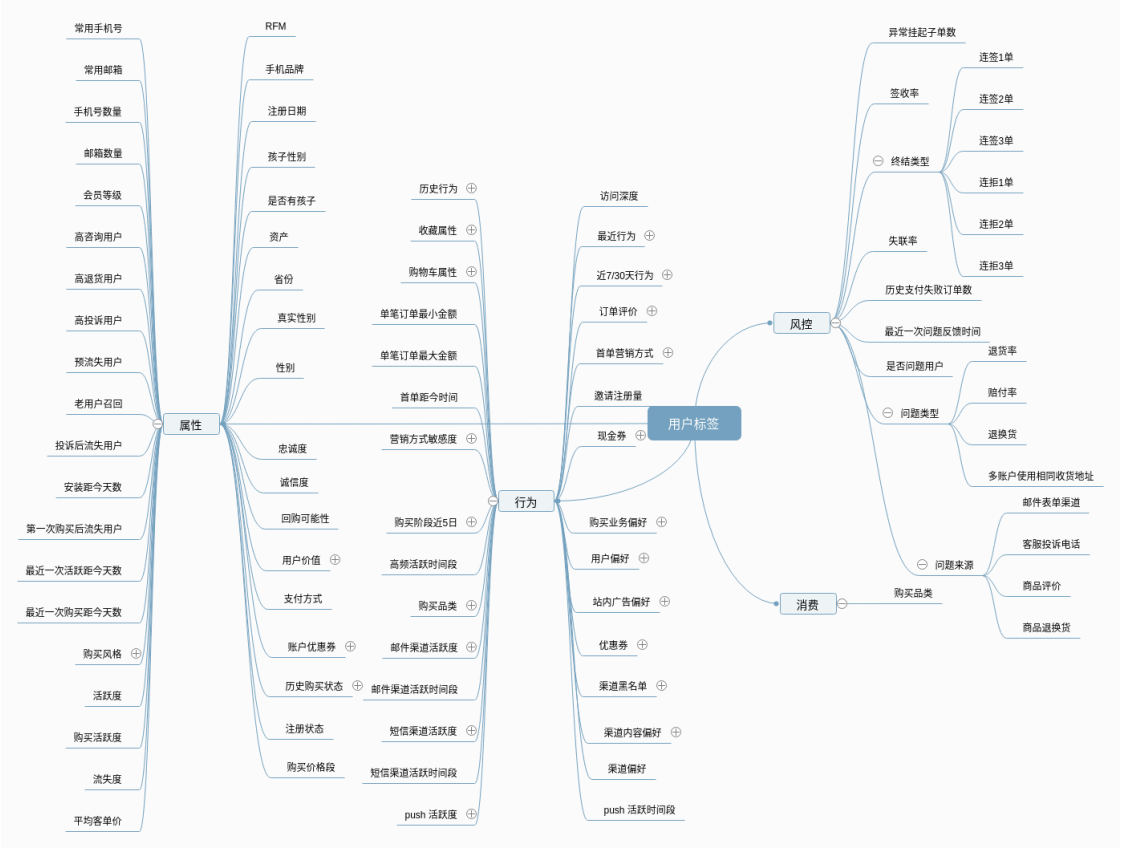
定义更新用户画像方法，首先处理用户行为日志，拆分文章主题词，接着计算用户标签的权重，最后再将用户属性信息加入到用户画像中

```
def update_user_profile():
    """
    定时更新用户画像的逻辑
    :return:
    """
    up = UpdateUserProfile()
    if up.update_user_action_basic():
        up.update_user_label()
        up.update_user_info()
```

在 Apscheduler 中添加定时更新用户画像任务，设定每隔 2 个小时更新一次

```
from apscheduler.schedulers.blocking import BlockingScheduler
from apscheduler.executors.pool import ProcessPoolExecutor

# 创建scheduler，多进程执行
executors = {
    'default': ProcessPoolExecutor(3)
}

scheduler = BlockingScheduler(executors=executors)

# 添加一个定时运行文章画像更新的任务，  每隔1个小时运行一次
scheduler.add_job(update_article_profile, trigger='interval',
hours=1)
# 添加一个定时运行用户画像更新的任务，  每隔2个小时运行一次
scheduler.add_job(update_user_profile, trigger='interval', hours=2)

scheduler.start()
```

另外说一下，在实际场景中，用户画像往往是非常复杂的，下面是电商场景的用户画像，可以了解一下。



7、构建离线文章特征和用户特征

前面我们完成了文章画像和用户画像的构建，画像数据主要是提供给召回阶段的各种召回算法使用。接下来，我们还要为排序阶段的各种排序模型做数据准备，通过特征工程将画像数据进一步加工为特征数据，以供排序模型直接使用。

我们可以将特征数据存储到 Hbase 中，这里我们先在 Hbase 中创建好 ctr_feature_article 表 和 ctr_feature_user 表，分别存储文章特征数据和用户特征数据

```
-- 文章特征表
create 'ctr_feature_article', 'article'
-- 如 article:13401 timestamp=1555635749357, value=
[18.0,0.08196639249252607,0.11217275332895373,0.1353835167902181,0.
16086650318453152,0.16356418791892943,0.16740082750337945,0.1809183
7445730974,0.1907214431716628,0.2.....................-0.0463463
4410271921,-0.06451843378804649,-0.021564142420785692,0.10212902152
136256]

-- 用户特征表
create 'ctr_feature_user', 'channel'
-- 如 4 column=channel:13, timestamp=1555647172980, value=[]
```

构建文章特征

文章特征包括文章关键词权重、文章频道以及文章向量，我们首先读取文章画像

```
spark.sql("use article")
article_profile = spark.sql("select * from article_profile")
```

在文章画像中筛选出权重最高的 K 个关键词的权重，作为文章关键词的权重向量

```
def article_profile_to_feature(row):
    try:
        article_weights = sorted(row.keywords.values())[:10]
    except Exception as e:
        article_weights = [0.0] * 10
    return row.article_id, row.channel_id, article_weights
article_profile =
article_profile.rdd.map(article_profile_to_feature).toDF(['article_
id', 'channel_id', 'weights'])
```

`article_profile` 结果如下所示，weights 即为文章关键词的权重向量

```
+----------+----------+--------------------+
|article_id|channel_id|             weights|
+----------+----------+--------------------+
|        26|        17|[0.19827163395829...|
|        29|        17|[0.26031398249056...|
|       474|        17|[0.49818598558926...|
|       964|        11|[0.42194661121527...|
|      1677|        17|[0.19827339246090...|
```

接下来，读取文章向量信息，再将频道 ID 和文章向量加入进来，利用 article_id 将 article_profile 和 article_vector 进行内连接，并将 weights 和 articlevector 转为 `vector` 类型

```
article_vector = spark.sql("select * from article_vector")
article_feature = article_profile.join(article_vector, on=
['article_id'], how='inner')

def feature_to_vector(row):

    from pyspark.ml.linalg import Vectors

    return row.article_id, row.channel_id,
Vectors.dense(row.weights), Vectors.dense(row.articlevector)

article_feature =
article_feature.rdd.map(feature_to_vector).toDF(['article_id',
'channel_id', 'weights', 'articlevector'])
```

最后，我们将 channel_id, weights, articlevector 合并为一列 features 即可（通常 channel_id 可以进行 one-hot 编码，我们这里先省略了）

```
from pyspark.ml.feature import VectorAssembler

columns = ['article_id', 'channel_id', 'weights', 'articlevector']
article_feature =
VectorAssembler().setInputCols(columns[1:4]).setOutputCol("features
").transform(article_feature)
```

`article_feature` 结果如下所示，features 就是我们准备好的文章特征

```
+----------+----------+--------------------+--------------------+--------------------+
|article_id|channel_id|             weights|       articlevector|            features|
+----------+----------+--------------------+--------------------+--------------------+
|        26|        17|[0.19827163395829...|[0.02069368539384...|[17.0,0.198271633...|
|        29|        17|[0.26031398249056...|[-0.1446092289546...|[17.0,0.260313982...|
|       474|        17|[0.49818598558926...|[0.17293323921293...|[17.0,0.498185985...|
```

最后，将文章特征结果保存到 Hbase 中

```
def save_article_feature_to_hbase(partition):
    import happybase
    pool = happybase.ConnectionPool(size=10, host='hadoop-master')
    with pool.connection() as conn:
        table = conn.table('ctr_feature_article')
        for row in partition:
            table.put('{}'.format(row.article_id).encode(),
                      {'article:{}'.format(row.article_id).encode():
str(row.features).encode()})

article_feature.foreachPartition(save_article_feature_to_hbase)
```

构建用户特征

由于用户在不同频道的偏好差异较大，所以我们要计算用户在每个频道的特征。首先读取用户画像，将空值列删除

```
spark.sql("use profile")

user_profile_hbase = spark.sql("select user_id,
information.birthday, information.gender, article_partial, env from
user_profile_hbase")
```

`user_profile_hbase` 结果如下所示，其中 article_partial 为用户标签及权重，如([‘18:vars': 0.2, '18: python':0.2, ...], ['19:java': 0.2, '19: javascript':0.2, ...], ...) 表示某个用户在 18 号频道的标签包括 var、python 等，在 19 号频道的标签包括 java、javascript等。

```
+--------------------+------+--------+--------------------+
|             user_id|gender|birthday|     article_partial|
+--------------------+------+--------+--------------------+
|              user:1|  null|     0.0|Map(18:vars -> 0....|
|             user:10|  null|     0.0|Map(18:tp2 -> 0.2...|
|             user:11|  null|     0.0|               Map()|
|user:110249052282...|  null|     0.0|               Map()|
|user:110319567345...|  null|    null|Map(18:Animal -> ...|
|user:110504528786...|  null|    null|Map(18:text -> 0....|
```

由于 gender 和 birthday 两列空值较多，我们将这两列去除（实际场景中也可以根据数据情况选择填充）

```
# 去除空值列
user_profile_hbase = user_profile_hbase.drop('env', 'birthday',
'gender')
```

提取用户 ID，获取 user_id 列的内容中 `:` 后面的数值即为用户 ID

```
def get_user_id(row):
    return int(row.user_id.split(":")[1]), row.article_partial

user_profile_hbase = user_profile_hbase.rdd.map(get_user_id)
```

将 `user_profile_hbase` 转为 DataFrame 类型

```
from pyspark.sql.types import *

_schema = StructType([
    StructField("user_id", LongType()),
    StructField("weights", MapType(StringType(), DoubleType()))
])

user_profile_hbase = spark.createDataFrame(user_profile_hbase,
schema=_schema)
```

接着，将每个频道内权重最高的 K 个标签的权重作为用户标签权重向量

```
def frature_preprocess(row):

    from pyspark.ml.linalg import Vectors

    user_weights = []
    for i in range(1, 26):
        try:
            channel_weights = sorted([row.weights[key] for key in
row.weights.keys() if key.split(':')[0] == str(i)])[:10]
            user_weights.append(channel_weights)
        except:
            user_weights.append([0.0] * 10)
    return row.user_id, user_weights

user_features =
user_profile_hbase.rdd.map(frature_preprocess).collect()
```

`user_features` 就是我们计算好的用户特征，数据结构类似 (10, [[0.2, 2.1, ...], [0.2, 2.1, ...]], ...)，其中元组第一个元素 10 即为用户 ID，第二个元素是长度为 25 的用户频道标签权重列表，列表中每个元素是长度为 K 的用户标签权重列表，代表用户在某个频道下的标签权重向量。

最后，将用户特征结果保存到 Hbase，利用 Spark 的 `batch()` 方法，按频道批量存储用户特征

```
import happybase

# 批量插入Hbase数据库中
pool = happybase.ConnectionPool(size=10, host='hadoop-master',
port=9090)
with pool.connection() as conn:
    ctr_feature = conn.table('ctr_feature_user')
    with ctr_feature.batch(transaction=True) as b:
        for i in range(len(user_features)):
            for j in range(25):
                b.put("{}".format(res[i][0]).encode(),{"channel:
{}".format(j+1).encode(): str(res[i][1][j]).encode()})
    conn.close()
```

Apscheduler 定时更新

定义文章特征和用户特征的离线更新方法

```
def update_ctr_feature():
    """
    更新文章特征和用户特征
    :return:
    """
    fp = FeaturePlatform()
    fp.update_user_ctr_feature_to_hbase()
    fp.update_article_ctr_feature_to_hbase()
```

在 Apscheduler 中添加定时更新文章特征和用户特征的任务，每隔 4 小时运行一次

```
from apscheduler.schedulers.blocking import BlockingScheduler
from apscheduler.executors.pool import ProcessPoolExecutor

# 创建scheduler，多进程执行
executors = {
    'default': ProcessPoolExecutor(3)
}

scheduler = BlockingScheduler(executors=executors)

# 添加一个定时运行文章画像更新的任务，  每隔1个小时运行一次
scheduler.add_job(update_article_profile, trigger='interval',
hours=1)
# 添加一个定时运行用户画像更新的任务，  每隔2个小时运行一次
scheduler.add_job(update_user_profile, trigger='interval', hours=2)
# 添加一个定时运行特征中心平台的任务，每隔4小时更新一次
scheduler.add_job(update_ctr_feature, trigger='interval', hours=4)

scheduler.start()
```

8、基于模型的离线召回

前面我们完成了所有的数据准备，接下来，就要开始召回阶段的工作了，可以做离线召回，也可以做在线召回，召回算法通常包括基于内容的召回和基于协同过滤的召回。ALS 模型是一种基于模型的协同过滤召回算法，本文将通过 ALS 模型实现离线召回。

首先，我们在 Hbase 中创建召回结果表 cb_recall，这里用不同列族来存储不同方式的召回结果，其中 als 表示模型召回，content 表示内容召回，online 表示在线召回。通过设置多个版本来存储多次召回结果，通过设置生存期来清除长时间未被使用的召回结果。

```
create 'cb_recall', {NAME=>'als', TTL=>7776000, VERSIONS=>999999}
alter 'cb_recall', {NAME=>'content', TTL=>7776000,
VERSIONS=>999999}
alter 'cb_recall', {NAME=>'online', TTL=>7776000, VERSIONS=>999999}

# 插入样例
put 'cb_recall', 'recall:user:5', 'als:2',[1,2,3,4,5,6,7,8,9,10]
put 'cb_recall', 'recall:user:2', 'content:1',
[45,3,5,10,289,11,65,52,109,8]
put 'cb_recall', 'recall:user:2', 'online:2',[1,2,3,4,5,6,7,8,9,10]
```

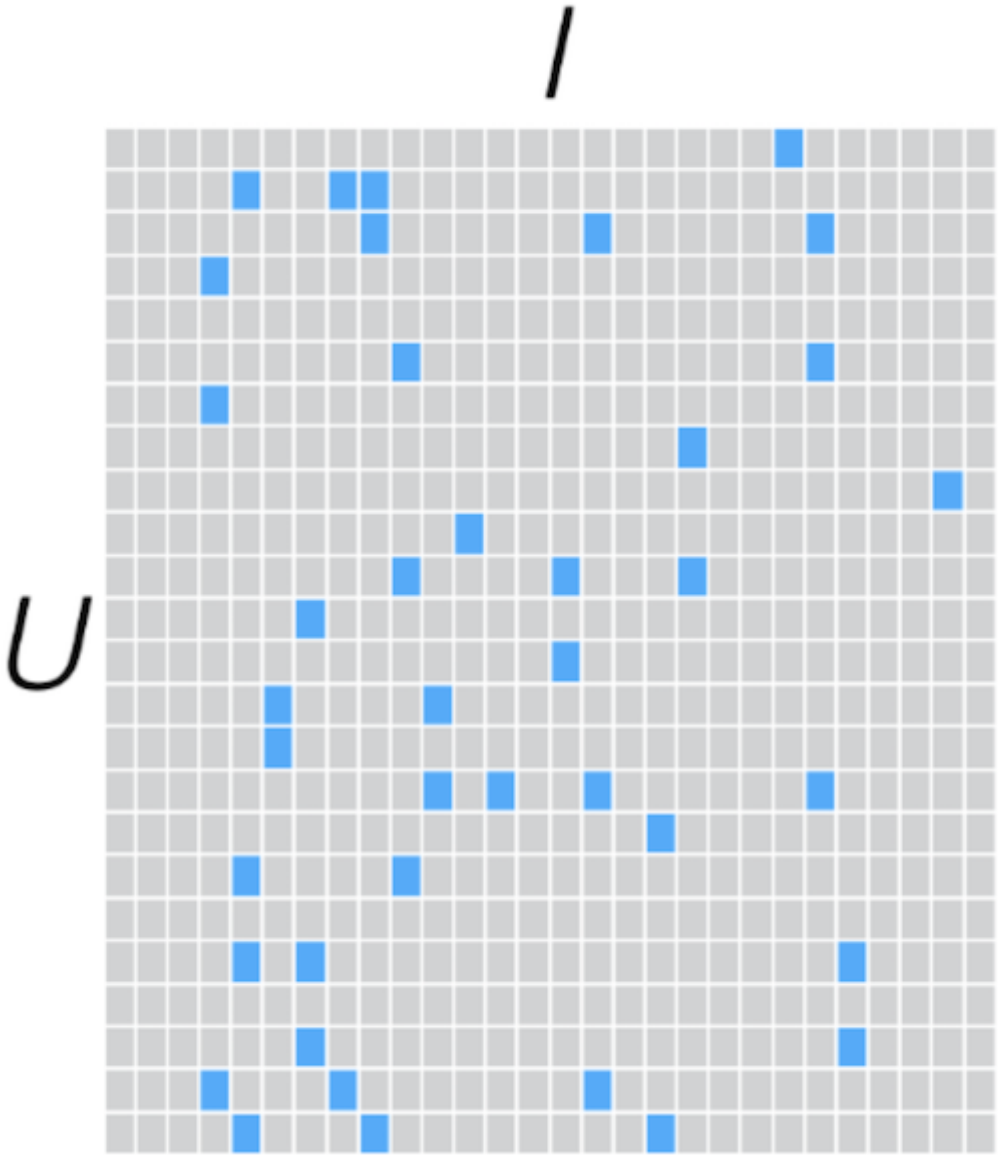
在 Hive 中建立外部表，用于离线分析

```
create external table cb_recall_hbase
(
    user_id STRING comment "userID",
    als     map<string, ARRAY<BIGINT>> comment "als recall",
    content map<string, ARRAY<BIGINT>> comment "content recall",
    online  map<string, ARRAY<BIGINT>> comment "online recall"
)
    COMMENT "user recall table"
    STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
        WITH SERDEPROPERTIES ("hbase.columns.mapping" =
":key,als:,content:,online:")
    TBLPROPERTIES ("hbase.table.name" = "cb_recall");
```

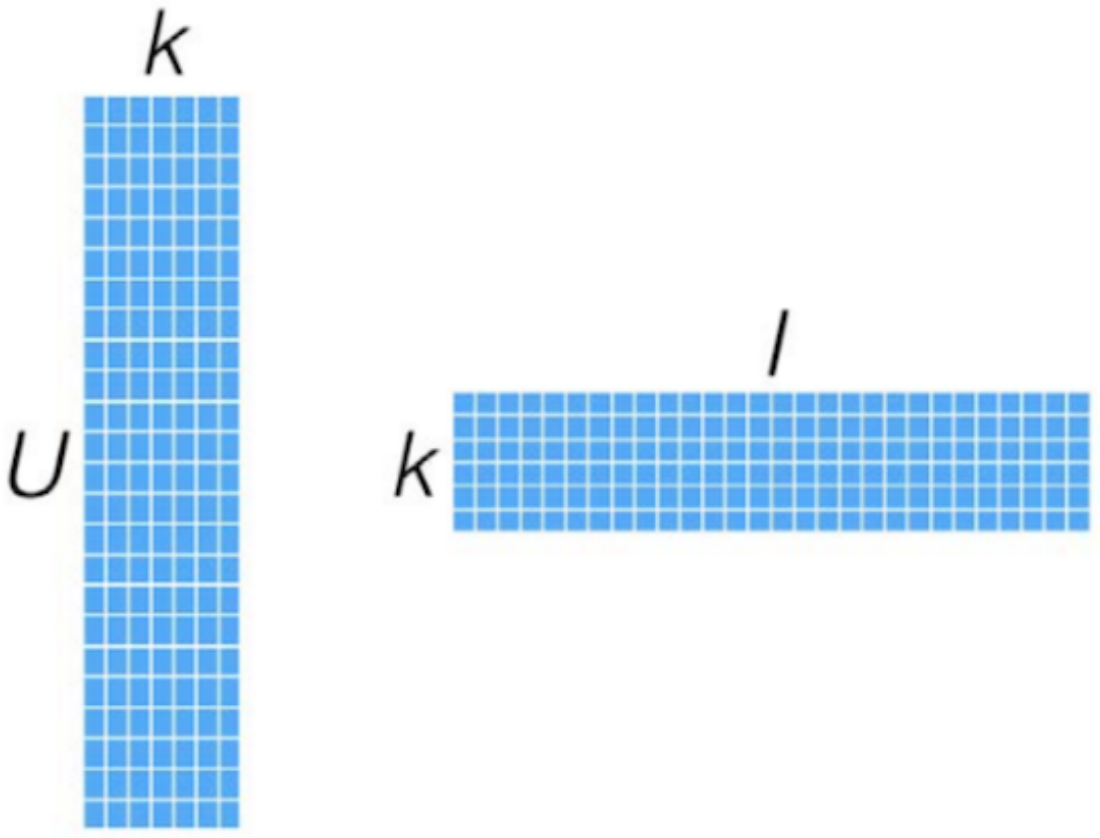
接着，在 Hbase 中创建历史召回结果表，用于过滤已历史召回结果，避免重复推荐。这里同样设置了多个版本来存储多次历史召回结果，设置了生存期来清除很长时间以前的历史召回结果

```
create 'history_recall', {NAME=>'channel', TTL=>7776000,
VERSIONS=>999999}

# 插入示例
put 'history_recall', 'recall:user:5', 'als:1',[1,2,3]
put 'history_recall', 'recall:user:5', 'als:1',[4,5,6,7]
put 'history_recall', 'recall:user:5', 'als:1',[8,9,10]
```

ALS 原理



我们先简单了解一下 ALS 模型，上图为用户和物品的关系矩阵，其中，每一行代表一个用户，每一列代表一个物品。蓝色元素代表用户查看过该物品，灰色元素代表用户未查看过该物品，假设有 m 个用户，n 个物品，为了得到用户对物品的评分，我们可以利用矩阵分解将原本较大的稀疏矩阵拆分成两个较小的稠密矩阵，即 m x k 维的用户隐含矩阵和 k x n 维的物品隐含矩阵，如下所示：



其中，用户矩阵的每一行就包括了影响用户偏好的 k 个隐含因子，物品矩阵的每一列就包括了影响物品内容的 k 个隐含因子。这里用户矩阵和物品矩阵中每个隐含因子的值就是利用交替最小二乘（Alternating Least Squares，ALS）优化算法计算而得的，所以叫做 ALS 模型。接下来，再将用户矩阵和物品矩阵相乘即可得到用户对物品的 m x n 维的评分矩阵，其中就包括每一个用户对每一个物品的评分了，进而可以根据评分进行推荐。

ALS 模型训练和预测

Spark 已经实现了 ALS 模型，我们可以直接调用。首先，我们读取用户历史点击行为，构造训练集数据，其中只需要包括用户 ID、文章 ID 以及是否点击

```
spark.sql('use profile')
user_article_basic = spark.sql("select user_id, article_id, clicked
from user_article_basic")
```

`user_article_basic` 结果如下所示，其中 clicked 表示用户对文章是否发生过点击

```
+------------------+------------------+-------+
|           user_id|        article_id|clicked|
+------------------+------------------+-------+
|1105045287866466304|             14225|  false|
|1106476833370537984|             14208|  false|
|1109980466942836736|             19233|  false|
|1109980466942836736|             44737|  false|
|1109993249109442560|             17283|  false|
|1111189494544990208|             19322|  false|
|1111524501104885760|             44161|  false|
|1112727762809913344|             18172|   true|
|1113020831425888256|1112592065390182400|  false|
|1114863735962337280|             17665|  false|
|1114863741448486912|             14208|  false|
|1114863751909081088|             13751|  false|
|1114863846486441984|             17940|  false|
|1114863941936218112|             15196|  false|
|1114863998437687296|             19233|  false|
|1114864164158832640|            141431|  false|
|1114864237131333632|             13797|  false|
|1114864354622177280|            134812|  false|
|1115089292662669312|1112608068731928576|  false|
|1115534909935452160|             18156|  false|
+------------------+------------------+-------+
```

我们需要将 clicked 由 boolean 类型转成 int 类型，即 true 为 1，false 为 0

```
def convert_boolean_int(row):
    return row.user_id, row.article_id, int(row.clicked)

user_article_basic =
user_article_basic.rdd.map(convert_boolean_int).toDF(['user_id',
'article_id', 'clicked'])
```

`user_article_basic` 结果如下所示，clicked 已经是 int 类型

```
+------------------+------------------+-------+
|           user_id|        article_id|clicked|
+------------------+------------------+-------+
|1105045287866466304|             14225|      0|
|1106476833370537984|             14208|      0|
|1109980466942836736|             19233|      0|
|1109980466942836736|             44737|      0|
|1109993249109442560|             17283|      0|
|1111189494544990208|             19322|      0|
|1111524501104885760|             44161|      0|
|1112727762809913344|             18172|      1|
|1113020831425888256|1112592065390182400|      0|
|1114863735962337280|             17665|      0|
|1114863741448486912|             14208|      0|
|1114863751909081088|             13751|      0|
|1114863846486441984|             17940|      0|
|1114863941936218112|             15196|      0|
|1114863998437687296|             19233|      0|
|1114864164158832640|            141431|      0|
|1114864237131333632|             13797|      0|
|1114864354622177280|            134812|      0|
|1115089292662669312|1112608068731928576|      0|
|1115534909935452160|             18156|      0|
+------------------+------------------+-------+
```

另外，Spark 的 ALS 模型还要求输入的用户 ID 和文章 ID 必须是从 1 开始递增的连续数字，所以需要利用 Spark 的 `Pipeline` 和 `StringIndexer`，将用户 ID 和文章 ID 建立从 1 开始递增的索引

```
from pyspark.ml.feature import StringIndexer
from pyspark.ml import Pipeline

user_indexer = StringIndexer(inputCol='user_id',
outputCol='als_user_id')
article_indexer = StringIndexer(inputCol='article_id',
outputCol='als_article_id')
pip = Pipeline(stages=[user_indexer, article_indexer])
pip_model = pip.fit(user_article_basic)
als_user_article = pip_model.transform(user_article_basic)
```

`als_user_article` 结果如下所示，als_user_id 和 als_article_id 即是 ALS 模型所需的用户索引和文章索引

user_id	article_id	clicked	als_user_id	als_article_id
1105045287866466304	14225	0	28.0	13.0
1106476833370537984	14208	0	14.0	1.0
1109980466942836736	19233	0	60.0	19.0
1109980466942836736	44737	0	60.0	17.0
1109993249109442560	17283	0	48.0	7.0
1111189494544990208	19322	0	7.0	140.0
1111524501104885760	44161	0	49.0	11.0
1112727762809913344	18172	1	45.0	55.0
1113020831425888256	1112592065390182400	0	71.0	29.0
1114863735962337280	17665	0	9.0	5.0
1114863741448486912	14208	0	62.0	1.0
1114863751909081088	13751	0	37.0	37.0
1114863846486441984	17940	0	2.0	73.0
1114863941936218112	15196	0	20.0	33.0
1114863998437687296	19233	0	24.0	19.0
1114864164158832640	141431	0	18.0	51.0
1114864237131333632	13797	0	4.0	50.0
1114864354622177280	134812	0	12.0	12.0
1115089292662669312	1112608068731928576	0	13.0	24.0
1115534909935452160	18156	0	42.0	53.0

接下来，将用户行为数据中的 als_user_id, als_article_id, clicked 三列作为训练集，对 ALS 模型进行训练，并利用 ALS 模型计算用户对文章的偏好得分，这里可以指定为每个用户保留偏好得分最高的 K 篇文章

```
from pyspark.ml.recommendation import ALS

top_k = 100
als = ALS(userCol='als_user_id', itemCol='als_article_id',
ratingCol='clicked')
als_model = als.fit(als_user_article)

recall_res = als_model.recommendForAllUsers(top_k)
```

`recall_res` 结果如下所示，其中，als_user_id 为用户索引，recommendations 为每个用户的推荐列表，包括文章索引和偏好得分，如 [[255,0.1], [10,0.08], ...]

```
+-----------+--------------------+
|als_user_id|     recommendations|
+-----------+--------------------+
|         31|[[255,0.17741075]...|
|         65|[[0,0.0], [10,0.0...|
|         53|[[0,0.0], [10,0.0...|
|         34|[[255,0.06449647]...|
|         28|[[255,0.010355139...|
|         26|[[255,0.094471075...|
|         27|[[255,0.1838476],...|
|         44|[[255,0.19231558]...|
|         12|[[255,0.42547882]...|
|         22|[[255,0.14787017]...|
|         47|[[255,0.18815793]...|
|          1|[[255,0.22982743]...|
|         52|[[255,0.24817318]...|
|         13|[[255,0.1511052],...|
|          6|[[255,0.262215], ...|
|         16|[[255,0.2500384],...|
|          3|[[207,0.91574323]...|
|         20|[[255,0.21161154]...|
|         40|[[255,0.06386929]...|
|         57|[[255,0.18380986]...|
+-----------+--------------------+
```

预测结果处理

接着，我们要将推荐结果中的用户索引和文章索引还原为用户 ID 和文章 ID，这就需要建立用户 ID 与用户索引的映射及文章 ID 与文章索引的映射，可以将前面包含用户索引和文章索引的用户行为数据 `als_user_article` 分别按照 user_id 和 article_id 分组，即可得到用户 ID 与用户索引的映射以及文章 ID 与文章索引的映射

```
user_real_index =
als_user_article.groupBy(['user_id']).max('als_user_id').withColumn
Renamed('max(als_user_id)', 'als_user_id')
article_real_index =
als_user_article.groupBy(['article_id']).max('als_article_id').with
ColumnRenamed('max(als_article_id)', 'als_article_id')
```

`user_real_index` 结果如下所示，即用户 ID 与用户索引的映射

```
+-------------------+-----------+
|            user_id|als_user_id|
+-------------------+-----------+
|1106473203766657024|       26.0|
|1113049054452908032|       44.0|
|1114863751909081088|       37.0|
|1115534909935452160|       42.0|
|1113100263847100416|       54.0|
|1103195673450250240|        5.0|
|1105045287866466304|       28.0|
|1114864237131333632|        4.0|
|1111524501104885760|       49.0|
|1109995264376045568|       19.0|
|1105105185656537088|       46.0|
|1110071654421102592|       64.0|
|1114863965080387584|       65.0|
|1114864128259784704|       17.0|
|1114864233264185344|       40.0|
|1115436666438287360|       29.0|
|1114863846486441984|        2.0|
|1115089292662669312|       13.0|
|1113316420155867136|       72.0|
|1114863902073552896|       16.0|
+-------------------+-----------+
```

再利用 `als_user_id` 将 `recall_res` 和 `user_real_index` 进行连接，加入用户 ID

```
recall_res = recall_res.join(user_real_index, on=['als_user_id'],
how='left').select(['als_user_id', 'recommendations', 'user_id'])
```

`recall_res` 结果如下所示，得到用户索引，推荐列表和用户 ID

```
+-----------+--------------------+-------------------+
|als_user_id|     recommendations|            user_id|
+-----------+--------------------+-------------------+
|          8|[[263,0.3481275],...|1109976363453906944|
|         67|[[255,0.4729983],...|1114096769035141120|
|         70|[[255,0.38408458]...|1115534898262704128|
|          0|[[255,0.58535063]...|1106396183141548032|
|         69|[[0,0.0], [10,0.0...|1114094806092480512|
|          7|[[255,0.18091725]...|1111189494544990208|
|         49|[[0,0.0], [10,0.0...|1111524501104885760|
|         29|[[255,0.10471068]...|1115436666438287360|
|         64|[[255,0.09094194]...|1110071654421102592|
|         47|[[255,0.18815793]...|1112995431274512384|
|         42|[[255,0.24995728]...|1115534909935452160|
|         44|[[255,0.19231558]...|1113049054452908032|
|         35|[[207,0.6278385],...|                  4|
|         62|[[255,0.10057049]...|1114863741448486912|
|         18|[[255,0.20005049]...|1114864164158832640|
|          1|[[255,0.22982743]...|1114864874141253632|
|         39|[[255,0.22850463]...|1115534631668547584|
|         34|[[255,0.06449647]...|1108264901190615040|
|         37|[[255,0.52489614]...|1114863751909081088|
|         25|[[255,0.37197587]...|1114865014205841408|
+-----------+--------------------+-------------------+
```

接下来，我们要构建出用户和文章的关系，利用 `explode()` 方法将 recommendations 中的每篇文章都转换为单独的一条记录，并只保留用户 ID 和文章索引这两列数据

```
import pyspark.sql.functions as F

recall_res = recall_res.withColumn('als_article_id',
F.explode('recommendations')).drop('recommendations').select(['user
_id', 'als_article_id'])
```

`recall_res` 结果如下所示，als_article_id 包括文章索引和偏好得分

```
+------------------+-----------------+
|           user_id|   als_article_id|
+------------------+-----------------+
|1109976363453906944|  [263,0.3481275]|
|1109976363453906944| [181,0.20810685]|
|1109976363453906944| [255,0.20628238]|
|1109976363453906944| [307,0.20628238]|
|1109976363453906944| [323,0.20628238]|
|1109976363453906944| [293,0.20628238]|
|1109976363453906944| [336,0.19855197]|
|1109976363453906944|   [164,0.104869]|
|1109976363453906944|[207,0.104758695]|
|1109976363453906944| [224,0.10435731]|
|1109976363453906944| [210,0.09976264]|
|1109976363453906944| [204,0.09689172]|
|1109976363453906944| [184,0.09689172]|
|1109976363453906944| [125,0.08567983]|
|1109976363453906944|[149,0.081252605]|
|1109976363453906944| [327,0.07774757]|
|1109976363453906944| [341,0.07774757]|
|1109976363453906944| [299,0.07774757]|
|1109976363453906944| [305,0.07774757]|
|1109976363453906944| [275,0.07774757]|
+------------------+-----------------+
```

我们将 als_article_id 中的偏好得分去除，只保留文章索引

```
def get_article_index(row):
    return row.user_id, row.als_article_id[0]

recall_res = recall_res.rdd.map(get_article_index).toDF(['user_id',
'als_article_id'])
```

`recall_res` 结果如下所示，得到用户 ID 和文章索引

```
+------------------+-------------+
|           user_id|als_article_id|
+------------------+-------------+
|1109976363453906944|          263|
|1109976363453906944|          181|
|1109976363453906944|          255|
|1109976363453906944|          307|
|1109976363453906944|          323|
|1109976363453906944|          293|
|1109976363453906944|          336|
|1109976363453906944|          164|
|1109976363453906944|          207|
|1109976363453906944|          224|
|1109976363453906944|          210|
|1109976363453906944|          204|
|1109976363453906944|          184|
|1109976363453906944|          125|
|1109976363453906944|          149|
|1109976363453906944|          327|
|1109976363453906944|          341|
|1109976363453906944|          299|
|1109976363453906944|          305|
|1109976363453906944|          275|
+------------------+-------------+
```

之前我们将文章 ID 和文章索引保存到了 `article_real_index`，这里利用 als_article_id 将 recall_res 和 `article_real_index` 进行连接，得到文章 ID

```
recall_res = recall_res.join(article_real_index, on=
['als_article_id'], how='left').select(['user_id', 'article_id'])
```

`recall_res` 结果如下所示，得到用户 ID 和要向其推荐的文章 ID

```
+------------------+----------+
|           user_id|article_id|
+------------------+----------+
|1108264901190615040|     13890|
|1114863751909081088|     13890|
|1114865014205841408|     13890|
|                10|     13890|
|                 5|     13890|
|1109995683777085440|     13890|
|1114864233264185344|     13890|
|1115089292662669312|     13890|
|1114864474352779264|     13890|
|1114865875103514624|     13890|
|1114863941936218112|     13890|
|1113004557979353088|     13890|
|1114863748553637888|     13890|
|1103195673450250240|     13890|
|1114865402044743680|     13890|
|1114863998437687296|     13890|
|                33|     13890|
|1114863735962337280|     13890|
|1106473203766657024|     13890|
|1114863902073552896|     13890|
+------------------+----------+
```

推荐结果存储

为了方便查询，我们需要将推荐结果按频道分别进行存储。首先，读取文章完整信息，得到频道 ID

```
spark.sql('use article')
article_data = spark.sql("select article_id, channel_id from
article_data")
```

利用 article_id 将 `recall_res` 和 `article_data` 进行连接，在推荐结果中加入频道 ID

```
recall_res = recall_res.join(article_data, on=['article_id'],
how='left')
```

`recall_res` 结果如下所示，推荐结果加入了频道 ID

```
+----------+-------------------+----------+
|article_id|            user_id|channel_id|
+----------+-------------------+----------+
|     13401|1114094806092480512|        18|
|     13401|1111524501104885760|        18|
|     13401|1114866560301793280|        18|
|     13401|1113316420155867136|        18|
|     13401|1109984273839947776|        18|
|     13401|1114865682668847104|        18|
|     13401|1114863965080387584|        18|
|     14805|1105045287866466304|        18|
|     14805|1114863846486441984|        18|
|     14805|1115535317173010432|        18|
|     14805|1114864128259784704|        18|
|     14805|1114871412419461120|        18|
|     14805|1114863759672737792|        18|
|     14805|                 10|        18|
|     14805|                  5|        18|
|     14805|1109995683777085440|        18|
|     14805|1114864233264185344|        18|
|     14805|1115089292662669312|        18|
|     14805|1114864474352779264|        18|
|     14805|1114865875103514624|        18|
+----------+-------------------+----------+
```

将推荐结果按照 user_id 和 channel_id 进行分组，利用 `collect_list()` 方法将文章 ID 合并为文章列表

```
recall_res = recall_res.groupBy(['user_id',
'channel_id']).agg(F.collect_list('article_id')).withColumnRenamed(
'collect_list(article_id)', 'article_list')
```

`recall_res` 结果如下所示，article_list 为某用户在某频道下的推荐文章列表

user_id	channel_id	article_list
23	18	[14805, 14839, 17...
1109993249109442560	7	[141437]
1113049054452908032	7	[141437]
1113100263847100416	5	[141440]
1114863751909081088	7	[141437]
38	13	[141431, 141431]
1114864233264185344	13	[141431, 141431]
10	5	[141440]
1106473203766657024	7	[141437]
33	18	[14805, 14839, 17...
1106473203766657024	null	[1112593324574769...
1	7	[141437]
1105093883106164736	13	[141431, 141431]
1114866560301793280	7	[141469]
1114864434305564672	7	[141437]
1105093883106164736	5	[141440]
1114863759672737792	5	[141440]
1113100263847100416	7	[141437]
1114863991156375552	13	[141431, 141431]
1113049054452908032	null	[1112593242529988...

最后，将推荐结果按频道分别存入召回结果表 cb_recall 及历史召回结果表 history_recall。注意，在保存新的召回结果之前需要根据历史召回结果进行过滤，防止重复推荐

```
recall_res = recall_res.dropna()
recall_res.foreachPartition(save_offline_recall_hbase)

def save_offline_recall_hbase(partition):
    """ALS模型离线召回结果存储
    """
    import happybase
    pool = happybase.ConnectionPool(size=10, host='hadoop-master',
port=9090)
    for row in partition:
        with pool.connection() as conn:
            # 读取历史召回结果表
            history_table = conn.table('history_recall')
            # 读取包含多个版本的历史召回结果
            history_article_data = history_table.cells('reco:his:
{}'.format(row.user_id).encode(),
                                                       'channel:
{}'.format(row.channel_id).encode())

            # 合并多个版本历史召回结果
            history_article = [] (比如有的用户会比较怀旧)
            if len(history_article_data) >= 2:
                for article in history_article_data[:-1]:
                    history_article.extend(eval(article))
            else:
                history_article = []

            # 过滤history_article
            recall_article = list(set(row.article_list) -
set(history_article))

            if recall_article:
                table = conn.table('cb_recall')
                table.put('recall:user:
{}'.format(row.user_id).encode(), {'als:
{}'.format(row.channel_id).encode(): str(recall_article).encode()})
                history_table.put("reco:his:
{}".format(row.user_id).encode(), {'channel:
{}'.format(row.channel_id): str(recall_article).encode()})
            conn.close()
```

可以根据用户 ID 和频道 ID 来查询召回结果

```
hbase(main):028:0> get 'cb_recall', 'recall:user:2'
COLUMN                        CELL
als:13                       timestamp=1558041569201, value=
[141431,14381, 17966, 17454, 14125, 16174]
```

Apscheduler 定时更新

在用户召回方法 `update_user_recall()` 中，增加基于模型的离线召回方法 `update_content_recall()`，首先读取用户行为日志，进行数据预处理，构建训练集，接着对 ALS 模型进行训练和预测，最后对预测出的推荐结果进行解析并按频道分别存入召回结果表和历史召回结果表

```
def update_user_recall():
    """
    用户的频道推荐召回结果更新逻辑
    :return:
    """
    ur = UpdateRecall(500)
    ur.update_als_recall()
```

添加定时更新用户召回结果的任务，每隔 3 小时运行一次

```
from apscheduler.schedulers.blocking import BlockingScheduler
from apscheduler.executors.pool import ProcessPoolExecutor

# 创建scheduler，多进程执行
executors = {
    'default': ProcessPoolExecutor(3)
}

scheduler = BlockingScheduler(executors=executors)

# 添加一个定时运行文章画像更新的任务，  每隔1个小时运行一次
scheduler.add_job(update_article_profile, trigger='interval',
hours=1)
# 添加一个定时运行用户画像更新的任务，  每隔2个小时运行一次
scheduler.add_job(update_user_profile, trigger='interval', hours=2)
# 添加一个定时运行用户召回更新的任务，每隔3小时运行一次
scheduler.add_job(update_user_recall, trigger='interval', hours=3)
# 添加一个定时运行特征中心平台的任务，每隔4小时更新一次
scheduler.add_job(update_ctr_feature, trigger='interval', hours=4)

scheduler.start()
```

9、基于内容的离线及在线召回

在上篇文章中，我们实现了基于模型的离线召回，属于基于协同过滤的召回算法。接下来，本文就讲一下另一个经典的召回方式，那就是如何实现基于内容的离线召回。相比于协同过滤来说，基于内容的召回会简单很多，主要思路就是召回用户点击过的文章的相似文章，通常也被叫做 u2i2i。

离线召回

首先，读取用户历史行为数据，得到用户历史点击过的文章

```
spark.sql('use profile')
user_article_basic = spark.sql("select * from user_article_basic")
user_article_basic = user_article_basic.filter('clicked=True')
```

`user_article_basic` 结果如下所示

```
+------------------+------------------+------------------+---------+------+-------+---------+--------+---------+
|           user_id|       action_time|        article_id|channel_id|shared|clicked|collected|exposure|read_time|
+------------------+------------------+------------------+---------+------+-------+---------+--------+---------+
|1112727762809913344|2019-04-03 12:51:57|             18172|       18| false|   true|     true|    true|    19413|
|                 1|2019-03-07 16:57:34|             44386|       18| false|   true|    false|    true|    17850|
|1109976363453906944|2019-03-25 11:52:31|             13728|       18| false|   true|    false|    true|    14218|
|1114864354622177280|2019-04-09 16:39:22|             17304|       18| false|   true|    false|    true|         |
|                23|2019-04-03 08:10:23|             44739|       18| false|   true|    false|    true|     7013|
|                 1|2019-03-17 10:32:01|             17632|       18| false|   true|    false|    true|     1014|
|1114863748553637888|2019-04-09 16:41:08|            141437|        7| false|   true|    false|    true|     2066|
|1109945594201763840|2019-04-06 23:56:56|             15140|       18| false|   true|    false|    true|     1433|
|1112715153402494976|2019-04-01 22:01:14|             17542|       18| false|   true|    false|    true|    20092|
|1114863751909081088|2019-04-09 16:40:43|             15139|       18| false|   true|    false|    true|         |
|                 2|2019-03-05 10:19:54|             44371|       18| false|   true|    false|    true|      938|
|                23|2019-04-02 15:06:37|1112593242529988608|        3| false|   true|    false|    true|     3366|
|                 4|2019-04-04 14:28:19|1112525856586072064|        7| false|   true|     true|    true|    54151|
|1114863751909081088|2019-04-07 20:13:33|1112608068731928576|        3| false|   true|    false|    true|     1711|
|1114863941936218112|2019-04-07 20:24:40|             18795|       18| false|   true|    false|    true|    57949|
|1114864354622177280|2019-04-09 16:41:05|             18156|       18| false|   true|    false|    true|     6901|
|1114865014205841408|2019-04-09 16:42:47|            141437|        7| false|   true|    false|    true|     5091|
|                 2|2019-03-07 10:06:20|             18103|       18| false|   true|    false|    true|      648|
|                 1|2019-03-22 00:52:31|             18335|       18| false|   true|    false|    true|    19983|
|                 1|2019-03-23 11:31:24|1108924834420621312|       18| false|   true|    false|    true|     3131|
+------------------+------------------+------------------+---------+------+-------+---------+--------+---------+
```

接下来，遍历用户历史点击过的文章，获取与之相似度最高的 K 篇文章即可。可以根据之前计算好的文章相似度表 article_similar 进行相似文章查询，接着根据历史召回结果进行过滤，防止重复推荐。最后将召回结果按照频道分别存入召回结果表及历史召回结果表

```
user_article_basic.foreachPartition(get_clicked_similar_article)

def get_clicked_similar_article(partition):
    """召回用户点击文章的相似文章
    """
    import happybase
    pool = happybase.ConnectionPool(size=10, host='hadoop-master')

    with pool.connection() as conn:
        similar_table = conn.table('article_similar')
        for row in partition:
            # 读取文章相似度表,根据文章ID获取相似文章
            similar_article =
similar_table.row(str(row.article_id).encode(),
                                                  columns=
[b'similar'])
            # 按照相似度进行排序
            similar_article_sorted =
sorted(similar_article.items(), key=lambda item: item[1],
reverse=True)
            if similar_article_sorted:
                # 每次行为推荐10篇文章
                similar_article_topk = [int(i[0].split(b':')[1])
for i in similar_article_sorted][:10]

                # 根据历史召回结果进行过滤
```

```
                history_table = conn.table('history_recall')
                history_article_data =
history_table.cells('reco:his:{}'.format(row.user_id).encode(),
'channel:{}'.format(row.channel_id).encode())
                # 将多个版本都加入历史文章ID列表
                history_article = []
                if len(history_article_data) >= 2:
                    for article in history_article_data[:-1]:
                        history_article.extend(eval(article))
                else:
                    history_article = []

                # 过滤history_article
                recall_article = list(set(similar_article_topk) -
set(history_article))

                # 存储到召回结果表及历史召回结果表
                if recall_article:
                    content_table = conn.table('cb_recall')
                    content_table.put("recall:user:
{}".format(row.user_id).encode(), {'content:
{}'.format(row.channel_id).encode(): str(recall_article).encode()})

                    # 放入历史召回结果表
                    history_table.put("reco:his:
{}".format(row.user_id).encode(), {'channel:
{}'.format(row.channel_id).encode(): str(recall_article).encode()})
```

可以根据用户 ID 和频道 ID 来查询召回结果

```
hbase(main):028:0> get 'cb_recall', 'recall:user:2'
COLUMN                      CELL
content:13                     timestamp=1558041569201, value=
[141431,14381, 17966, 17454, 14125, 16174]
```

最后，使用 Apscheduler 定时更新。在用户召回方法 `update_user_recall()` 中，增加基于内容的离线召回方法 `update_content_recall()`，首先读取用户行为日志，并筛选用户点击的文章，接着读取文章相似表，获取相似度最高的 K 篇文章，然后根据历史召回结果进行过滤，防止重复推荐，最后，按频道分别存入召回结果表及历史召回结果表

```
def update_user_recall():
    """
    用户的频道推荐召回结果更新逻辑
    :return:
    """
    ur = UpdateRecall(500)
    ur.update_als_recall()
    ur.update_content_recall()
```

之前已经添加好了定时更新用户召回结果的任务，每隔 3 小时运行一次，这样就完成了基于内容的离线召回。

```
from apscheduler.schedulers.blocking import BlockingScheduler
from apscheduler.executors.pool import ProcessPoolExecutor

# 创建scheduler，多进程执行
executors = {
    'default': ProcessPoolExecutor(3)
}

scheduler = BlockingScheduler(executors=executors)

# 添加一个定时运行文章画像更新的任务，  每隔1个小时运行一次
scheduler.add_job(update_article_profile, trigger='interval',
hours=1)
# 添加一个定时运行用户画像更新的任务，  每隔2个小时运行一次
scheduler.add_job(update_user_profile, trigger='interval', hours=2)
# 添加一个定时运行用户召回更新的任务，每隔3小时运行一次
scheduler.add_job(update_user_recall, trigger='interval', hours=3)
# 添加一个定时运行特征中心平台的任务，每隔4小时更新一次
scheduler.add_job(update_ctr_feature, trigger='interval', hours=4)

scheduler.start()
```

在线召回

前面我们实现了基于内容的离线召回，接下来我们将实现基于内容的在线召回。在线召回的实时性更好，能够根据用户的线上行为实时反馈，快速跟踪用户的偏好，也能够解决用户冷启动问题。离线召回和在线召回唯一的不同就是，离线召回读取的是用户历史行为数据，而在线召回读取的是用户实时的行为数据，从而召回用户当前正在阅读的文章的相似文章。

首先，我们通过 Spark Streaming 读取 Kafka 中的用户实时行为数据，Spark Streaming 配置如下

```
from pyspark import SparkConf
from pyspark.sql import SparkSession
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils
from setting.default import DefaultConfig
import happybase

SPARK_ONLINE_CONFIG = (
        ("spark.app.name", "onlineUpdate"),
        ("spark.master", "yarn"),
        ("spark.executor.instances", 4)
    )

KAFKA_SERVER = "192.168.19.137:9092"

# 用于读取hbase缓存结果配置
pool = happybase.ConnectionPool(size=10, host='hadoop-master',
port=9090)
conf = SparkConf()
conf.setAll(SPARK_ONLINE_CONFIG)
sc = SparkContext(conf=conf)
stream_c = StreamingContext(sc, 60)

# 基于内容召回配置,用于收集用户行为
similar_kafkaParams = {"metadata.broker.list":
DefaultConfig.KAFKA_SERVER, "group.id": 'similar'}
SIMILAR_DS = KafkaUtils.createDirectStream(stream_c, ['click-
trace'], similar_kafkaParams)
```

Kafka 中的用户行为数据，如下所示

```
{"actionTime":"2019-12-10
21:04:39","readTime":"","channelId":18,"param":{"action": "click",
"userId": "2", "articleId": "116644", "algorithmCombine": "C2"}}
```

接下来，利用 Spark Streaming 将用户行为数据传入到 `get_similar_online_recall()` 方法中，这里利用 `json.loads()` 方法先将其转换为了 json 格式，注意用户行为数据在每条 Kafka 消息的第二个位置

```
SIMILAR_DS.map(lambda x:
json.loads(x[1])).foreachRDD(get_similar_online_recall)
```

接着，遍历用户行为数据，这里可能每次读取到多条用户行为数据。筛选出被点击、收藏或分享过的文章，并获取与其相似度最高的 K 篇文章，再根据历史召回结果表进行过滤，防止重复推荐，最后，按频道分别存入召回结果表及历史召回结果表

```
def get_online_similar_recall(rdd):
    """
    获取在线相似文章
    :param rdd:
    :return:
    """
    import happybase

    topk = 10
    # 初始化happybase连接
    pool = happybase.ConnectionPool(size=10, host='hadoop-master',
port=9090)
    for data in rdd.collect():

        # 根据用户行为筛选文章
        if data['param']['action'] in ["click", "collect",
"share"]:
            with pool.connection() as conn:
                similar_table = conn.table("article_similar")

                # 根据用户行为数据涉及文章找出与之最相似文章(基于内容的相似)
                similar_article =
similar_table.row(str(data["param"]["articleId"]).encode(),
columns=[b"similar"])
                similar_article = sorted(similar_article.items(),
key=lambda x: x[1], reverse=True)  # 按相似度排序

                if similar_article:
                    similar_article_topk = [int(i[0].split(b":")
[1]) for i in similar_article[:topk]] # 选取K篇作为召回推荐结果

                    # 根据历史召回结果进行过滤
                    history_table = conn.table('history_recall')
                    history_article_data =
history_table.cells(b"reco:his:%s" % data["param"]
["userId"].encode(), b"channel:%d" % data["channelId"])
                    # 将多个版本都加入历史文章ID列表
```

```
                    # 将多个版本都加入历史文章ID列表
                    history_article = []
                    if len(history_article_data) >1:
                        for article in history_article_data[:-1]:
                            history_article.extend(eval(article))
                    else:
                        history_article = []

                    # 过滤history_article
                    recall_article = list(set(similar_article_topk)
- set(history_article))

                    # 如果有召回结果,按频道分别存入召回结果表及历史召回结果表
                    if recall_article:
                        recall_table = conn.table("cb_recall")
                        recall_table.put(b"recall:user:%s" %
data["param"]["userId"].encode(), {b"online:%d" %
data["channelId"]: str(recall_article).encode()})
                        history_table.put(b"reco:his:%s" %
data["param"]["userId"].encode(), {b"channel:%d" %
data["channelId"]: str(recall_article).encode()})

                conn.close()
```

可以根据用户 ID 和频道 ID 来查询召回结果

```
hbase(main):028:0> get 'cb_recall', 'recall:user:2'
COLUMN                    CELL
online:13                   timestamp=1558041569201, value=
[141431,14381, 17966, 17454, 14125, 16174]
```

创建 `online_update.py`，加入基于内容的在线召回逻辑

```
if __name__ == '__main__':
    ore = OnlineRecall()
    ore.update_content_recall()
    stream_sc.start()
    _ONE_DAY_IN_SECONDS = 60 * 60 * 24
    try:
        while True:
            time.sleep(_ONE_DAY_IN_SECONDS)
    except KeyboardInterrupt:
        pass
```

利用 Supervisor 进行进程管理，并开启实时运行，配置如下，其中 environment 需要指定运行所需环境

```
[program:online]
environment=JAVA_HOME=/root/bigdata/jdk,SPARK_HOME=/root/bigdata/sp
ark,HADOOP_HOME=/root/bigdata/hadoop,PYSPARK_PYTHON=/miniconda2/env
s/reco_sys/bin/python
,PYSPARK_DRIVER_PYTHON=/miniconda2/envs/reco_sys/bin/python,PYSPARK
_SUBMIT_ARGS='--packages org.apache.spark:spark-streaming-kafka-0-
8_2.11:2.2.2 pyspark-shell'
command=/miniconda2/envs/reco_sys/bin/python
/root/toutiao_project/reco_sys/online/online_update.py
directory=/root/toutiao_project/reco_sys/online
user=root
autorestart=true
redirect_stderr=true
stdout_logfile=/root/logs/onlinesuper.log
loglevel=info
stopsignal=KILL
stopasgroup=true
killasgroup=true
```

扫一扫关注微信公众号！专注于搜索和推荐系统，尝试使用算法去更好的服务于用户，包括但不局限于机器学习，深度学习，强化学习，自然语言理解，知识图谱，还不定时分享技术，资料，思考等文章！

10、基于热门文章和新文章的在线召回

在上篇文章中我们实现了基于内容的在线召回，接下来，我们将实现基于热门文章和新文章的在线召回。主要思路是根据点击次数，统计每个频道下的热门文章，根据发布时间统计每个频道下的新文章，当推荐文章不足时，可以根据这些文章进行补足。

由于数据量较小，这里采用 Redis 存储热门文章和新文章的召回结果，数据结构如下所示

热门文章召回	结构	示例
popular_recall	ch:{}:hot	ch:18:hot

新文章召回	结构	示例
new_article	ch:{}:new	ch:18:new

热门文章存储，键为 `ch:频道ID:hot` 值为 `分数` 和 `文章ID`

```
# ZINCRBY key increment member
# ZSCORE
# 为有序集 key 的成员 member 的 score 值加上增量 increment 。
client.zincrby("ch:{}:hot".format(row['channelId']), 1,
row['param']['articleId'])

# ZREVRANGE key start stop [WITHSCORES]
client.zrevrange(ch:{}:new, 0, -1)
```

新文章存储，键为 `ch:{频道ID}:new` 值为 `文章ID:时间戳`

```
# ZADD ZRANGE
# ZADD key score member [[score member] [score member] ...]
# ZRANGE page_rank 0 -1
client.zadd("ch:{}:new".format(channel_id), {article_id:
time.time()})
```

热门文章在线召回

首先，添加 Spark Streaming 和 Kafka 的配置，热门文章读取由业务系统发送到 Kafka 的 click-trace 主题中的用户实时行为数据

```
KAFKA_SERVER = "192.168.19.137:9092"
click_kafkaParams = {"metadata.broker.list": KAFKA_SERVER}
HOT_DS = KafkaUtils.createDirectStream(stream_c, ['click-trace'],
click_kafkaParams)
```

接下来，利用 Spark Streaming 读取 Kafka 中的用户行为数据，筛选出被点击过的文章，将 Redis 中的文章热度分数进行累加即可

```
client = redis.StrictRedis(host=DefaultConfig.REDIS_HOST,
port=DefaultConfig.REDIS_PORT, db=10)

def update_hot_redis(self):
    """
    收集用户行为，更新热门文章分数
    :return:
    """
    def update_hot_article(rdd):
        for data in rdd.collect():
            # 过滤用户行为
            if data['param']['action'] in ['exposure', 'read']:
                pass
            else:
                client.zincrby("ch:
{}:hot".format(data['channelId']), 1, data['param']['articleId'])

    HOT_DS.map(lambda x:
json.loads(x[1])).foreachRDD(update_hot_article)
```

测试，写入用户行为日志

```
echo {\"actionTime\":\"2019-04-10
21:04:39\",\"readTime\":\"\",\"channelId\":18,\"param\":
{\"action\": \"click\", \"userId\": \"2\", \"articleId\":
\"14299\", \"algorithmCombine\": \"C2\"}} >> userClick.log
```

查询热门文章

```
127.0.0.1:6379[10]> keys *
1) "ch:18:hot"
127.0.0.1:6379[10]> ZRANGE "ch:18:hot" 0 -1
1) "14299"
```

新文章在线召回

首先，添加 Spark Streaming 和 Kafka 的配置，新文章读取由业务系统发送到 Kafka 的 new-article 主题中的最新发布文章数据

```
NEW_ARTICLE_DS = KafkaUtils.createDirectStream(stream_c, ['new-
article'], click_kafkaParams)
```

接下来，利用 Spark Streaming 读取 Kafka 的新文章，将其按频道添加到 Redis 中，Redis 的值为当前时间

```
def  update_new_redis(self):
    """更新频道最新文章
    :return:
    """
    def add_new_article(rdd):
        for row in rdd.collect():
            channel_id, article_id = row.split(',')
            client.zadd("ch:{}:new".format(channel_id),
{article_id: time.time()})

    NEW_ARTICLE_DS.map(lambda x: x[1]).foreachRDD(add_new_article)
```

还需要在 Kafka 的启动脚本中添加 new-article 主题监听配置，这样就可以收到业务系统发送过来的新文章了，重新启动 Flume 和 Kafka

```
/root/bigdata/kafka/bin/kafka-topics.sh --zookeeper
192.168.19.137:2181 --create --replication-factor 1 --topic new-
article --partitions 1
```

测试，向 Kafka 发送新文章数据

```
from kafka import KafkaProducer

# kafka消息生产者
kafka_producer = KafkaProducer(bootstrap_servers=
['192.168.19.137:9092'])

# 构造消息并发送
msg = '{},{}'.format(18, 13891)
kafka_producer.send('new-article', msg.encode())
```

查看新文章

```
127.0.0.1:6379[10]> keys *
1) "ch:18:hot"
2) "ch:18:new"
127.0.0.1:6379[10]> ZRANGE "ch:18:new" 0 -1
1) "13890"
2) "13891"
```

最后，修改 `online_update.py`，加入基于热门文章和新文章的在线召回逻辑，开启实时运行即可

```
if __name__ == '__main__':
    ore = OnlineRecall()
    ore.update_content_recall()
    ore.update_hot_redis()
    ore.update_new_redis()
    stream_sc.start()
    # 使用 ctrl+c 可以退出服务
    _ONE_DAY_IN_SECONDS = 60 * 60 * 24
    try:
        while True:
            time.sleep(_ONE_DAY_IN_SECONDS)
    except KeyboardInterrupt:
        pass
```

到这里，我们就完成了召回阶段的全部工作，包括基于模型和基于内容的离线召回，以及基于内容、热门文章和新文章的在线召回。通过召回，我们可以从数百万甚至上亿的原始物品数据中，筛选出和用户相关的几百、几千个可能感兴趣的物品，后面，我们将要进入到排序阶段，对召回的几百、几千个物品进行进一步的筛选和排序。

11、基于 LR 模型的离线排序

前面，我们已经完成了召回阶段的全部工作，通过召回，我们可以从数百万甚至上亿的原始物品数据中，筛选出和用户相关的几百、几千个可能感兴趣的物品。接下来，我们将要进入到排序阶段，对召回的几百、几千个物品进行进一步的筛选和排序。

排序流程包括离线排序和在线排序：

- 离线排序 读取前天（第 T - 2 天）之前的用户行为数据作为训练集，对离线模型进行训练；训练完成后，读取昨天（第 T - 1 天）的用户行为数据作为验证集进行预测，根据预测结果对离线模型进行评估；若评估通过，当天（第 T 天）即可将离线模型更新到定时任务中，定时执行预测任务；明天（第 T + 1 天）就能根据今天的用户行为数据来观察更新后离线模型的预测效果。（注意：数据生产有一天时间差，第 T 天生成第 T - 1 天的数据）

- 在线排序 读取前天（第 T - 2 天）之前的用户行为数据作为训练集，对在线模型进行训练；训练完成后，读取昨天（第 T - 1 天）的用户行为数据作为验证集进行预测，根据预测结果对在线模型进行评估；若评估通过，当天（第 T 天）即可将在线模型更新到线上，实时执行排序任务；明天（第 T + 1 天）就能根据今天的用户行为数据来观察更新后在线模型的预测效果。

这里再补充一个数据集划分的小技巧：可以横向划分，随机或按用户或其他样本选择策略；也可以纵向划分，按照时间跨度，比如一周的数据中，周一到周四是训练集，周五周六是测试集，周日是验证集。

利用排序模型可以进行评分预测和用户行为预测，通常推荐系统利用排序模型进行用户行为预测，比如点击率（CTR）预估，进而根据点击率对物品进行排序，目前工业界常用的点击率预估模型有如下 3 种类型：

- 宽模型 + 特征工程 LR / MLR + 非 ID 类特征（人工离散 / GBDT / FM），可以使用 Spark 进行训练
- 宽模型 + 深模型 Wide&Deep，DeepFM，可以使用 TensorFlow 进行训练
- 深模型： DNN + 特征 Embedding，可以使用 TensorFlow 进行训练

这里的宽模型即指线性模型，线性模型的优点包括：

- 相对简单，训练和预测的计算复杂度都相对较低
- 可以集中精力发掘新的有效特征，且可以并行化工作
- 解释性较好，可以根据特征权重做解释

本文我们将采用逻辑回归作为离线模型，进行点击率预估。逻辑回归（Logistic Regression，LR）是基础的二分类模型，也是监督学习的一种，通过对有标签的训练集数据进行特征学习，进而可以对测试集（新数据）的标签进行预测。我们这里的标签就是指用户是否对文章发生了点击行为。

构造训练集

读取用户历史行为数据，将 clicked 作为训练集标签

```
spark.sql("use profile")
user_article_basic = spark.sql("select * from
user_article_basic").select(['user_id', 'article_id', 'clicked'])
```

`user_article_basic` 结果如下所示

user_id	article_id	clicked
1105045287866466304	14225	false
1106476833370537984	14208	false
1109980466942836736	19233	false
1109980466942836736	44737	false
1109993249109442560	17283	false
1111189494544990208	19322	false
1111524501104885760	44161	false
1112727762809913344	18172	true
1113020831425888256	1112592065390182400	false
1114863735962337280	17665	false
1114863741448486912	14208	false
1114863751909081088	13751	false
1114863846486441984	17940	false
1114863941936218112	15196	false
1114863998437687296	19233	false
1114864164158832640	141431	false
1114864237131333632	13797	false
1114864354622177280	134812	false
1115089292662669312	1112608068731928576	false
1115534909935452160	18156	false

之前我们已经计算好了文章特征和用户特征，并存储到了 Hbase 中。这里我们遍历用户历史行为数据，根据其中文章 ID 和用户 ID 分别获取文章特征和用户特征，再将标签转为 int 类型，这样就将一条用户行为数据构造成为了一个样本，再将所有样本加入到训练集中

```
train = []
for user_id, article_id, clicked in user_article_basic:
    try:
        article_feature =
eval(hbu.get_table_row('ctr_feature_article',
'{}'.format(article_id).encode(), 'article:
{}'.format(article_id).encode()))
    except Exception as e:
        article_feature = []
    try:
        user_feature = eval(hbu.get_table_row('ctr_feature_user',
'{}'.format(temp.user_id).encode(), 'channel:
{}'.format(temp.channel_id).encode()))
    except Exception as e:
        user_feature = []

    if not article_feature:
        article_feature = [0.0] * 111
    if not user_feature:
        user_feature = [0.0] * 10

    sample = []
    sample.append(user_feature)
    sample.append(article_feature)
    sample.append(int(clicked))

    train.append(sample)
```

接下来，还需要利用 Spark 的 Vectors 将 `array<double>` 类型的 article_feature 和 user_feature 转为 `vector` 类型

```
columns = ['article_feature', 'user_feature', 'clicked']

def list_to_vector(row):
    from pyspark.ml.linalg import Vectors

    return Vectors.dense(row[0]), Vectors.dense(row[1]), row[2]

train = train.rdd.map(list_to_vector).toDF(columns)
```

再将 article_feature, user_feature 合并为统一输入到 LR 模型的特征列 features，这样就完成训练集的构建

```
train =
VectorAssembler().setInputCols(columns[0:1]).setOutputCol('features
').transform(train)
```

模型训练

Spark 已经实现好了 LR 模型，通过指定训练集 train 的特征列 features 和标签列 clicked，即可对 LR 模型进行训练，再将训练好的模型保存到 HDFS

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression

lr = LogisticRegression()
model =
lr.setLabelCol("clicked").setFeaturesCol("features").fit(train)
model.save("hdfs://hadoop-master:9000/headlines/models/lr.obj")
```

加载训练好的 LR 模型，调用 `transform()` 对训练集做出预测（实际场景应该对验证集和训练集进行预测）

```
from pyspark.ml.classification import LogisticRegressionModel

online_model = LogisticRegressionModel.load("hdfs://hadoop-
master:9000/headlines/models/lr.obj")
sort_res = online_model.transform(train)
```

预测结果 `sort_res` 中包括 clicked 和 probability 列，其中 clicked 为样本标签的真实值，probability 是包含两个元素的列表，第一个元素是预测的不点击概率，第二个元素则是预测的点击概率，可以提取点击率（CTR）

```
def get_ctr(row):
    return float(row.clicked), float(row.probability[1])

score_label = sort_res.select(["clicked",
"probability"]).rdd.map(get_ctr)
```

模型评估

离线模型评估指标包括：

- 评分准确度 通常是均方根误差（RMSE），用来评估预测评分的效果
- 排序能力 通常采用 AUC（Area Under the Curve），即 ROC 曲线下方的面积
- 分类准确率（Precision） 表示在 Top K 推荐列表中，用户真实点击的物品所占的比例
- 分类召回率（Recall） 表示在用户真实点击的物品中，出现在 Top K 推荐列表中所占的比例

当模型更新后，还可以根据商业指标进行评估，比例类的包括： 点击率（CTR）、转化率（CVR），绝对类的包括：社交关系数量、用户停留时长、成交总额（GMV）等。

推荐系统的广度评估指标包括：

- 覆盖率 表示被有效推荐（推荐列表长度大于 c）的用户占全站用户的比例，公式如下：

$$Con_{UV} = \frac{N_{l>c}}{N_{UV}}$$

- 失效率 表示被无效推荐（推荐列表长度为 0）的用户占全站用户的比例，公式如下：

$$Lost_{UV} = \frac{N_{l=0}}{N_{UV}}$$

- 新颖性
- 更新率 表示推荐列表的变化程度，当前周期与上个周期相比，推荐列表中不同物品的比例

$$Update = \frac{N_{diff}}{N_{last}}$$

推荐系统的健康评估指标包括：

- 个性化 用于衡量推荐的个性化程度，是否大部分用户只消费小部分物品，可以计算所有用户推荐列表的平均相似度
- 基尼系数 用于衡量推荐系统的马太效应，反向衡量推荐的个性化程度。将物品按照累计推荐次数排序，排序位置为 i，推荐次数占总推荐次数的比例为 P_i，推荐次数越不平均，基尼系数越接近 1，公式为：

$$Gini = \frac{1}{n}\sum_{i=1}^{n} P_i(2i - n - i)$$

- 多样性 通常是在类别维度上衡量推荐结果的多样性，可以衡量各个类别在推荐时的熵

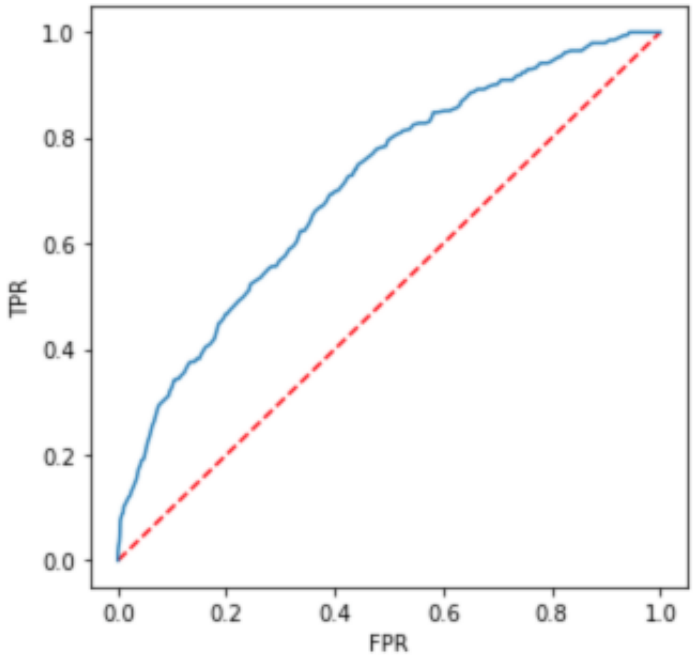
$$Div = \frac{\sum_{i=1}^{n} -P_i \log(P_i)}{n \log(n)}$$

其中，物品共包括 n 个类别，类别 i 被推荐次数占总推荐次数的比例为 P_i，分母是各个类别最均匀时对应的熵，分子是实际推荐结果的类别分布熵。这是整体推荐的多样性，还可以计算每次推荐和每个用户推荐的多样性。

我们这里主要根据 AUC 进行评估，首先利用 `model.summary.roc` 绘制 ROC 曲线

```
import matplotlib.pyplot as plt

plt.figure(figsize=(5,5))
plt.plot([0, 1], [0, 1], 'r--')
plt.plot(model.summary.roc.select('FPR').collect(),
         model.summary.roc.select('TPR').collect())
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.show()
```

ROC 曲线如下所示，曲线下面的面积即为 AUC（Area Under the Curve），AUC 值越大，排序效果越好



利用 Spark 的 `BinaryClassificationMetrics()` 计算 AUC

```
from pyspark.mllib.evaluation import BinaryClassificationMetrics

metrics = BinaryClassificationMetrics(score_label)
metrics.areaUnderROC
```

也可以利用 sklearn 的 `roc_auc_score()` 计算 AUC，`accuracy_score()` 计算准确率

```
from sklearn.metrics import accuracy_score, roc_auc_score,
import numpy as np

arr = np.array(score_label.collect())
# AUC
roc_auc_score(arr[:, 0], arr[:, 1]) # 0.719274521004087

# 准确率
accuracy_score(arr[:, 0], arr[:, 1].round()) # 0.9051438053097345
```

扫一扫关注微信公众号！专注于搜索和推荐系统，尝试使用算法去更好的服务于用户，包括但不局限于机器学习，深度学习，强化学习，自然语言理解，知识图谱，还不定时分享技术，资料，思考等文章！

12、基于 FTRL 模型的在线排序

构造 TFRecord 训练集

和前面的 LR 离线模型一样，FTRL 模型首先也是要完成训练集的构建。在上篇文章中，我们已经知道，可以通过读取用户历史行为数据，及文章特征和用户特征，构建出训练集 `train`，其中包括 features 和 label 两列数据，features 是文章特征和用户特征的组合。在 TensorFlow 通常使用 TFRecord 文件进行数据的存取。接下来，我们就要将 `train` 保存到 TFRecord 文件中。首先开启会话，将 `train` 中的特征和标签分别传入 `write_to_tfrecords()` 方法，并利用多线程执行

```
import tensorflow as tf

with tf.Session() as sess:
    # 创建线程协调器
    coord = tf.train.Coordinator()
    # 开启子线程去读取数据
    threads = tf.train.start_queue_runners(sess=sess, coord=coord)
    # 存入数据
    write_to_tfrecords(train.iloc[:, 0], train.iloc[:, 1])
    # 关闭子线程，回收
    coord.request_stop()
    coord.join(threads)
```

接着，在 `write_to_tfrecords()` 方法中，遍历训练集数据，将每个样本构造为 `tf.train.Example`，其中 feature 为 BytesList 类型，label 为 Int64List 类型，并保存到 TFRecords 文件中

```
def write_to_tfrecords(feature_batch, click_batch):
    """将用户与文章的点击日志构造的样本写入TFRecords文件
    """
    # 1、构造tfrecords的存储实例
    writer =
tf.python_io.TFRecordWriter("./train_ctr_20190605.tfrecords")

    # 2、循环将所有样本一个个封装成example，写入文件
    for i in range(len(click_batch)):
        # 取出第i个样本的特征值和目标值，格式转换
        click = click_batch[i]
        feature = feature_batch[i].tostring()
        # 构造example
        example =
tf.train.Example(features=tf.train.Features(feature={
            "feature":
tf.train.Feature(bytes_list=tf.train.BytesList(value=[feature])),
            "label":
tf.train.Feature(int64_list=tf.train.Int64List(value=[click]))
        }))
        # 序列化example,写入文件
        writer.write(example.SerializeToString())

    writer.close()
```

离线训练

FTRL（Follow The Regularized Leader）模型是一种获得稀疏模型的优化方法，我们利用构建好的 TFRecord 样本数据对 FTRL 模型进行离线训练。首先，定义 `read_ctr_records()` 方法来读取 TFRecord 文件，并通过调用 `parse_tfrecords()` 方法遍历解析每个样本，并设置了批大小和迭代次数

```
def read_ctr_records():
    train =
tf.data.TFRecordDataset(["./train_ctr_20190605.tfrecords"])
    train = train.map(parse_tfrecords)
    train = train.batch(64)
    train = train.repeat(10000)
```

解析每个样本，将 TFRecord 中序列化的 feature 列，解析成 channel_id (1), article_vector (100), user_weights (10), article_weights (10)

```
FEATURE_COLUMNS = ['channel_id', 'article_vector', 'user_weigths',
'article_weights']

def parse_tfrecords(example):
    features = {
        "feature": tf.FixedLenFeature([], tf.string),
        "label": tf.FixedLenFeature([], tf.int64)
    }
    parsed_features = tf.parse_single_example(example, features)
    feature = tf.decode_raw(parsed_features['feature'], tf.float64)
    feature = tf.reshape(tf.cast(feature, tf.float32), [1, 121])

    channel_id = tf.cast(tf.slice(feature, [0, 0], [1, 1]),
tf.int32)
    article_vector = tf.reduce_sum(tf.slice(feature, [0, 1], [1,
100]), axis=1)
    user_weights = tf.reduce_sum(tf.slice(feature, [0, 101], [1,
10]), axis=1)
    article_weights = tf.reduce_sum(tf.slice(feature, [0, 111], [1,
10]), axis=1)

    label = tf.cast(parsed_features['label'], tf.float32)

    # 构造字典 名称-tensor
    tensor_list = [channel_id, article_vector, user_weights,
article_weights]
    feature_dict = dict(zip(FEATURE_COLUMNS, tensor_list))

    return feature_dict, label
```

指定输入特征的数据类型，并定义 FTRL 模型 `model`

```
# 定义离散类型特征
article_id =
tf.feature_column.categorical_column_with_identity('channel_id',
num_buckets=25)
# 定义连续类型特征
article_vector = tf.feature_column.numeric_column('article_vector')
user_weigths = tf.feature_column.numeric_column('user_weigths')
article_weights =
tf.feature_column.numeric_column('article_weights')

feature_columns = [article_id, article_vector, user_weigths,
article_weights]

model =
tf.estimator.LinearClassifier(feature_columns=feature_columns,

optimizer=tf.train.FtrlOptimizer(learning_rate=0.1,

l1_regularization_strength=10,

l2_regularization_strength=10))
```

通过调用 `read_ctr_records()` 方法，来读取 TFRecod 文件中的训练数据，并设置训练步长，对定义好的 FTRL 模型进行训练及预估

```
model.train(read_ctr_records, steps=1000)
result = model.evaluate(read_ctr_records)
```

通常需要编写离线任务，定时读取用户行为数据作为训练集和验证集，对训练集及验证集进行 CTR 预估，并根据离线指标对结果进行分析，决定是否更新模型。

在线排序

通常在线排序是根据用户实时的推荐请求，对召回结果进行 CTR 预估，进而计算出排序结果并返回。我们需要根据召回结果构造测试集，其中每个测试样本包括用户特征和文章特征。首先，根据用户 ID 和频道 ID 读取用户特征（用户在每个频道的特征不同，所以是分频道存储的）

```
try:
    user_feature = eval(hbu.get_table_row('ctr_feature_user',
                                '{}'.format(temp.user_id).encode(),
                                'channel:
{}'.format(temp.channel_id).encode()))
except Exception as e:
    user_feature = []
```

再根据用户 ID 读取召回结果

```
recall_set = read_hbase_recall('cb_recall',
                'recall:user:{}'.format(temp.user_id).encode(),
                'als:{}'.format(temp.channel_id).encode())
```

接着，遍历召回结果，获取文章特征，并将用户特征合并，作为测试样本

```
test = []
for article_id in recall_set:
    try:
        article_feature =
eval(hbu.get_table_row('ctr_feature_article',
                                    '{}'.format(article_id).encode(),
                                    'article:
{}'.format(article_id).encode()))
    except Exception as e:
        article_feature = []

    if not article_feature:
        article_feature = [0.0] * 111
    feature = []
    feature.extend(user_feature)
    feature.extend(article_feature)

    test.append(f)
```

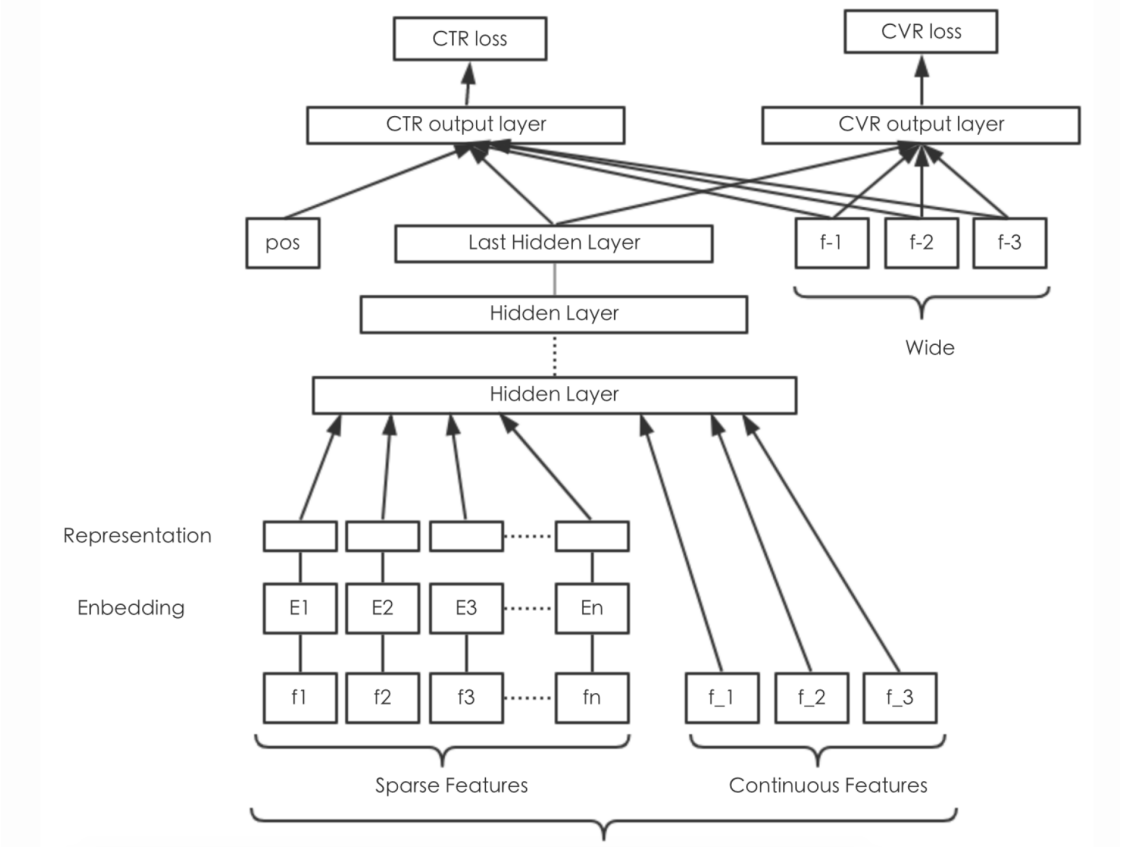
加载本地 FTRL 模型并对测试样本进行 CTR 预估

```
test_array = np.array(test)
model.load_weights('/root/toutiao_project/reco_sys/offline/models/c
kpt/ctr_lr_ftrl.h5')
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
    predictions =
self.model.predict(sess.run(tf.constant(test_array)))
```

对结果进行排序并提取 CTR 最高的前 K 个文章，这样就得到了 FTRL 模型在线排序的结果。

```
res =
pd.DataFrame(np.concatenate((np.array(recall_set).reshape(len(recal
l_set), 1), predictions),
                                    axis=1), columns=['article_id',
'prob'])

res_sort = res.sort_values(by=['prob'], ascending=True)

# 排序后，只将排名在前100个文章ID作为推荐结果返回给用户
if len(res_sort) > 100:
    recall_set = list(res_sort.iloc[:100, 0])
recall_set = list(res_sort.iloc[:, 0])
```

13、基于 Wide&Deep 模型的在线排序



上图是 Wide&Deep 模型的网络结构，深度学习可以通过嵌入（Embedding）表达出更精准的用户兴趣及物品特征，不仅能减少人工特征工程的工作量，还能提高模型的泛化能力，使得用户行为预估更加准确。Wide&Deep 模型适合高维稀疏特征的推荐场景，兼有稀疏特征的可解释性和深模型的泛化能力。通常将类别特征做 Embedding 学习，再将 Embedding 稠密特征输入深模型中。Wide 部分的输入特征包括：类别特征和离散化的数值特征，Deep部分的输入特征包括：数值特征和 Embedding 后的类别特征。其中，Wide 部分使用 FTRL + L1；Deep 部分使用 AdaGrad，并且两侧是一起联合进行训练的。

离线训练

TensorFlow 实现了很多深度模型，其中就包括 Wide&Deep，API 接口为 `tf.estimator.DNNLinearCombinedClassifier`，我们可以直接使用。在上篇文章中已经实现了将训练数据写入 TFRecord 文件，在这里可以直接读取

```
@staticmethod
def read_ctr_records():
    dataset =
tf.data.TFRecordDataset(["./train_ctr_201905.tfrecords"])
    dataset = dataset.map(parse_tfrecords)
    dataset = dataset.shuffle(buffer_size=10000)
    dataset = dataset.repeat(10000)
    return dataset.make_one_shot_iterator().get_next()
```

解析每个样本，将 TFRecord 中序列化的 feature 列，解析成 channel_id (1), article_vector (100), user_weights (10), article_weights (10)

```
def parse_tfrecords(example):
    features = {
        "label": tf.FixedLenFeature([], tf.int64),
        "feature": tf.FixedLenFeature([], tf.string)
    }
    parsed_features = tf.parse_single_example(example, features)

    feature = tf.decode_raw(parsed_features['feature'], tf.float64)
    feature = tf.reshape(tf.cast(feature, tf.float32), [1, 121])
    # 特征顺序 1 channel_id,  100 article_vector, 10 user_weights, 10
article_weights
    # 1 channel_id类别型特征,  100维文章向量求平均值当连续特征，10维用户权重
求平均值当连续特征
    channel_id = tf.cast(tf.slice(feature, [0, 0], [1, 1]),
tf.int32)
    vector = tf.reduce_sum(tf.slice(feature, [0, 1], [1, 100]),
axis=1, keep_dims=True)
    user_weights = tf.reduce_sum(tf.slice(feature, [0, 101], [1,
10]), axis=1, keep_dims=True)
    article_weights = tf.reduce_sum(tf.slice(feature, [0, 111], [1,
10]), axis=1, keep_dims=True)

    label = tf.reshape(tf.cast(parsed_features['label'],
tf.float32), [1, 1])

    # 构造字典 名称-tensor
    FEATURE_COLUMNS = ['channel_id', 'vector', 'user_weigths',
'article_weights']
    tensor_list = [channel_id, vector, user_weights,
article_weights]

    feature_dict = dict(zip(FEATURE_COLUMNS, tensor_list))

    return feature_dict, label
```

指定输入特征的数据类型，并定义 Wide&Deep 模型 model

```
# 离散类型
channel_id =
tf.feature_column.categorical_column_with_identity('channel_id',
num_buckets=25)
# 连续类型
vector = tf.feature_column.numeric_column('vector')
user_weigths = tf.feature_column.numeric_column('user_weigths')
article_weights =
tf.feature_column.numeric_column('article_weights')

wide_columns = [channel_id]

# embedding_column用来表示类别型的变量
deep_columns = [tf.feature_column.embedding_column(channel_id,
dimension=25),
                vector, user_weigths, article_weights]

estimator =
tf.estimator.DNNLinearCombinedClassifier(model_dir="./ckpt/wide_and
_deep",

linear_feature_columns=wide_columns,

dnn_feature_columns=deep_columns,

dnn_hidden_units=[1024, 512, 256])
```

通过调用 read_ctr_records() 方法，来读取 TFRecod 文件中的训练数据，并设置训练步长，对定义好的 FTRL 模型进行训练及预估

```
model.train(read_ctr_records, steps=1000)
result = model.evaluate(read_ctr_records)
```

可以用上一次模型的参数作为当前模型的初始化参数，训练完成后，通常会进行离线指标分析，若符合预期即可导出模型

```
columns = wide_columns + deep_columns
feature_spec = tf.feature_column.make_parse_example_spec(columns)
serving_input_receiver_fn =
tf.estimator.export.build_parsing_serving_input_receiver_fn(feature
_spec)
model.export_savedmodel("./serving_model/wdl/",
serving_input_receiver_fn)
```

TFServing 部署

安装

```
docker pull tensorflow/serving
```

启动

```
docker run -p 8501:8501 -p 8500:8500 --mount
type=bind,source=/root/toutiao_project/reco_sys/server/models/servi
ng_model/wdl,target=/models/wdl -e MODEL_NAME=wdl -t
tensorflow/serving
```

- -p 8501:8501 为端口映射（-p 主机端口 : docker 容器程序）
- TFServing 使用 8501 端口对外提供 HTTP 服务，使用8500对外提供 gRPC 服务，这里同时开放了两个端口的使用
- --mount type=bind,source=/home/ubuntu/detectedmodel/wdl,target=/models/wdl 为文件映射，将主机（source）的模型文件映射到 docker 容器程序（target）的位置，以便 TFServing 使用模型，target 参数为 /models/模型名称
- -e MODEL_NAME= wdl 设置了一个环境变量，名为 MODEL_NAME，此变量被 TFServing 读取，用来按名字寻找模型，与上面 target 参数中的模型名称对应
- -t 为 TFServing 创建一个伪终端，供程序运行
- tensorflow/serving 为镜像名称

在线排序

通常在线排序是根据用户实时的推荐请求，对召回结果进行 CTR 预估，进而计算出排序结果并返回。我们需要根据召回结果构造测试集，其中每个测试样本包括用户特征和文章特征。首先，根据用户 ID 和频道 ID 读取用户特征（用户在每个频道的特征不同，所以是分频道存储的）

```
try:
    user_feature = eval(hbu.get_table_row('ctr_feature_user',
                                '{}'.format(temp.user_id).encode(),
                                'channel:
{}'.format(temp.channel_id).encode()))
except Exception as e:
    user_feature = []
```

再根据用户 ID 读取召回结果

```
recall_set = read_hbase_recall('cb_recall',
                'recall:user:{}'.format(temp.user_id).encode(),
                'als:{}'.format(temp.channel_id).encode())
```

接着，遍历召回结果，获取文章特征，并将用户特征合并，构建样本

```
examples = []
for article_id in recall_set:
    try:
        article_feature =
eval(hbu.get_table_row('ctr_feature_article',
                                    '{}'.format(article_id).encode(),
                                    'article:
{}'.format(article_id).encode()))
    except Exception as e:
        article_feature = []

    if not article_feature:
        article_feature = [0.0] * 111

    channel_id = int(article_feature[0])
    # 计算后面若干向量的平均值
    vector = np.mean(article_feature[11:])
    # 用户权重特征
    user_feature = np.mean(user_feature)
    # 文章权重特征
    article_feature = np.mean(article_feature[1:11])

    # 构建example
    example = tf.train.Example(features=tf.train.Features(feature={
                "channel_id":
tf.train.Feature(int64_list=tf.train.Int64List(value=
[channel_id])),
                "vector":
tf.train.Feature(float_list=tf.train.FloatList(value=[vector])),
                'user_weigths':
tf.train.Feature(float_list=tf.train.FloatList(value=
[user_feature])),
                'article_weights':
tf.train.Feature(float_list=tf.train.FloatList(value=
[article_feature])),
            }))

    examples.append(example)
```

调用 TFServing 的模型服务，获取排序结果

```
with grpc.insecure_channel("127.0.0.1:8500") as channel:
    stub =
prediction_service_pb2_grpc.PredictionServiceStub(channel)
    request = classification_pb2.ClassificationRequest()
    # 构造请求，指定模型名称，指定输入样本
    request.model_spec.name = 'wdl'
    request.input.example_list.examples.extend(examples)
    # 发送请求，获取排序结果
    response = stub.Classify(request, 10.0)
```

这样，我们就实现了 Wide&Deep 模型的离线训练和 TFServing 模型部署以及在线排序服务的调用。使用这种方式，线上服务需要将特征发送给TF Serving，这不可避免引入了网络 IO，给带宽和预估时延带来压力。可以通过并发请求，召回多个召回结果集合，然后并发请求 TF Serving 模型服务，这样可以有效降低整体预估时延。还可以通过特征 ID 化，将字符串类型的特征名哈希到 64 位整型空间，这样有效减少传输的数据量，降低使用的带宽。

模型同步

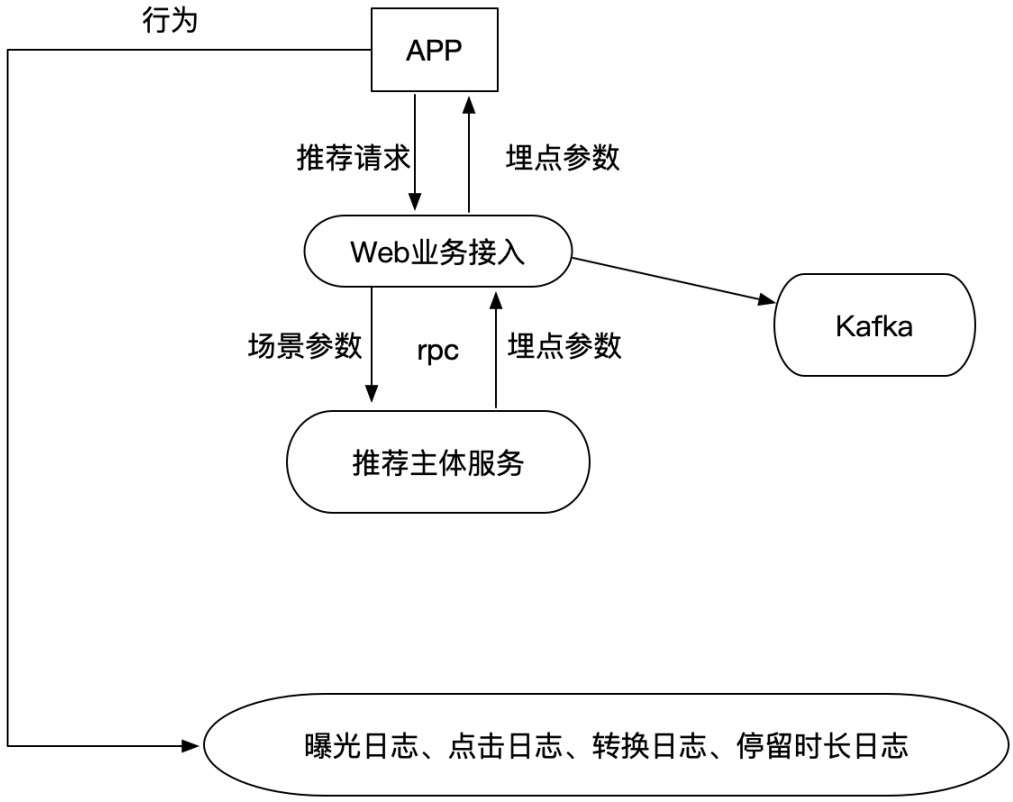
实际环境中，我们可能还要经常将离线训练好的模型同步到线上服务机器，大致同步过程如下：

- 同步前，检查模型 md5 文件，只有该文件更新了，才需要同步
- 同步时，随机链接 HTTPFS 机器并限制下载速度
- 同步后，校验模型文件 md5 值并备份旧模型

同步过程中，需要处理发生错误或者超时的情况，可以设定触发报警或重试机制。通常模型的同步时间都在分钟级别。

14、推荐中心

在前面的文章中，我们实现了召回和排序，接下来将进入推荐逻辑处理阶段，通常称为推荐中心，推荐中心负责接收应用系统的推荐请求，读取召回和排序的结果并进行调整，最后返回给应用系统。推荐中心的调用流程如下所示：



推荐接口设计

通常推荐接口包括 Feed 流推荐和相似文章推荐

- Feed 流推荐：根据用户偏好，获取推荐文章列表（这里的时间戳用于区分是刷新推荐列表还是查看历史推荐列表） 参数：用户 ID，频道 ID，推荐文章数量，请求推荐的时间戳 结果：曝光参数，每篇文章的行为埋点参数，上一条推荐的时间戳
- 相似文章推荐：当用户浏览某文章时，获取该文章的相似文章列表 参数：文章 ID，推荐文章数量 结果：文章 ID 列表

行为埋点参数：

```
{
    "param": '{"action": "exposure", "userId": 1, "articleId":
[1,2,3,4],  "algorithmCombine": "c1"}',
    "recommends": [
        {"article_id": 1, "param": {"click": "{"action": "click",
"userId": "1", "articleId": 1, "algorithmCombine": 'c1'}",
"collect": "...", "share": "...","read":"..."}},
        {"article_id": 2, "param": {"click": "...", "collect":
"...", "share": "...", "read":"..."}},
        {"article_id": 3, "param": {"click": "...", "collect":
"...", "share": "...", "read":"..."}},
        {"article_id": 4, "param": {"click": "...", "collect":
"...", "share": "...", "read":"..."}}
    ]
    "timestamp": 1546391572
}
```

这里接口采用 gRPC 框架，在 user_reco.proto 文件中定义 Protobuf 序列化协议，其中定义了 Feed 流推荐接口：`rpc user_recommend(User) returns (Track) {}` 和相似文章接口：`rpc article_recommend(Article) returns(Similar) {}`

```
syntax = "proto3";

message User {
    string user_id = 1;
    int32 channel_id = 2;
    int32 article_num = 3;
    int64 time_stamp = 4;
}
// int32 ---> int64 article_id
message Article {
    int64 article_id = 1;
    int32 article_num = 2;

}

message param2 {
    string click = 1;
    string collect = 2;
    string share = 3;
    string read = 4;
}

message param1 {
    int64 article_id = 1;
    param2 params = 2;
}

message Track {
    string exposure = 1;
    repeated param1 recommends = 2;
    int64 time_stamp = 3;
}

message Similar {
    repeated int64 article_id = 1;
}

service UserRecommend {
    rpc user_recommend(User) returns (Track) {}
    rpc article_recommend(Article) returns(Similar) {}
}
```

接着，通过如下命令生成服务端文件 user_reco_pb2.py 和客户端文件 user_reco_pb2_grpc.py

```
python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=.
user_reco.proto
```

定义参数解析类，用于解析推荐请求的参数，包括用户 ID、频道 ID、文章数量、请求时间戳以及算法名称

```
class Temp(object):
    user_id = -10
    channel_id = -10
    article_num = -10
    time_stamp = -10
    algo = ""
```

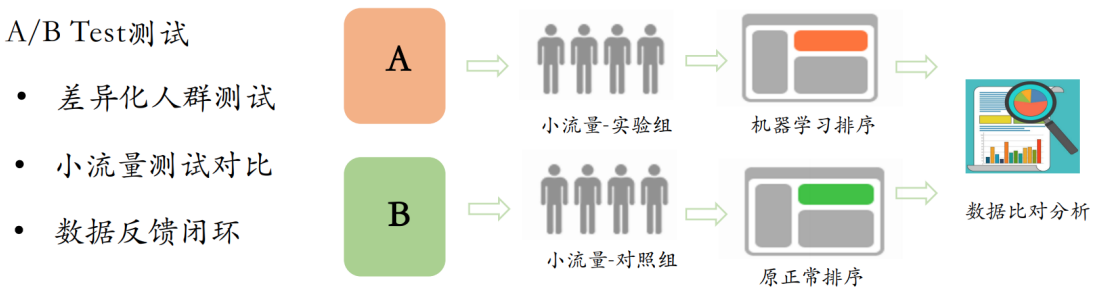
定义封装埋点参数方法，其中参数 `res` 为推荐结果，参数 `temp` 为用户请求参数，将推荐结果封装为在 user_reco.proto 文件中定义的 Track 结构，其中携带了文章对埋点参数，包括了事件名称、算法名称以及时间等等，方便后面解析用户对文章对行为信息

```
def add_track(res, temp):
    """
    封装埋点参数
    :param res: 推荐文章id列表
    :param temp: rpc参数
    :return: 埋点参数
        文章列表参数
        单文章参数
    """
    # 添加埋点参数
    track = {}

    # 准备曝光参数
    # 全部字符串形式提供，在hive端不会解析问题
    _exposure = {"action": "exposure", "userId": temp.user_id,
"articleId": json.dumps(res),
                 "algorithmCombine": temp.algo}

    track['param'] = json.dumps(_exposure)
    track['recommends'] = []

    # 准备其它点击参数
    for _id in res:
        # 构造字典
```

```
            _dic = {}
            _dic['article_id'] = _id
            _dic['param'] = {}

            # 准备click参数
            _p = {"action": "click", "userId": temp.user_id,
"articleId": str(_id),
                  "algorithmCombine": temp.algo}

            _dic['param']['click'] = json.dumps(_p)
            # 准备collect参数
            _p["action"] = 'collect'
            _dic['param']['collect'] = json.dumps(_p)
            # 准备share参数
            _p["action"] = 'share'
            _dic['param']['share'] = json.dumps(_p)
            # 准备detentionTime参数
            _p["action"] = 'read'
            _dic['param']['read'] = json.dumps(_p)

            track['recommends'].append(_dic)

        track['timestamp'] = temp.time_stamp
        return track
```

AB Test 流量切分

由于推荐算法和策略是需要不断改进和完善等，所以 ABTest 也是推荐系统不可或缺的功能。可以根据用户 ID 将流量切分为多个桶（Bucket），每个桶对应一种排序策略，桶内流量将使用相应的策略进行排序，使用 ID 进行流量切分能够保证用户体验的一致性。通常 ABTest 过程如下所示：



通过定义 AB Test 参数，可以实现为不同的用户使用不同的推荐算法策略，其中 `COMBINE` 为融合方式， `RECALL` 为召回方式， `SORT` 为排序方式， `CHANNEL` 为频道数量， `BYPASS` 为分桶设置， `sort_dict` 为不同的排序服务对象。可以看到 Algo-1 使用 LR 进行排序，而 Algo-2 使用 Wide&Deep 进行排序

```
from collections import namedtuple

# ABTest参数信息
param = namedtuple('RecommendAlgorithm', ['COMBINE',
                                          'RECALL',
                                          'SORT',
                                          'CHANNEL',
                                          'BYPASS']
                   )

RAParam = param(
    COMBINE={
        'Algo-1': (1, [100, 101, 102, 103, 104], [200]),  # 首页推
荐，所有召回结果读取+LR排序
        'Algo-2': (2, [100, 101, 102, 103, 104], [201])  # 首页推荐，
所有召回结果读取 排序
    },
    RECALL={
        100: ('cb_recall', 'als'),  # 离线模型ALS召
回，recall:user:1115629498121 column=als:18
        101: ('cb_recall', 'content'),  # 离线word2vec的画像内容召回
'recall:user:5', 'content:1'
        102: ('cb_recall', 'online'),  # 在线word2vec的画像召回
'recall:user:1', 'online:1'
        103: 'new_article',  # 新文章召回 redis当中    ch:18:new
        104: 'popular_article',  # 基于用户协同召回结果 ch:18:hot
        105: ('article_similar', 'similar')  # 文章相似推荐结果 '1'
'similar:2'
    },
    SORT={
        200: 'LR',
        201: 'WDL‘
    },
    CHANNEL=25,
    BYPASS=[
            {
                "Bucket": ['0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', 'a', 'b', 'c', 'd'],
                "Strategy": "Algo-1"
            },
            {
```

```
                "BeginBucket": ['e', 'f'],
                "Strategy": "Algo-2"
            }
        ]
)

sort_dict = {
    "LR": lr_sort_service,
    "WDL": wdl_sort_service
}
```

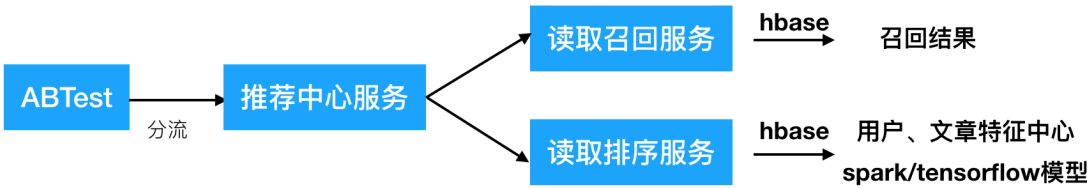
流量切分，将用户 ID 进行哈希，然后取哈希结果的第一个字符，将包含该字符的策略桶所对应的算法编号赋值到此用户请求参数的 `algo` 属性中，后面将调用该编号对应的算法策略为此用户计算推荐数据

```
import hashlib
from setting.default import DefaultConfig, RAParam

# 进行分桶实现分流，制定不同的实验策略
bucket = hashlib.md5(user_id.encode()).hexdigest()[:1]
if bucket in RAParam.BYPASS[0]['Bucket']:
    temp.algo = RAParam.BYPASS[0]['Strategy']
else:
    temp.algo = RAParam.BYPASS[1]['Strategy']
```

推荐中心逻辑

推荐中心逻辑主要包括：

- 接收应用系统发送的推荐请求，解析请求参数
- 进行 ABTest 分流，为用户分配推荐策略
- 根据分配的算法调用召回服务和排序服务，读取推荐结果
- 根据业务进行调整，如过滤、补足、合并信息等
- 封装埋点参数，返回推荐结果



首先，在 Hbase 中创建历史推荐结果表 history_recommend，用于存储用户历史推荐结果

```
create 'history_recommend', {NAME=>'channel', TTL=>7776000,
VERSIONS=>999999}   86400
# 每次指定一个时间戳,可以达到不同版本的效果
put 'history_recommend', 'reco:his:1', 'channel:18', [17283,
140357, 14668, 15182, 17999, 13648, 12884,18135]
```

继续在 Hbase 中创建待推荐结果表 wait_recommend，用于存储经过多路召回并且排序之后的待推荐结果，当 wait_recommend 没有数据时，才再次调用排序服务计算出新的待推荐结果并写入到 wait_recommend，所以不需设置多个版本。注意该表与 cb_recall 的区别，cb_recall 存储的是还未经排序的召回结果。

```
create 'wait_recommend', 'channel'

put 'wait_recommend', 'reco:1', 'channel:18', [17283, 140357,
14668, 15182, 17999, 13648, 12884,18135]
put 'wait_recommend', 'reco:1', 'channel:0', [17283, 140357, 14668,
15182, 17999, 13648, 12884, 17302, 13846]
```

用户获取 Feed 流推荐数据时，如果用户向下滑动，发出的是刷新推荐列表的请求，需要传入当前时间作为请求时间戳参数，该请求时间戳必然大于 Hbase 历史推荐结果表中的请求时间戳，那么程序将获取新的推荐列表，并返回 Hbase 历史推荐结果表中最近一次推荐的请求时间戳，用于查询历史推荐结果；如果用户向上滑动，发出的是查看历史推荐结果的请求，需要传入前面刷新推荐列表时返回的最近一次推荐的请求时间戳，该请求时间戳必然小于等于 Hbase 历史推荐结果中最近一次推荐的时间戳，那么程序将获取小于等于该请求时间戳的最近一次历史推荐结果，并返回小于该推荐结果最近一次推荐的时间戳，也就是上一次推荐的时间戳，下面是具体实现。

在获取推荐列表时，首先获取用户的历史数据库中最近一次时间戳 `last_stamp`，没有则将 `last_stamp` 置为 0

```
try:
    last_stamp = self.hbu.get_table_row('history_recommend',
                                        'reco:his:
{}'.format(temp.user_id).encode(),
                                        'channel:
{}'.format(temp.channel_id).encode(),
                                        include_timestamp=True)[1]
except Exception as e:
    last_stamp = 0
```

- 如果用户请求的时间戳小于历史推荐结果中最近一次请求的时间戳 `last_stamp`，那么该请求为用户获取历史推荐结果 1.如果没有历史推荐结果，则返回时间戳 `0` 以及空列表 `[]` 2.如果历史推荐结果只有一条，则返回这一条历史推荐结果并返回时间戳 `0`，表示已经没有历史推荐结果（APP 可以显示已经没有历史推荐记录了） 3.如果历史推荐结果有多条，则返回历史推荐结果中第一条推荐结果（最近一次），然后返回历史推荐结果中第二条推荐结果的时间戳

```
if temp.time_stamp < last_stamp:
    try:
        row = self.hbu.get_table_cells('history_recommend',
                                       'reco:his:
{}'.format(temp.user_id).encode(),
                                       'channel:
{}'.format(temp.channel_id).encode(),
                                       timestamp=temp.time_stamp +
1,
                                       include_timestamp=True)
    except Exception as e:
        row = []
        res = []

    if not row:
        temp.time_stamp = 0
        res = []
    elif len(row) == 1 and row[0][1] == temp.time_stamp:
        res = eval(row[0][0])
        temp.time_stamp = 0
    elif len(row) >= 2:
        res = eval(row[0][0])
        temp.time_stamp = int(row[1][1])

    res = list(map(int, res))
    # 封装推荐结果
    track = add_track(res, temp)
    # 曝光参数设置为空
    track['param'] = ''
```

（注意：这里将用户请求的时间戳 +1，因为 Hbase 只能获取小于该时间戳的历史推荐结果）

- 如果用户请求的时间戳大于 Hbase 历史推荐结果中最近一次请求的时间戳 `last_stamp`，那么该请求为用户刷新推荐列表，需要读取推荐结果并返回。如果结果为空，需要调用 `user_reco_list()` 方法，再次计算推荐结果，再返回。

```
if temp.time_stamp > last_stamp:
    # 获取缓存
    res = redis_cache.get_reco_from_cache(temp, self.hbu)
    # 如果结果为空，需要再次计算推荐结果 进行召回+排序，同时写入到hbase待推荐结果列表
    if not res:
        res = self.user_reco_list(temp)

    temp.time_stamp = int(last_stamp)
    track = add_track(res, temp)
```

定义 `user_reco_list()` 方法，首先要读取多路召回结果，根据为用户分配的算法策略，读取相应路径的召回结果，并进行重后合并

```
reco_set = []
# (1, [100, 101, 102, 103, 104], [])
for number in RAParam.COMBINE[temp.algo][1]:
    if number == 103:
        _res = self.recall_service.read_redis_new_article(temp.channel_id)
        reco_set = list(set(reco_set).union(set(_res)))
    elif number == 104:
        _res = self.recall_service.read_redis_hot_article(temp.channel_id)
        reco_set = list(set(reco_set).union(set(_res)))
    else:
        # 100, 101, 102召回结果读取
        _res = self.recall_service.read_hbase_recall(RAParam.RECALL[number][0],
                                                     'recall:user:{}'.format(temp.user_id).encode(),
                                                     '{}:{}'.format(RAParam.RECALL[number][1],
                                                                    temp.channel_id).encode())
        reco_set = list(set(reco_set).union(set(_res)))
```

接着，过滤当前该请求频道的历史推荐结果，如果不是 0 频道还需过滤 0 频道的历史推荐结果

```
history_list = []
data = self.hbu.get_table_cells('history_recommend',
                                'reco:his:
{}'.format(temp.user_id).encode(),
                                'channel:
{}'.format(temp.channel_id).encode())

for _ in data:
    history_list = list(set(history_list).union(set(eval(_))))

data = self.hbu.get_table_cells('history_recommend',
                                'reco:his:
{}'.format(temp.user_id).encode(),
                                'channel:{}'.format(0).encode())

for _ in data:
    history_list = list(set(history_list).union(set(eval(_))))

reco_set = list(set(reco_set).difference(set(history_list)))
```

最后，根据分配的算法策略，调用排序服务，将分数最高的 N 个推荐结果返回，并写入历史推荐结果表，如果还有剩余的排序结果，将其余写入待推荐结果表

```
# 使用指定模型对召回结果进行排序
# temp.user_id,  reco_set
_sort_num = RAParam.COMBINE[temp.algo][2][0]
# 'LR'
reco_set = sort_dict[RAParam.SORT[_sort_num]](reco_set, temp,
self.hbu)

if not reco_set:
    return reco_set
else:

    # 如果reco_set小于用户需要推荐的文章
    if len(reco_set) <= temp.article_num:
        res = reco_set
    else:
        # 大于要推荐的文章结果
        res = reco_set[:temp.article_num]

        # 将剩下的文章列表写入待推荐的结果
        self.hbu.get_table_put('wait_recommend',
                               'reco:
{}'.format(temp.user_id).encode(),
                               'channel:
{}'.format(temp.channel_id).encode(),

str(reco_set[temp.article_num:]).encode(),
                               timestamp=temp.time_stamp)

    # 直接写入历史记录当中，表示这次又成功推荐一次
    self.hbu.get_table_put('history_recommend',
                           'reco:his:
{}'.format(temp.user_id).encode(),
                           'channel:
{}'.format(temp.channel_id).encode(),
                           str(res).encode(),
                           timestamp=temp.time_stamp)

    return res
```

到这里，推荐中心的基本逻辑已经结束。下面是读取多路召回结果的实现细节：通过指定列族，读取基于模型、离线内容以及在线的召回结果，并删除 cb_recall 的召回结果

```
def read_hbase_recall_data(self, table_name, key_format,
column_format):
    """
    读取cb_recall当中的推荐数据
    读取的时候可以选择列族进行读取als, online, content

    :return:
    """
    recall_list = []
    data = self.hbu.get_table_cells(table_name, key_format,
column_format)
    # data是多个版本的推荐结果[[],[],[],]
    for _ in data:
        recall_list = list(set(recall_list).union(set(eval(_))))
    self.hbu.get_table_delete(table_name, key_format,
column_format)
    return recall_list
```

读取 redis 中的新文章

```
def read_redis_new_article(self, channel_id):
    """
    读取新文章召回结果
    :param channel_id: 提供频道
    :return:
    """
    _key = "ch:{}:new".format(channel_id)
    try:
        res = self.client.zrevrange(_key, 0, -1)
    except Exception as e:
        res = []

    return list(map(int, res))
```

读取 redis 中的热门文章，并选取热度最高的前 K 个文章

```
def read_redis_hot_article(self, channel_id):
    """
    读取热门文章召回结果
    :param channel_id: 提供频道
    :return:
    """
    _key = "ch:{}:hot".format(channel_id)
    try:
        res = self.client.zrevrange(_key, 0, -1)
    except Exception as e:

    # 由于每个频道的热门文章有很多，因为 保留文章点击次数
    res = list(map(int, res))
    if len(res) > self.hot_num:
        res = res[:self.hot_num]
    return res
```

读取相似文章

```
def read_hbase_article_similar(self, table_name, key_format,
article_num):
    """获取文章相似结果
    :param article_id: 文章id
    :param article_num: 文章数量
    :return:
    """
    try:
        _dic = self.hbu.get_table_row(table_name, key_format)

        res = []
        _srt = sorted(_dic.items(), key=lambda obj: obj[1],
reverse=True)
        if len(_srt) > article_num:
            _srt = _srt[:article_num]
        for _ in _srt:
            res.append(int(_[0].decode().split(':')[1]))
    except Exception as e:
        res = []
    return res
```

使用缓存策略

- 如果 redis 缓存中存在数据，就直接从 redis 缓存中获取推荐结果
- 如果 redis 缓存为空而 Hbase 的待推荐结果表 wait_recommend 不为空，则从 wait_recommend 中获取推荐结果，并将一定数量的待推荐结果放入 redis 缓存中
- 若 redis 和 wait_recommend 都为空，则需读取召回结果并进行排序，将排序结果写入 Hbase 的待推荐结果表 wait_recommend 中及 redis 中

（每次读取的推荐结果都要将其写入 Hbase 的历史推荐结果表 history_recommend 中）

读取 redis 缓存

```
#读取redis对应的键
key = 'reco:{}:{}:art'.format(temp.user_id, temp.channel_id)
# 读取，删除，返回结果
pl = cache_client.pipeline()

# 读取redis数据
res = cache_client.zrevrange(key, 0, temp.article_num - 1)
if res:
    # 手动删除读取出来的缓存结果
    pl.zrem(key, *res)
```

如果 redis 缓存为空

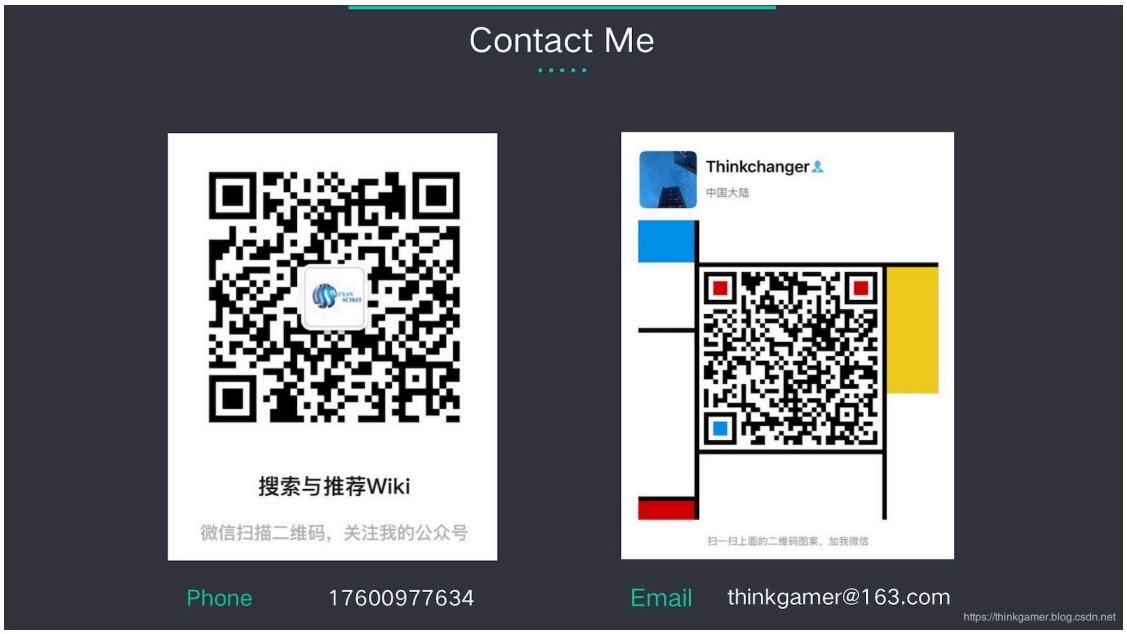
```
else:
    # 删除键
    cache_client.delete(key)
    try:
        # 从wait_recommend中读取
        wait_cache = eval(hbu.get_table_row('wait_recommend',
                                            'reco:
{}'.format(temp.user_id).encode(),
                                            'channel:
{}'.format(temp.channel_id).encode()))
    except Exception as e:
        wait_cache = []

    # 如果为空则直接返回空
    if not wait_cache:
        return wait_cache

    # 如果wait_recommend中有数据
    if len(wait_cache) > 100:
        cache_redis = wait_cache[:100]

        # 前100个数据放入redis
        pl.zadd(key, dict(zip(cache_redis,
range(len(cache_redis)))))

        # 100个后面的数据，在放回wait_recommend
        hbu.get_table_put('wait_recommend',
                          'reco:{}'.format(temp.user_id).encode(),
                          'channel:
{}'.format(temp.channel_id).encode(),
                          str(wait_cache[100:]).encode())

    else:
        # 清空wait_recommend数据
        hbu.get_table_put('wait_recommend',
                          'reco:{}'.format(temp.user_id).encode(),
                          'channel:
{}'.format(temp.channel_id).encode(),
                          str([]).encode())

        # 所有不足100个数据，放入redis
        pl.zadd(key, dict(zip(wait_cache, range(len(wait_cache)))))

    res = cache_client.zrange(key, 0, temp.article_num - 1)
```

最后，在 Supervisor 中配置 gRPC 实时推荐程序

```
[program:online]
environment=JAVA_HOME=/root/bigdata/jdk,SPARK_HOME=/root/bigdata/sp
ark,HADOOP_HOME=/root/bigdata/hadoop,PYSPARK_PYTHON=/miniconda2/env
s/reco_sys/bin/python
,PYSPARK_DRIVER_PYTHON=/miniconda2/envs/reco_sys/bin/python
command=/miniconda2/envs/reco_sys/bin/python
/root/toutiao_project/reco_sys/abtest/routing.py
directory=/root/toutiao_project/reco_sys/abtest
user=root
autorestart=true
redirect_stderr=true
stdout_logfile=/root/logs/recommendsuper.log
loglevel=info
stopsignal=KILL
stopasgroup=true
killasgroup=true
```

参考

- https://www.bilibili.com/video/av68356229
- https://pan.baidu.com/s/1-uvGJ-mEskjhtaial0Xmgw（学习资源已保存至网盘， 提取码：eakp）



扫一扫 关注微信公众号！专注于搜索和推荐系统，尝试使用算法去更好的服务于用户，包括但不局限于机器学习，深度学习，强化学习，自然语言理解，知识图谱，还不定时分享技术，资料，思考等文章！