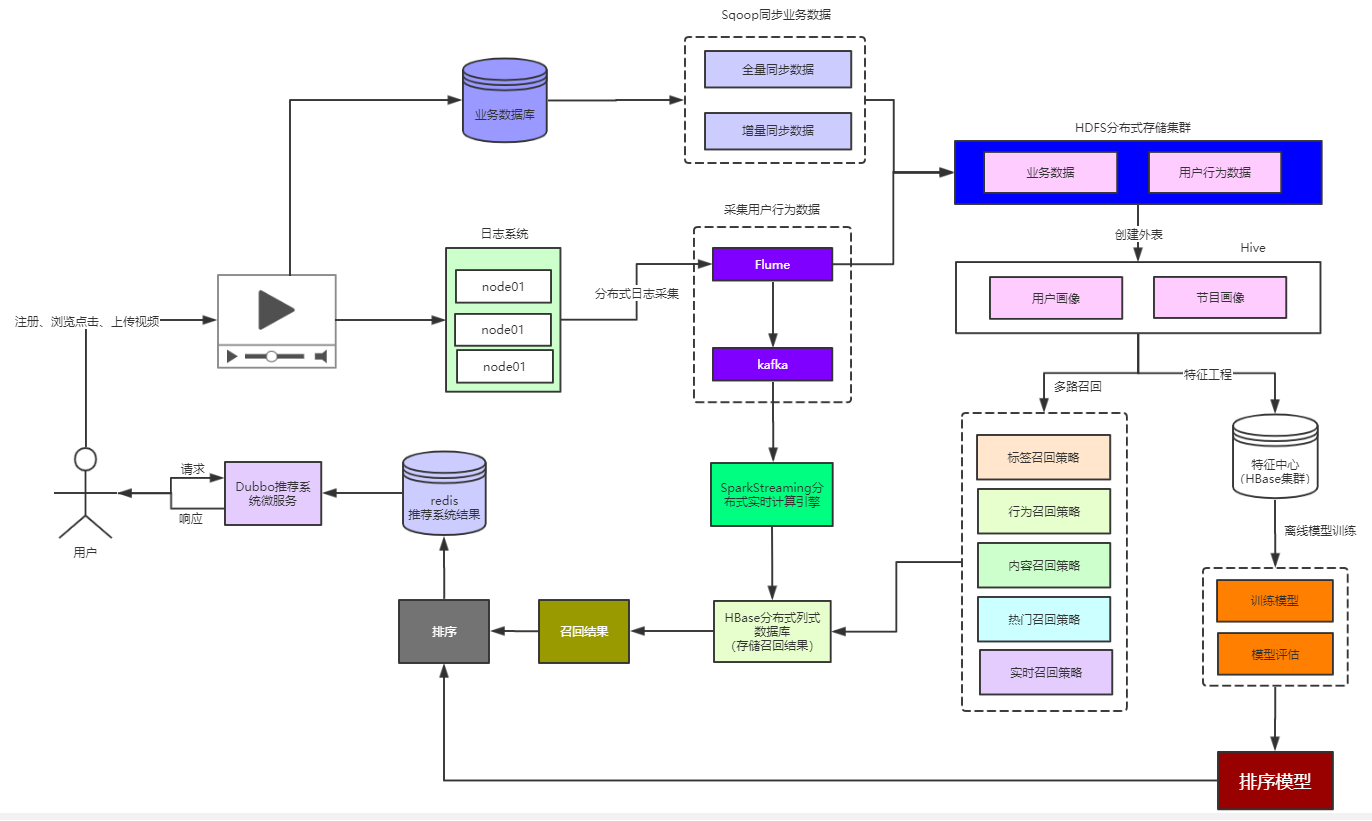
## 基于节目的个性化实时推荐系统开发流程

### 推荐系统架构图

基于lambda架构的离线与实时计算框架对用户行为数据、业务数据构建用户画像与节目画像，使用机器学习算法模型预测用户与节目之间的关系，为用户进行更加智能的节目推送

架构：

1. 行为数据采集
2. 业务数据定时全量、增量同步
3. 离线计算：用户画像、节目画像
4. 在线计算：实时召回、热门召回
5. Dubbo对外提供推荐服务





### 准备开发环境

准备虚拟机：node01、node02、node03、node04

分别搭建HDFS、Spark、Hive、HBase、Flume、Kafka、Sqoop

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 虚拟机名称 | HDFS | Spark | Hive | HBase | Flume | Sqoop | Kafka | Yarn |
| node01 | √ | √ | √ | √ | √ | √ | √ | √ |
| node02 | √ | √ | ❌ | √ | ❌ | ❌ | √ | √ |
| node03 | √ | √ | ❌ | √ | ❌ | ❌ | √ | √ |
| node04 | √ | √ | ❌ | √ | ❌ | ❌ |  | √ |

存储 计算 重合率要高

### 二、准备数据

用户行为数据（通过产品埋点获得）

节目信息数据（原始数据在MySQL）

用户信息数据（原始数据在MySQL）

### 三、安装Sqoop

节目信息数据、用户信息数据存储在mysql数据库，为了避免直接连接操作业务数据库，要通过Sqoop工具将数据将数据同步到集群中，方便后续对数据的多维度分析

Sqoop安装步骤：

1. 下载sqoop安装包

<https://mirrors.tuna.tsinghua.edu.cn/apache/sqoop/1.4.7/sqoop-1.4.7.bin__hadoop-2.6.0.tar.gz>

1. 配置环境变量

vi ~/.bash\_profile

export SQOOP\_HOME=/opt/software/sqoop/sqoop-1.4.7.bin\_\_hadoop-2.6.0

export PATH=$PATH:$SQOOP\_HOME/bin

source .bash\_profile

1. 进入sqoop的conf目录

修改文件名：sqoop-env-template.sh->sqoop-env.sh

1. 配置sqoop-env.sh

|  |
| --- |
| export HADOOP\_PREFIX=/opt/software/hadoop/hadoop-2.6.5  export HADOOP\_HOME=${HADOOP\_PREFIX}  export HADOOP\_COMMON\_HOME=${HADOOP\_PREFIX}  export HADOOP\_HDFS\_HOME=${HADOOP\_PREFIX}  export HADOOP\_YARN\_HOME=${HADOOP\_PREFIX}  export HADOOP\_MAPRED\_HOME=${HADOOP\_PREFIX}  export HIVE\_HOME=/opt/software/hive/hive-2.3.3 |

1. 拷贝jar

将hive安装包中的hive-common-2.3.3.jar拷贝到sqoop的lib目录下

将mysql的驱动jar（mysql-connector-java-5.1.34.jar）拷贝到sqoop的lib目录下

### 数据迁移

节目信息数据（192.168.150.111 mysql）迁移至Hive

用户信息数据（192.168.150.111 mysql）迁移至Hive

因为数据的特点，用户数据及节目数据会有新增及修改问题，导入数据分为全量导入和增量导入

1. 全量导入
2. 用户数据从mysql中导入到HDFS上 node01

|  |
| --- |
| sqoop import --connect 'jdbc:mysql://192.168.150.111/program\_db' \  --username root \  --password 123123 \  --table user\_info \  --columns id,sn,create\_date,province,city \  --warehouse-dir '/data' \  --fields-terminated-by ',' |

基于HDFS上的数据，创建Hive外表与HDFS关联数据 node01

|  |
| --- |
| create external table user\_info(  id INT,  sn STRING comment "article\_id",  create\_date STRING comment "item create time",  province STRING comment "province",  city STRING comment "city"  )  COMMENT "user info"  row format delimited fields terminated by ','  LOCATION '/data/user\_info'; |

1. 节目信息数据从mysql中导入到HDFS上

|  |
| --- |
| sqoop import --connect 'jdbc:mysql://192.168.150.111/program\_db' \  --username root \  --password 123123 \  --table item\_info \  --columns id,create\_date,air\_date,title,name,des,keywords,focus,length,content\_model,area,language,quality,is\_3d \  --warehouse-dir '/data' \ 这里是讲table表 item\_info 创建了一个文件目录，实际数据在这里面  --fields-terminated-by "\001" \  --lines-terminated-by "\n" |

基于HDFS上的数据，创建Hive外表与HDFS关联数据

|  |
| --- |
| create external table item\_info(  id BIGINT comment "item unique key",  create\_date String comment "create date",  air\_date String comment "air\_date",  title String comment "title",  name String comment "name",  desc String comment "desc",  keywords String comment "keywords",  focus String comment "focus",  length BIGINT comment "length",  content\_model String comment "content\_model",  area String comment "area",  language String comment "language",  quality String comment "quality",  is\_3d String comment "is\_3d"  )  COMMENT "item\_info"  row format delimited fields terminated by '\001'  lines-terminated-by "\n"  LOCATION '/data/item\_info'; |

2、增量导入

增量导入用户数据（新增或者修改用户数据，last\_mod最后更新时间都会更新）

|  |
| --- |
| insert into user\_info values(3781945,'11111111111','2018-11-30 09:18:21','山东','青岛',null);  update user\_info set province='中国' where id = 3781944; |
| 通过更新时间来增量导入数据  sqoop import --connect jdbc:mysql://192.168.150.111/program\_db \  --username root \  --password 123123 \  --table user\_info \  --columns id,sn,create\_date,province,city \  --target-dir '/data/user\_info' \ 这里注意，这个dir和上面的warehouse-dir有区别，这里是带了表明的，才能找到数据  --incremental lastmodified \  --check-column last\_mod \  --last-value '2020-03-08 20:10:00' \  --fields-terminated-by ',' \  --merge-key id |

增量导入节目数据（新增或者修改节目数据，last\_mod最后更新时间都会更新）

|  |
| --- |
| update item\_info set title='一条狗的使命2' where id = 157781; |
| sqoop import --connect jdbc:mysql://192.168.150.111/program\_db \  --username root \  --password 123123 \  --table item\_info \  --columns id,create\_date,air\_date,title,name,des,keywords,focus,length,content\_model,area,language,quality,is\_3d \  --target-dir '/data/item\_info' \  --incremental lastmodified \ 这个代表是根据最新的更新时间来进行导入  --check-column last\_mod \ 下面的merge-key策略会更新最后修改时间字段，然后在拉取的时候，就可以根据时间来获取数据了  --last-value '2020-03-05 23:10:00' \  --fields-terminated-by '\001' \  --merge-key id merge-key这里注意，在两条相同记录时，是基于原纪录合并的 |

### 设置定时任务同步数据

安装crontab：

yum install crontabs

启动服务：

service crond start

关闭服务：

service crond start

重启服务：

service crond restart

重新载入配置：

service crond reload

查看crontab服务状态：

service crond status

编辑定时任务：

crontab –e

\*/5 \* \* \* \* sh /root/import\_incremental.sh >> /root/log.log

删除定时任务：

crontab -r

查看定时任务列表：

crontab -l



|  |
| --- |
| #!/bin/bash  time=`date +"%Y-%m-%d %k:%M:%S" -d "-6 minutes"`  echo $time  #增量同步用户数据到HDFS中  sqoop import --connect jdbc:mysql://192.168.150.111/program\_db \  --username root \  --password 123123 \  --table user\_info \  --columns id,sn,create\_date,province,city \  --target-dir '/data/user\_info' \  --incremental lastmodified \  --check-column last\_mod \  --last-value "${time}" \  --fields-terminated-by ',' \  --merge-key id  #增量同步节目信息数据  sqoop import --connect jdbc:mysql://192.168.150.111/program\_db \  --username root \  --password 123123 \  --table item\_info \  --columns id,create\_date,air\_date,title,name,des,keywords,focus,length,content\_model,area,language,quality,is\_3d \  --target-dir '/data/item\_info' \  --incremental lastmodified \  --check-column last\_mod \  --last-value "${time}" \  --fields-terminated-by ',' \  --merge-key id |

1. **实时采集用户行为数据FLUME**

用户行为数据是通过前端产品的埋点采集的，埋点：跟踪用户对产品的使用情况，通过对埋点采集来的数据做数据分析，不断的优化产品

埋点的设置要考虑多维度产品，不同产品对埋点要求不一致，比如：推荐系统还需要在推荐列表中设置埋点，以评估推荐模型的状态

（1）收集用户行为数据

采用flume将业务服务器上的数据采集到大数据平台

flume

|  |
| --- |
| a1.sources = r1  a1.sinks = k1  a1.channels = c1  # Describe/configure the source  a1.sources.r1.type = exec  #监控日志文件  a1.sources.r1.command = tail -F /root/test.log  a1.sources.r1.channels = c1  # Describe the sink  a1.sinks.k1.type = hdfs  a1.sinks.k1.channel = c1  #采集日志数据存放在HDFS上  a1.sinks.k1.hdfs.path = /flume/user\_action/dt=%Y%m%d/  #设置HDFS文件名称的前缀  a1.sinks.k1.hdfs.filePrefix = events-  #时间四舍五入  a1.sinks.k1.hdfs.round = true  #时间上进行“舍弃”的值  a1.sinks.k1.hdfs.roundValue = 10  #时间上进行“舍弃”的值的单位 18：00 10  a1.sinks.k1.hdfs.roundUnit = minute  #基于时间间隔来进行文件滚动，默认是30，即每隔30秒滚动一个文件。0就是不使用这个策略  a1.sinks.k1.hdfs.rollInterval = 0  #基于文件大小进行文件滚动，默认是1024，即当文件大于1024个字节时，关闭当前文件，创建新的文#件。0就是不使用这个策略  a1.sinks.k1.hdfs.rollSize = 50000  #基于event数量进行文件滚动。默认是10，即event个数达到10时进行文件滚动。0就是不使用这个#策略  a1.sinks.k1.hdfs.rollCount = 10  #每次从channel中拿10条数据写入到hdfs的临时文件中  a1.sinks.k1.hdfs.batchSize = 10  a1.sinks.k1.hdfs.useLocalTimeStamp = true  #生成的文件类型，默认是Sequencefile，可用DataStream，则为普通文本  a1.sinks.k1.hdfs.fileType = DataStream  # Use a channel which buffers events in memory  a1.channels.c1.type = memory  #channle 容纳的最大event条数  a1.channels.c1.capacity = 1000  #从source传递给sink的每个事务（每个处理过程）最大event条数  a1.channels.c1.transactionCapacity = 100  # Bind the source and sink to the channel  a1.sources.r1.channels = c1  a1.sinks.k1.channel = c1 |

启动Flume：

|  |
| --- |
| bin/flume-ng agent -c conf -f conf/tail-hdfs.conf -n a1 |

1. 创建Hive外表

node01 hive

|  |
| --- |
| create external table user\_action(  sn STRING comment "sn",  item\_id INT comment "item\_id",  duration BIGINT comment "duration",  time String comment "time",  position BIGINT comment "position"  )  COMMENT "user\_action"  PARTITIONED BY(dt STRING)  row format delimited fields terminated by ','  LOCATION '/flume/user\_action'; |
| 加载分区数据到hive表（不加载是看不到数据的）刷新元数据  load data inpath '/flume/user\_action/dt=20200307' OVERWRITE INTO TABLE user\_action PARTITION (dt="20200307"); |

1. **构建节目画像**

为了更好的理解节目,需要基于节目数据构建节目画像，每个节目的关键词是节目画像中非常重要的组成部分，本章节结合TF-IDF和TextRank算法提取关键词部分

1. 通过TF-IDF提取文章关键词

TF-IDF（term frequency–inverse document frequency，词频-逆向文件频率）是一种用于信息检索（information retrieval）与文本挖掘（text mining）的常用加权技术。  
TF-IDF是一种统计方法，用以评估一字词对于一个文件集或一个语料库中的其中一份文件的重要程度。字词的重要性随着它在文件中出现的次数成正比增加，但同时会随着它在语料库中出现的频率成反比下降

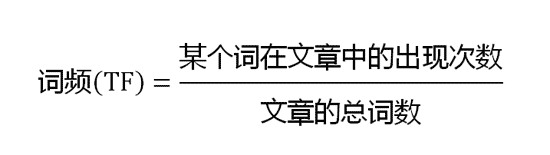
TF-IDF的主要思想是：如果某个单词在一篇文章中出现的频率TF高，并且在其他文章中很少出现，则认为此词或者短语具有很好的类别区分能力，适合用来分类

* TF是词频(Term Frequency)：词频（TF）表示词条（关键字）在文本中出现的频率
* 逆向文件频率 (IDF) ：某一特定词语的IDF，可以由总文件数目除以包含该词语的文件数目，再将得到的商取对数得到

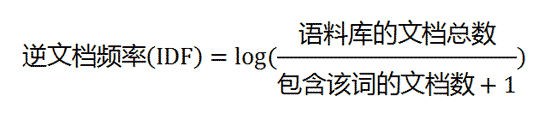
总之：TF-IDF实际上是：TF \*IDF。某一特定文件内的高词语频率，以及该词语在整个文件集合中的低文件频率，可以产生出高权重的TF-IDF。因此，TF-IDF倾向于过滤掉常见的词语，保留重要的词语

TF-IDF计算步骤：

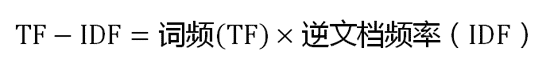
1. 计算词频TF



1. 计算逆文档频率IDF



1. 计算TF-IDF



实现代码：

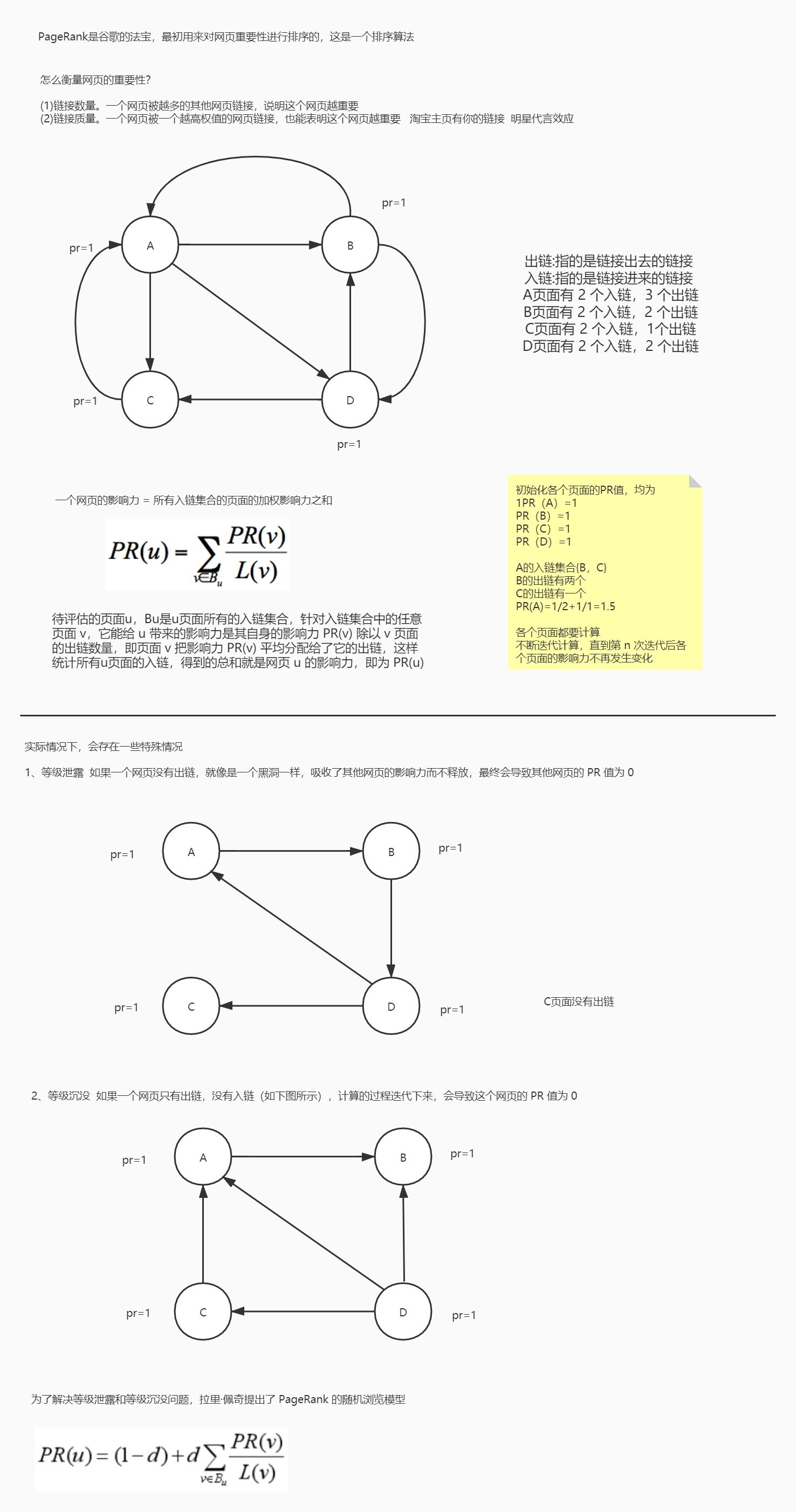
|  |
| --- |
| */\*\*  \* 对节目信息分词，然后求TF-IDF值  \*/* object ComputeTFIDF {  def main(args: Array[String]): Unit = {  //通过SparkSessionBase创建Spark会话  val session = SparkSessionBase.*createSparkSession*()  import session.implicits.\_  session.sql("use program")  //获取节目信息，然后对其进行分词  // val articleDF = session.sql("select \* from item\_info limit 20")  val articleDF = session.table("item\_info")  val seg = new SegmentWordUtil()  val words\_df = articleDF.*rdd*.mapPartitions(seg.segeFun).toDF("item\_id", "words")  // words\_df.show(false)   //创建CountVectorizer对象，统计所有影响的词，形成词袋  val countVectorizer = new CountVectorizer()  countVectorizer.setInputCol("words")  countVectorizer.setOutputCol("features")  countVectorizer.setVocabSize(10000)  countVectorizer.setMinDF(1.0)   //训练词袋模型  var cvModel = countVectorizer.fit(words\_df)  // //保存词袋模型到hdfs上  // cvModel.write.overwrite().save("hdfs://node01:9000/recommond\_program/models/CV.model")  // //通过spark sql读取模型内容  // session.read.parquet("hdfs://node01:9000/recommond\_program/models/CV.model/data/\*").show()  // //这是所有的词  // cvModel.vocabulary.foreach(println)  */\*\*  \* 使用词袋模型 统计每个节目信息的词频  \* +----------+--------------------+--------------------+  \* |article\_id| words| features\_count|  \* +----------+--------------------+--------------------+  \* | 157781|[miguel, bose, co...|(895,[282,302,345...| 向量长度是895，282,302,345等位置是1，其他为0 一个比较稀疏的向量  \* | 157906|[麦当娜, 摇滚乐, 注, 入了,...|(895,[10,38,170,2...|  \*/* val cv\_result = cvModel.transform(words\_df)  //创建IDF对象  val idf = new IDF()  idf.setInputCol("features")  idf.setOutputCol("features\_tfidf")  //计算每个词的逆文档频率  val idfModel = idf.fit(cv\_result)  idfModel.write.overwrite().save("hdfs://node01:9000/recommond\_program/models/IDF.model")   session.read.parquet("hdfs://node01:9000/recommond\_program/models/IDF.model/data").show()   */\*\*  \* 将每个单词对应的IDF（逆文档频率） 保存在Hive表中  \*/* //整理数据格式（index,word,IDF）  val keywordsWithIDFList = new ListBuffer[(Int, String, Double)]  val words = cvModel.vocabulary  val idfs = idfModel.idf.toArray  for (index <- 0 until (words.length)) {  keywordsWithIDFList += ((index, words(index), idfs(index)))  }  //保存数据  session.sql("use tmp\_program")  session  .sparkContext  .parallelize(keywordsWithIDFList)  .toDF("index", "keywords", "idf")  .write  .mode(SaveMode.*Overwrite*)  .insertInto("keyword\_idf")    val tfIdfResult = idfModel.transform(cv\_result)  tfIdfResult.show()   //根据TFIDF来排序  val keyword2TFIDF = tfIdfResult.*rdd*.mapPartitions(partition => {  val rest = new ListBuffer[(Long, Int, Double)]  val topN = 20  while (partition.hasNext) {  val row = partition.next()  var idfVals: List[Double] = row.getAs[SparseVector]("features\_tfidf").values.toList  val tmpList = new ListBuffer[(Int, Double)]  for (i <- 0 until (idfVals.length)) tmpList += ((i, idfVals(i)))  val buffer = tmpList.sortBy(\_.\_2).reverse  for (item <- buffer.take(topN))  rest += ((row.getAs[Long]("item\_id"), item.\_1, item.\_2))  }  rest.iterator  }).toDF("item\_id", "index", "tfidf")  keyword2TFIDF.show(10)   keyword2TFIDF.createGlobalTempView("keywordsByTable")  //获取索引对应的单词，组织格式 保存Hive表  session.sql("select \* from keyword\_idf a join global\_temp.keywordsByTable b on a.index = b.index")  .select("item\_id", "word", "tfidf")  .write  .mode(SaveMode.*Overwrite*)  .insertInto("keyword\_tfidf")  session.close()  } } |

1. 通过TextRank提取文章关键词

TF-IDF需要在语料库上计算IDF(逆文档频率)，TextRank利用一篇文档内部的词语间的共现信息(语义)便可以抽取关键词

TextRank算法的一个重要特点是可以脱离语料库的背景，仅对单篇文档进行分析就可以提取该文档的关键词。基本思想来源于Google的PageRank算法。这种算法是1997年，Google创始人拉里.佩奇和谢尔盖.布林在构建早期的搜索系统原型时提出的一种链接分析算法，基本思想有两条:

1. 链接数量。一个网页被越多的其他网页链接，说明这个网页越重要
2. 链接质量。一个网页被一个越高权值的网页链接，也能表明这个网页越重要





TextRank算法将文本中的语法单元视作图中的节点，如果两个语法单元存在一定语法关系（例如共现），则这两个语法单元在图中就会有一条边相互连接，通过一定的迭代次数，最终不同的节点会有不同的权重，权重高的语法单元可以作为关键词，PageRank的思想来解释它：

1. 如果一个单词出现在很多单词后面的话，那么说明这个单词比较重要

（2）一个TextRank值很高的单词后面跟着的一个单词，那么这个单词的TextRank值会相应地因此而提高

TextRank实现步骤:

1. 对每个句子进行分词
2. 过滤掉除指定词性外的其他单词，过滤掉出现在停用词表的单词
3. 将剩下的单词中循环选择一个单词，将其与其后面n个单词分别组合成n条边

举一个例子：

薛佳凝新恋情疑似曝光，那个让胡歌落泪的前任开始了新感情

1. 分词

[薛佳凝，新恋情，疑似，曝光，那个，让，胡歌，落泪，的，前任，开始了，新感情]

1. 过滤掉停用词

[薛佳凝，新恋情，疑似，曝光，胡歌，落泪，前任，开始了，新感情]

1. 假设n=3

薛佳凝 -> 新恋情，疑似，曝光

新恋情 -> 疑似，曝光，胡歌

疑似 -> 曝光，胡歌，落泪

曝光 -> 胡歌，落泪，前任

胡歌 -> 落泪，前任，开始了

落泪 -> 前任，开始了，新感情

前任 -> 开始了，新感情

开始了 -> 新感情

可以形成一个比较复杂的Graph图

基于Graph图依据PageRank的计算方式计算

|  |
| --- |
| class TextRank extends Serializable {  var numKeyword: Int = 10  /\* 关键词个数 \*/  var d: Double = 0.85f  /\* 阻尼系数 \*/  var max\_iter: Int = 200  /\* 最大迭代次数 \*/  var min\_diff: Double = 0.001f  /\* 最小变化区间 \*/  private final var index: Int = 0   def transform(document: Iterable[\_]): mutable.HashMap[String, mutable.HashSet[String]] = {  val keyword = mutable.HashMap.empty[String, mutable.HashSet[String]]  val que = mutable.Queue.empty[String]  document.foreach { term =>  val word = term.toString  if (!keyword.contains(word)) {  /\* 初始化，对每个分词分配一个 HashSet 空间\*/  keyword.put(word, mutable.HashSet.empty[String])  }  que.enqueue(word)  if (que.size > 5) {  que.dequeue()  }   for (w1 <- que) {  for (w2 <- que) {  if (!w1.equals(w2)) {  keyword.apply(w1).add(w2)  keyword.apply(w2).add(w1)  }  }  }  }  keyword  }   def rank(document: mutable.HashMap[String, mutable.HashSet[String]]): mutable.HashMap[String, Double] = {  var score = mutable.HashMap.empty[String, Double]  breakable ({  for (iter <- 1 to max\_iter) {  val tmpScore = mutable.HashMap.empty[String, Double]  var max\_diff: Double = 0f  for (word <- document) {  tmpScore.put(word.\_1, 1 - d)  for (element <- word.\_2) {  val size = document.apply(element).size  if (0 == size) println("document.apply(element).size == 0 :element: " + element + "keyword: " + word.\_1)  if (word.\_1.equals(element)) println("word.\_1.equals(element): " + element + "keyword: " + word.\_1)  if ((!word.\_1.equals(element)) && (0 != size)) {  /\* 计算，这里计算方式可以和TextRank的公式对应起来 \*/  tmpScore.put(word.\_1, tmpScore.apply(word.\_1) + ((d / size) \* score.getOrElse(word.\_1, 0.0d)))  }  }  /\* 取出每次计算中变化最大的值，用于下面得比较，如果max\_diff的变化低于min\_diff，则停止迭代 \*/  max\_diff = Math.max(max\_diff, Math.abs(tmpScore.apply(word.\_1) - score.getOrElse(word.\_1, 0.0d)))  }  score = tmpScore  if (max\_diff <= min\_diff) break()  }  })  score  }   def sortByRank(doc: mutable.HashMap[String, Double]) = {  val mapDoc = doc.toSeq  val reverse = mapDoc.sortBy(-\_.\_2).take(10).toMap  reverse  } } |

|  |
| --- |
| object ComputeTextRank {    def main(args: Array[String]): Unit = {  //通过SparkSessionBase创建Spark会话  val session = SparkSessionBase.createSparkSession()  session.sql("use program")  //获取节目信息，然后对其进行分词 // val articleDF = session.sql("select \* from item\_info limit 20")  val articleDF = session.table("item\_info")  val seg = new SegmentWordUtil()  val wordsRDD = articleDF.rdd.mapPartitions(seg.segeFun)   //计算每个节目 每个单词的TR值  val tralgm = new TextRank()  val transformGraphRDD = wordsRDD.map(x => (x.\_1, tralgm.transform(x.\_2)))  val rankRDD = transformGraphRDD.map(x => (x.\_1, tralgm.rank(x.\_2)))   /\*\*  \* 将每个节目 每个单词的TR值与对应的单词的IDF相乘  \* （1）创建广播变量 将idf\_keywords\_values（word idf）表数据 作为广播变量  \* （2）遍历sortByRankRDD 匹配单词，TR\*IDF  \*/  val word2IDFMap = mutable.Map[String, Double]()  session.table("keyword\_idf").rdd.collect().foreach(row => {  word2IDFMap += ((row.getAs[String]("word"), row.getAs[Double]("idf")))  })  val word2IDFBroad = session.sparkContext.broadcast(word2IDFMap)   //将每篇文章中每个单词的tr\*对用的IDF值 作为筛选关键词的依据  val keyWordsWithWeightsRDD = rankRDD.map(data => {  val itemID = data.\_1  val word2TR = data.\_2  val word2IDFMap = word2IDFBroad.value  val list = new ListBuffer[(Long, String, Double)]  val word2Weights = word2TR.map(t => {  val word = t.\_1  val tr = t.\_2  var weights = 0d  if (word2IDFMap.contains(word)) {  weights = word2IDFMap(word) \* tr  } else {  weights = tr  }  (word, weights)  })  (itemID, word2Weights)  })   //根据混合的weight值排序，选择topK个单词  val sortByWeightRDD = keyWordsWithWeightsRDD  .filter(\_.\_2.size > 10)  .map(x => (x.\_1, sortByWeights(x.\_2)))  .flatMap(explode)  //keyWordsWithWeightsRDD转成DF  import session.implicits.\_  val word2WeightsDF = sortByWeightRDD.toDF("item\_id", "word", "weight")  session.sql("use tmp\_program")  word2WeightsDF.write.mode(SaveMode.Overwrite).insertInto("keyword\_tr")  session.close()  }   def explode(data: (Long, Map[String, Double])) = {  val itemID = data.\_1  val ds = data.\_2  val list = new ListBuffer[(Long, String, Double)]  for (elem <- ds) {  list += ((itemID, elem.\_1, elem.\_2))  }  list.iterator  }   def sortByWeights(doc: mutable.HashMap[String, Double]) = {  val mapDoc = doc.toSeq  val reverse = mapDoc.sortBy(-\_.\_2).take(10).toMap  reverse  } } |

1. 合并关键词

只将未来做训练集时用到的字段（节目关键词）构建到节目画像中，其他无关字段被忽略

1. 把通过TF-IDF指标提取出来的关键词与通过TextRank提取出来的关键词进行合并
2. 合并结果保存在Hive数据库 （tmp\_program.item\_keyword）

代码实现：

|  |
| --- |
| val session = SparkSessionBase.createSparkSession()  import session.implicits.\_  session.sql("use tmp\_program")  /\*\*  \* +-------+--------------------+  \* |item\_id| keyword|  \* +-------+--------------------+  \* | 159131| [音乐, 性感]|  \* | 158531| [乐队, 歌, 最]|  \* | 159356| [中, 李]|  \* | 158306| [乐队, 最, 演唱]|  \* 合并 作为关键词  \*/  val sqlText = "" +  "SELECT w.item\_id, collect\_set(w.word) AS keyword1, collect\_set(k.word) AS keyword2 " +  "FROM keyword\_tr w " +  " JOIN keyword\_tfidf k ON (w.item\_id = k.item\_id) " +  "GROUP BY w.item\_id"  val mergeDF = session.sql(sqlText)  mergeDF.rdd.map(row => {  val itemID = row.getAs[Int]("item\_id")  val keyword1 = row.getAs[Seq[String]]("keyword1")  val keyword2 = row.getAs[Seq[String]]("keyword2")  val keywords = keyword1.union(keyword2).distinct.toArray  (itemID, keywords)  }).toDF("item\_id", "keyword")  .write  .mode(SaveMode.Overwrite)  .insertInto("item\_keyword") } |

1. 构建节目画像
2. 节目的关键词表与节目信息表关联
3. 关联结果保存至Hive数据库（tmp\_program.item\_profile）

代码实现：

|  |
| --- |
| val session = SparkSessionBase.createSparkSession() session.sql("use tmp\_program") val sqlText = "SELECT b.id, a.keyword, b.create\_date, b.air\_date, b.length " +  ", b.content\_model, b.area, b.language, b.quality, b.is\_3d " +  "FROM tmp\_program.item\_keyword a " +  "JOIN program.item\_info b ON a.item\_id = b.id "; val restDF = session.sql(sqlText) restDF  .write  .mode(SaveMode.Overwrite)  .saveAsTable("item\_profile") session.close() |

1. 计算Word2Vec 词向量
2. Word2Vec算法思想

* 词向量：将文字通过一串数字向量表示
* One-hot采用稀疏方式来表达，但是简单易操作

薯条[0，0，0，0，0，0，1，0，0，0]

薯片[0，0，1，0，0，0，0，0，0，0]

汽车[1，0，0，0，0，0，0，0，0，0]

One-hot编码方式仅仅是将词符号化，会存在维度过大，过于稀疏并且通过向量无法看出两个词之间 的距离（相似性）

* Word2Vec的思想来源是统计语言模型

统计语言模型： 统计语言模型把语言（词的组合）看作一个随机事件，并赋予相应的概率来描述其 的合理性

统计语言模型是为一个长度为 m 的字符串确定一个概率分布 P(w1; w2; ...; wm)，表示其存在的可 能性，其中 w1 到 m 依次表示这段文本中的各个词

* 常用统计语言模型N-gram Model

在中文表达中，每一个词的出现是受前面的词影响的

今天 天气 真好 真好出现的概率是受天气影响的

今天 阳光 明媚 明媚出现的概率是受阳光影响的

今天 偶遇 一个 老同学 一个人 一个东西 一个物体 一个有趣的事情....

所以，当前词的出现可能与之前的N个词有关系

举例：有一个由n个词组成的句子，如何衡量它的概率呢？假设，每一个单词都要依赖于从第一个单词到的影响：



|  |
| --- |
| Tips：通过预料库中所有文本的分词结果统计出相应概率  1）w1 w2 w4 w5  2）w1 w3 w5 w6  3）w1 w2 w3 w5  4）w1 w2 w3 w4  统计第4句话的概率    1 \* 3/4 \* 2/3 \* 1/2 = 1/4  这句话的合理性概率是1/4  如果第4句话语言方式非常古怪，非同寻常，那么基于预料中的数据计算出来的合理性概率会很低 |

这样来衡量概率，存在的问题：

1. 参数空间过大
2. 数据稀疏严重，有些组合出现的情况可能没有，组合阶数越高越明显

为了解决N-gram模型存在的问题，基于这样一种假设，当前词的出现只与前面第N-1个词相关，而与其它任何词都不相关

1. Gram(N=2，Bi-Gram)：



1. Gram(N=3，Tri-Gram)：



随着 n 的提升，我们拥有了更多的前置信息量，可以更加准确地预测下一个词

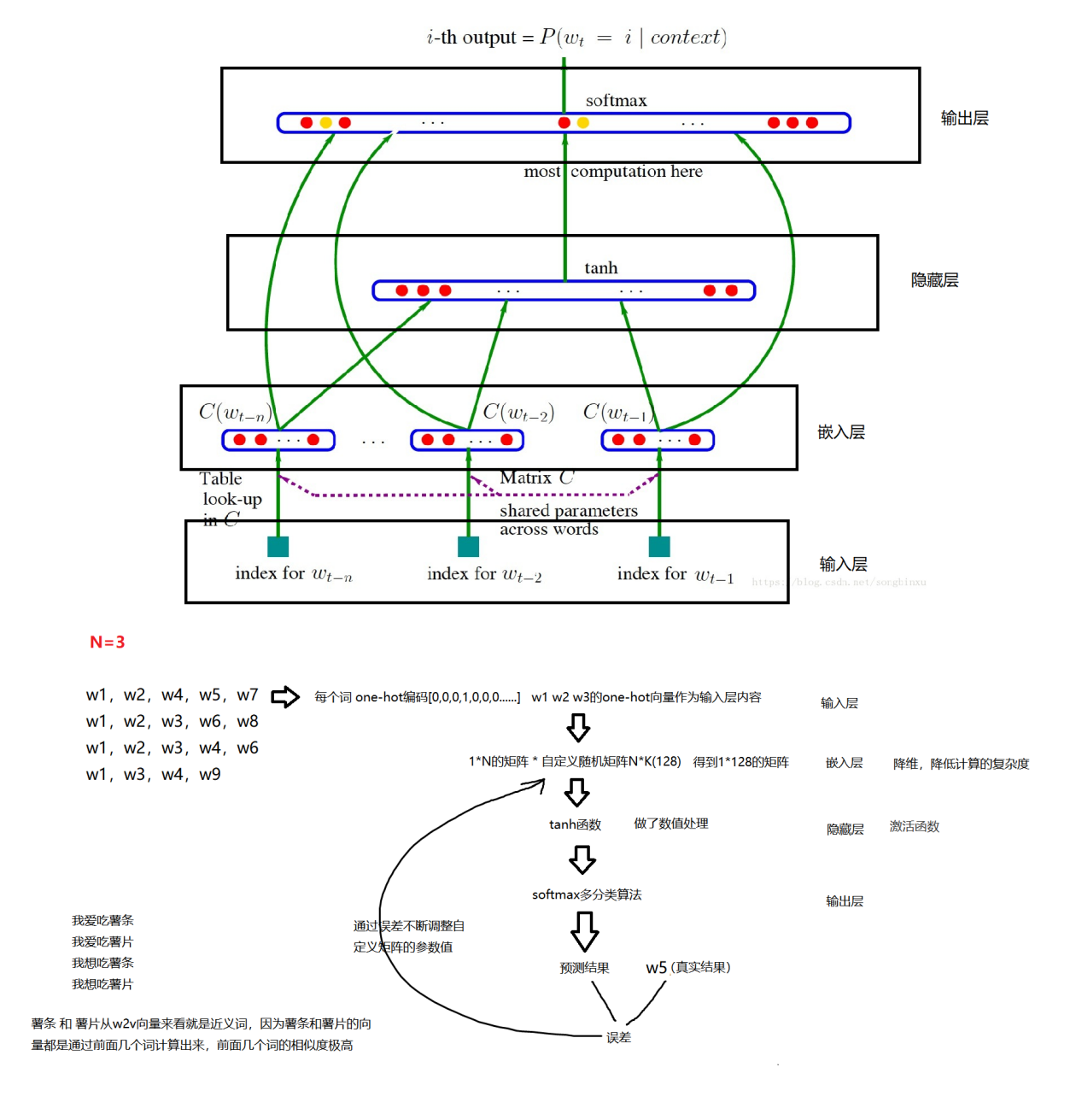
Bi-Gram，Tri-Gram会存在信息损失的问题，但是大大降低了计算的复杂度，tradeoff

Bi-Gram，Tri-Gram依然可能存在维度爆炸问题

* 统计语言模型总结：

统计语言模型就是计算一个句子的概率值大小，整句的概率就是表明该句子越合理。N-gram是典型 的统计语言模型，它做出了一种假设，当前词的出现只与前面N-1个词相关，而与其它任何词都不 相关，整句的概率就是各个词出现概率的乘积。它其中存在很多问题，再求每一个词出现的概率时， 随着N的提升，能够拥有更多的前置信息量，可以使得当前词的预测更加准确，但是当N过大时会 出现稀疏问题，导致很多词的概率值为0，为解决这一问题，因此常用的为bigram 或 trigram，这 就导致N-gram无法获得上文的长时依赖。另一方面N-gram 只是基于频次进行统计，没有足够的 泛化能力

* 2003年 Bengio 提出，神经网络语言模型( neural network language model, NNLM)的思想是提出词向量的概念，代替 N-gram使用离散变量(高维)，采用连续变量(具有一定维度的实数向量)来进行单词的分布式表示，解决了维度爆炸的问题，同时通过词向量可获取词之间的相似性
* 所建立的语言模型的任务是根据窗口大小内的上文来预测下一个词，因此从另一个角度看它就是一个使用神经网络编码的n-gram模型



是一个最简单的神经网络，仅由四层构成，输入层、嵌入层、隐藏层、输出层

（2）计算节目的Word2Vec 词向量

#### **SparkSession加载Hive中的节目信息表（item\_info）**

#### **对节目信息表中的desc、title、name这三个字段进行合并**

#### **将合并完成的信息做分词处理**

#### **基于分词结果计算Word2Vec词向量**

#### **Word2Vec词向量结果保存到HDFS分布式存储集群中**

**代码实现：**

|  |
| --- |
| //通过SparkSessionBase创建Spark会话 val session = SparkSessionBase.createSparkSession() import session.implicits.\_ session.sql("use program") //获取节目信息，然后对其进行分词 // val articleDF = session.sql("select \* from item\_info limit 20") val articleDF = session.table("item\_info") val seg = new SegmentWordUtil() val words\_df = articleDF.rdd.mapPartitions(seg.segeFun).toDF("item\_id", "words")  val w2v = new Word2Vec w2v.setInputCol("words") w2v.setOutputCol("model") w2v.setVectorSize(1000) w2v.setMinCount(3)  val w2vModel = w2v.fit(words\_df) w2vModel.write.overwrite().save("hdfs://node01:9000/recommond\_program/models/w2v.model") session.read.parquet("hdfs://node01:9000/recommond\_program/models/w2v.model/data/\*").show() session.close() |

1. **基于节目画像计算节目之间的相似度**

在实时召回策略中，根据用户浏览内容实时召回与当前节目相似的节目，相似的节目是在离线部分计算完成，相似度计算一般都是T+1，用户在实时浏览节目的时候，直接去查询相似节目，这一步是为后续召回策略做准备

相似度计算的方式有N多种：

1、欧氏距离

2、余弦相似度

3、Jaccard相似度

Jaccard系数是常见的衡量两个向量（或集合）相似度的度量



们令A和B的交集的元素数量设为 ，A和B的非交集元素数量设为 ，则Jaccard相似度即为

多种不同的相似度计算方式，都用遍历每一对数据计算。但当数据维度或者数据量增大到一定程度时，计算复杂度直线飙升了

1）两个用户之间相似度的计算随着维度的增加而增加

2）计算每一个用户和其他所有用户之间的相似度的复杂度随着用户规模的增长，呈平方增长

LSH+Minhash算法可以解决相似度计算的复杂性

#### Minhash降维算法

原始问题的关键在于计算复杂度太大。所以，如果能够找到一种很好的方法将原始集合压缩成更小的集合，而且又不失去相似性，那么可以缩短计算时间

* 三篇文章：

01：我爱你中国

02：爱你我的中国

03：死鬼，我好讨厌你哦

* 对三篇文章向量化

|  |  |  |  |
| --- | --- | --- | --- |
|  | 01 | 02 | 03 |
| 我爱你 | 1 | 0 | 0 |
| 爱你 | 1 | 1 | 0 |
| 中国 | 1 | 1 | 0 |
| 我的 | 0 | 1 | 0 |
| 死鬼 | 0 | 0 | 1 |
| 好讨厌 | 0 | 0 | 1 |
| 讨厌 | 0 | 0 | 1 |

* 如果每篇文章很长,语料库很大，每篇文章对应的向量就会非常的稀疏
* Minhash算法的降维思想：将行的顺序打乱，找到每篇文章中第一个不为0的行号，组成向量

|  |  |  |  |
| --- | --- | --- | --- |
|  | 01 | 02 | 03 |
| 爱你 | 1 | 1 | 0 |
| 中国 | 1 | 1 | 0 |
| 死鬼 | 0 | 0 | 1 |
| 讨厌 | 0 | 0 | 1 |
| 好讨厌 | 0 | 0 | 1 |
| 我爱你 | 1 | 0 | 0 |
| 我的 | 0 | 1 | 0 |

第一次打乱顺序

|  |  |  |
| --- | --- | --- |
| 01 | 02 | 03 |
| 1 | 1 | 3 |

打乱T次顺序

|  |  |  |
| --- | --- | --- |
| 01 | 02 | 03 |
| 3 | 3 | 3 |
| 3 | 3 | 5 |
| 4 | 4 | 2 |
| 5 | 5 | 3 |
| 6 | 5 | 2 |
| 4 | 4 | 4 |
| ... | ... | ... |

如果01和02文本相似度很高，打乱顺序后再去寻找不为0的行号，行号相似度也极高

于是乎：经过这种方式可以将原本N（词袋大小）\*M（语料库大小）的矩阵转换成T（随机次数）\*M（语料库大小）

一般来说T远远小于N（20W）

#### **局部敏感哈希LSH(Locality Sensitive Hashing) 计算相似度算法**

LSH算法基于一个假设，如果两个文本本身是相似的，那么经过哈希函数转换以后的它们也具有很高的相似度

之前我们在使用MD5、SHA1、CRC32哈希函数的时候，希望尽量避免hash碰撞，但是LSH偏向于依赖碰撞，希望相似的文本发生碰撞的概率要尽可能高一些。不相似的文章，发生碰撞的概率尽可能低一些

|  |  |  |
| --- | --- | --- |
| 01 | 02 | 03 |
| 3 | 3 | 3 |
| 3 | 3 | 5 |
| 4 | 4 | 2 |
| 5 | 5 | 3 |
| 6 | 5 | 2 |
| 4 | 4 | 4 |

算法原理:

将矩阵拆分成n段，第一段称之为brand1，第二段称之为brand2，将每段的向量通过Hash函数（md5、sha1）计算，Hash值就是对应的Hash桶下标

为什么要分成n段？

提高文本被分到同一个桶的概率

如果整体来看01，02并不是完全相同，直接使用Hash函数对整体计算Hash值，那么Hash值很大概率上不同，很大概率分到不同的桶中

如果分段来看，brand1中的01和02会被分到同一个桶中，但是brand2中的01和02不会分到同一个桶中

每一个向量在图中被分为了b段（每一列为一个向量），每一段有r行。在任意一个band中分到了同一个桶内，就成为候选相似用户

假设两个文本的相似度为，在一个brand中两个文本向量至少有一个值不相同的概率为，则所有brand都不同的概率为，至少有一个brand相同的概率为

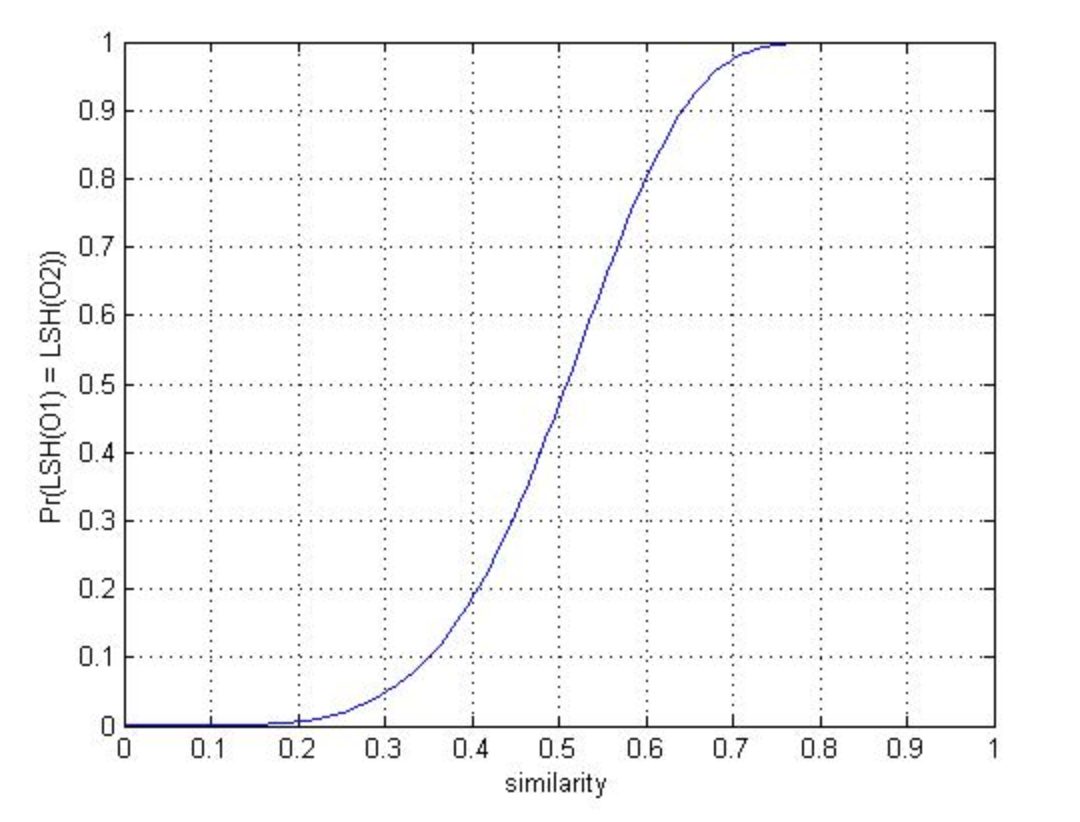
s∈[0,1]是这两个文档的相似度

当s=0.8时，两个文档被映射到同一个哈希桶的概率是

Pr(LSH(O1)=LSH(O2))=1−(1−0.85)5=0.9996439421094793

当s=0.2时，两个文档被映射到同一个哈希桶的概率是

Pr(LSH(O1)=LSH(O2))=1−(1−0.25)5=0.0063805813047682



* 相似度高于某个值的时候，到同一个桶概率会变得非常大，无限逼近于1
* 相似度低于某个值的时候，概率会变得非常小，无限逼近于0

#### 计算文本之间的相似度

* LSH算法
* 基于KMeans聚类算法

基于LSH计算相似度

在计算相似度之前，需要把每篇文章进行向量化，怎么向量化呢？

TF-IDF值？

TF-IDF只是考虑了词频和逆文档词频，并没有考虑每篇文章的上下文，就是词的组合搭配

词向量：当初在计算每个单词向量的时候，是结合上下文计算的，如果这两个单词在语义上比较相似，这两个单词的词向量在空间中的距离就比较近

结合TF-IDF值与单词的词向量这两个指标，会更加符合实际场景

所以这里我们将每篇文章中每个单词的TF-IDF值与单词的词向量相乘求和取均值

中国 tf-idf值：1.0

中国 词向量：（1，2，3）

sum（1.0 \* （1，2，3））/向量长度=value

这种直接相乘求和取均值的方式虽然会有折损，但是在计算复杂度上来说，效率提升了很多

1. 加载Hive数据库中的keyword\_tfidf表，整理成(itemID + "\_" + word, tf-idf)格式，放入到广播变量中
2. 加载Word2Vec模型，整理成（word，vector）格式，放入到广播变量中
3. 加载Hive数据库中的keyword\_idf表，整理成(word, index)格式，放入到广播变量中
4. 遍历Hive数据库中的item\_keyword数据
5. 将每个单词的tf-idf值与Word2Vec做数值计算