Spark推荐系列之Word2vec算法介绍、实现和应用说明

Original Thinkgamer 搜索与推荐Wiki 5/26

Scan to Follow

收录于话题
#Spark推荐实战 6 #精品小系列内容 35 #推荐相关笔记 48

Spark推荐实战系列目前已经更新：

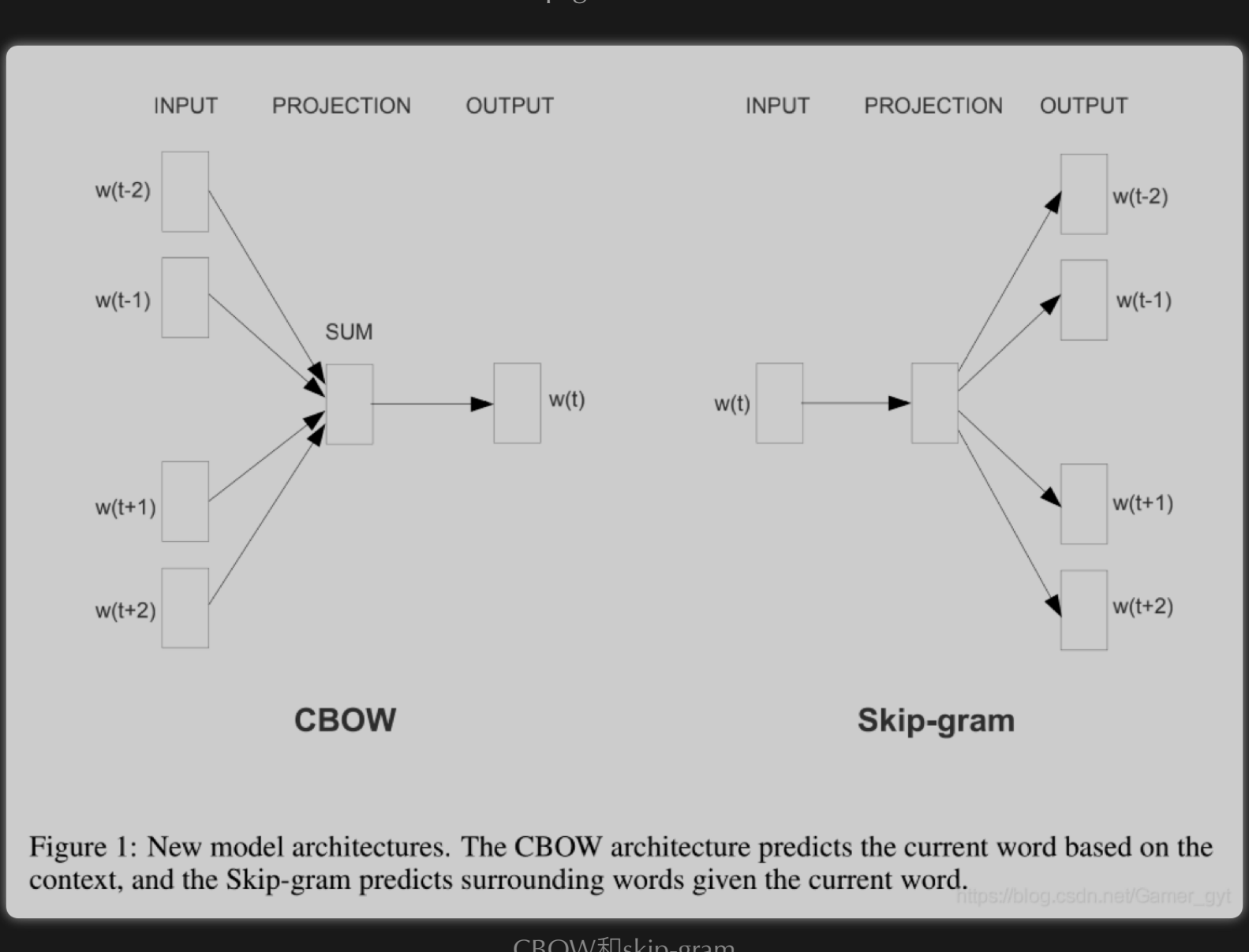
- Spark推荐实战系列之Swing算法介绍、实现与在阿里飞猪的实战应用
- Spark推荐实战系列之ALS算法实现分析
- Spark中如何使用矩阵运算间接实现i2i
- FP-Growth算法原理、Spark实现和应用介绍
- Spark推荐系列之Word2vec算法介绍、实验和应用说明

更多精彩内容，请持续关注「搜索与推荐Wiki」！

1. 背景

word2vec 是Google 2013年提出的用于计算词向量的工具，在论文Efficient Estimation of Word Representations in Vector Space中，作者提出了Word2vec计算工具，并通过对比NNLM、RNNLM语言模型验证了word2vec的有效性。

word2vec工具中包含两种模型：CBOW和skip-gram。论文中介绍的比较简单，如下图所示，CBOW是通过上下文的词预测中心词，Skip-gram则是通过输入词预测上下文的词。



CBOW和skip-gram

Word2vec 开启了Embedding的相关工作，自从embedding开始大规模走进推荐系统中，下面我们就来看一下Word2vec算法的原理、Spark实现和应用说明。

2. 算法原理

Word2vec包含了两种模型，分别是CBOW和Skip-gram，CBOW又分为：

- One-word context
- multi-word context

Cbow_One-word context

其中单词的总个数为V，隐藏层的神经元个数为N，输入层到隐藏层的权重矩阵为W_{V*N}，隐藏层到输出层的权重矩阵为W'_{N*V}。

multi-word context

此时的h 表达式为：

$$h = \frac{1}{C}W^T(x_1 + x_2 + \ldots + x_C) = \frac{1}{C}(v_{w_1} + v_{w_2} + \ldots + v_{w_C})^T$$

其中 C 表示上下文单词的个数，$w_1, w_2, \ldots, w_C$ 表示上下文单词，v_w 表示单词的输入向量（注意和输入层x区别）。

目标函数为：

$$E = -log\,p(w_O|w_{I_1}, w_{I_2}, \ldots, w_{I_C}) = -u_j * log\sum_{j'=1}^{V} exp(u'_j) = -(v'_{w_O})^T * h + log\sum_{j'=1}^{V} exp((v'_{w_j})^T$$

Skip-gram 对应的图如下：

Skip-gram

从输入层到隐藏层：

$$h = W^T_{k,\cdot} := v^T_{w_I}$$

从隐藏层到输出层：

$$p(w_{c,j} = w_{O,c}|w_I) = y_{c,j} = \frac{exp(u_{c,j})}{\sum_{j'=1}^{V} exp(u_{j'})}$$

其中：

- w_I 表示的是输入词
- $w_{c,j}$ 表示输出层第c个词实际落在了第j个神经元
- $w_{O,c}$ 表示输出层第c个词应该落在第O个神经元
- $y_{c,j}$ 表示输出层第c个词实际落在了第j个神经元上归一化后的概率
- $u_{c,j}$ 表示输出层第c个词实际落在了第j个神经元上未归一化的值

因为基于word2vec框架进行模型训练要求语料库非常大，这样才能保证结果的准确性，但随着预料库的增大，随之而来的就是计算的耗时和资源的消耗。那么有没有优化的余地呢？比如可以牺牲一定的准确性来加快训练速度，答案就是 hierarchical softmax 和 negative sampling。

在论文《Distributed Representations of Words and Phrases and their Compositionality》中介绍了训练word2vec的两个技（同样在论文《word2vec Parameter Learning Explained》中进行了详细的解释和说明），下面来具体看一下。

这里就不展开叙述了，可以参考之前的文章：论文｜万物皆可Vector之Word2vec：2个模型、2个优化及实战使用

3.Spark实现

在spark的mllib实现了对word2vec的封装，基于MLLib进行实现和应用

主函数

```
def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder().master("local[10]").appName("Word2Vec").enabl
    Logger.getRootLogger.setLevel(Level.WARN)

    val dataPath = "data/ml-100k/u.data"
    val data: RDD[Seq[String]] = spark.sparkContext.textFile(dataPath)
        .map(_.split("\t"))
        .map(l => (l(0), l(1)))
        .groupByKey()
        .map(l => l._2.toArray.toSeq)

    val word2vec = new Word2Vec()
        .setMinCount(minCount)
        .setNumPartitions(numPartitions)
        .setNumIterations(numIterations)
        .setVectorSize(vectorSize)
        .setLearningRate(learningRate)
        .setSeed(seed)

    val model = word2vec.fit(data)

    // 输出item id对应的embedding向量到
    saveItemEmbedding(spark, model)

    // 使用spark自带的防范计算item 相似度
    calItemSim(spark, model)

    // 自定义余弦相似度 计算item 相似度
    calItemSimV2(spark, model)

    spark.close()
}
```

保存item embedding

```
def saveItemEmbedding(spark: SparkSession, model: Word2VecModel) = {
    val normalizer1 = new Normalizer()

    val itemVector = spark.sparkContext.parallelize(
        model.getVectors.toArray.map(l => (l._1, normalizer1.transform(new DenseVect
    ).map(l => (l._1, l._2.toArray.mkString(",")))

    println(s"itemVector count: ${itemVector.count()}")
    itemVector.take(10).map(l => l._1 + "\t" + l._2).foreach(println)

    import spark.implicits._
    val itemVectorDF = itemVector.toDF("itemid", "vector")
        .select("itemid", "vector")
    itemVectorDF.show(10)
    // itemVectorDF.write.save("xxxx")
}
```

计算item相似度

```
def calItemSim(spark: SparkSession, model: Word2VecModel) = {
    // 这种方法离线对比 使用余弦计算item相似度 好像没有后者好
    val itemsRDD: RDD[String] = spark.sparkContext.parallelize(model.getVectors.keyS

    import spark.implicits._
    val itemSimItemDF = itemsRDD.map(l => (l, model.findSynonyms(l, 500)))
        .flatMap(l => for(one <- l._2) yield (l._1, one._1, one._2) )
        .toDF("source_item", "target_item", "sim_score")
    itemSimItemDF.show(10)
}

def calItemSimV2(spark: SparkSession, model: Word2VecModel) = {
    val normalizer1 = new Normalizer()

    val itemVector = spark.sparkContext.parallelize(
        model.getVectors.toArray.map(l => (l._1, normalizer1.transform(new DenseVect
    ).map(l => (l._1, l._2.toArray.toVector))

    import spark.implicits._
    val itemSimItem = itemVector.cartesian(itemVector)
        .map(l => (l._1._1, (l._2._1, calCos(l._1._2, l._2._2))))
        .groupByKey()
        .map(l => (l._1, l._2.toArray.sortBy(_._2).reverse.slice(0, 500)))
        .flatMap(l => for(one <- l._2) yield (l._1, one._1, one._2))
        .toDF("source_item", "target_item", "sim_score")
    itemSimItem.show(10)
}

def calCos(vector1: Vector[Double], vector2: Vector[Double]): Double = {
    //对公式部分分子进行计算
    val member = vector1.zip(vector2).map(d => d._1 * d._2).sum

    //求出分母第一个变量值
    val temp1 = math.sqrt(vector1.map(num => {
        math.pow(num, 2)
    }).sum)
    //求出分母第二个变量值
    val temp2 = math.sqrt(vector2.map(num => {
        math.pow(num, 2)
    }).sum)

    //求出分母
    val denominator = temp1 * temp2

    //进行计算
    member / denominator
}
```

4.应用

其实在产出Item Embedding之后，在召回阶段，可以进行i2i的召回，或者u2i的召回，具体使用方式如下描述：

- i2i：我们可以离线计算出Item 的相似 item列表或者实时通过es、faiss检索得到i2i，这样线上可以进行u2i & i2i的实时触发召回（实时召回一般效果都是比较好的，只要挖掘的i2i别太离谱就行）
- u2i：可以根据用户最近点击的若干个spu，来做一个avg pooling，得到用户的embedding，继而离线或者在线进行embedding的相似计算&检索，得到u2i的召回

在排序阶段，可以用item embedding的数据作为特征来使用，但是需要注意，在产出embedding之后，使用时一般进行vector的正则（normalizer），进入算法后更方便算法使用

如果是基于语义信息产出的item embedding，也可以在展示机制方面进行使用，其大概使用原理为：避免相邻的item 相似性过高（具体可以参考MMR算法）

word2vec想要达到一个好的效果前提是：系统数据比较丰富，对于数据比较稀疏的序列，word2vec学习出来的item embedding表达能力并不好。

关注我们不错过每一篇精彩

搜索与推荐Wiki
专注于搜索和推荐系统，以系列分享为主，持续打造精品内容！
209篇原创内容
Official Account

「搜索与推荐Wiki」猜你喜欢

1、论文｜LINE算法原理、代码实战和应用
2、推荐系统里的那些坑儿
3、ABTest流量分发和业界的一些做法经验
4、聊一聊海量公众号下我是如何进行筛选和内容消费的

—完—

收录于话题 #精品小系列内容·35个 >

下一篇 · FP-Growth算法原理、Spark实现和应用介绍

Read more