基于内容的商品相似度计算与商品召回

网站初期，没有大量的用户数据，因此初期的离线推荐将以基于物品画像的召回推荐为主，

主要实现逻辑：

- 根据商品关键词 对应的权重值，结合该关键词 对应的词向量，进行加权求平均，计算出该商品的向量值

  1件sku 有n个关键词，1个关键词(带有1个权重)有 1个embedding词向量表示
  该sku的商品表示 = (第i词的词向量表示*第i词的权重)加和

- 利用相似度得出 每件商品 与之相似的 topN件商品

In [2]:

```
import os
# 配置pyspark和spark driver运行时 使用的python解释器
JAVA_HOME = '/root/bigdata/jdk'
PYSPARK_PYTHON = '/miniconda2/envs/py365/bin/python'
# 当存在多个版本时，不指定很可能会导致出错
os.environ['PYSPARK_PYTHON'] = PYSPARK_PYTHON
os.environ['PYSPARK_DRIVER_PYTHON'] = PYSPARK_PYTHON
os.environ['JAVA_HOME'] = JAVA_HOME
# 配置spark信息
from pyspark import SparkConf
from pyspark.sql import SparkSession

SPARK_APP_NAME = "itemFigureRecall"
SPARK_URL = "spark://192.168.58.100:7077"

conf = SparkConf()    # 创建spark config对象
config = (
    ("spark.app.name", SPARK_APP_NAME),    # 设置启动的spark的app名称，没有提供，将随机
    ("spark.executor.memory", "2g"),    # 设置该app启动时占用的内存用量，默认1g，指一
    ("spark.master", SPARK_URL),    # spark master的地址
    ("spark.executor.cores", "2"),    # 设置spark executor使用的CPU核心数，指一台虚拟机
    ("hive.metastore.uris", "thrift://localhost:9083"),    # 配置hive元数据的访问，否则

    # 以下三项配置，可以控制执行器数量
#     ("spark.dynamicAllocation.enabled", True),
#     ("spark.dynamicAllocation.initialExecutors", 1),    # 1个执行器
#     ("spark.shuffle.service.enabled", True)
#   ('spark.sql.pivotMaxValues', '99999'),  # 当需要pivot DF，且值很多时，需要修改，默
)
# 查看更详细配置及说明：https://spark.apache.org/docs/latest/configuration.html

conf.setAll(config)

# 利用config对象，创建spark session
spark = SparkSession.builder.config(conf=conf).enableHiveSupport().getOrCreate()
```

In [2]:
```
sku_tag_weights = spark.sql('select * from sku_tag_merge_weights')
# tag就是tfidf那章 每个sku_id对应的summary 切完词之后的关键词
sku_tag_weights.show()
```

In [4]:
```
# sku_tag_weights.groupBy('tag').count().count()
# 38328
```

In [6]:
```
# 试着查看下 1号sku所有的关键词+权重
spark.sql('select * from sku_tag_merge_weights where sku_id = 1 order by weights').sho
```

```
+------+--------+-------+------------------+------------------+------------------+
|sku_id|industry|    tag|           textrank|             tfidf|           weights|
+------+--------+-------+------------------+------------------+------------------+
|     1|  电子产品|   颜色| 0.2530432277207418|0.01276759418263924| 0.1329054109516905|
|     1|  电子产品|   产品| 0.4408521260642171|0.09632014671815864| 0.2685861363911879|
|     1|  电子产品|   版本| 0.4975153001572373|0.06629382447805561|0.28190456231764643|
|     1|  电子产品|   特惠|0.36817954724375584|0.22333878637923998|0.29575916681149794|
|     1|  电子产品|   内存|  0.446169908963079|0.14835294069708402| 0.2972614248300815|
|     1|  电子产品|   官方|0.47547103618754005|0.13582584454306587|  0.305648440365303|
|     1|  电子产品|   尺寸| 0.5566745152156337|0.19895352844206426|  0.377814021828849|
|     1|  电子产品|   屏幕| 0.5578306702654822| 0.2235040348005383| 0.3906673525330102|
|     1|  电子产品|   电脑| 0.6908128340350108| 0.0975781721443329| 0.3941955030896719|
|     1|  电子产品|   即发| 0.5205724449688625| 0.2792169451200764| 0.3998946950444694|
|     1|  电子产品|   才华|0.41078350930249075| 0.5267698970521265|0.46877670317730863|
|     1|  电子产品|   银色| 0.6435786348103103|0.31994359335574984| 0.4817611140830301|
|     1|  电子产品|   黑五|0.46917715195597554| 0.5267698970521265|  0.497973524504051|
|     1|  电子产品|  core| 0.4930019268112531| 0.5267698970521265| 0.5098859119316899|
|     1|  电子产品|  笔记本| 0.8552990950150414| 0.2962833493350267| 0.5757912221750341|
|     1|  电子产品|   英寸|               1.0|0.19796136718089044| 0.5989806835904452|
|     1|  电子产品|   Pro| 0.8179559858686922|0.40670814294663504| 0.6123320644076636|
|     1|  电子产品| Apple| 0.9654271256288627|0.49085358694295506| 0.7281403562859089|
|     1|  电子产品|MacBook| 0.9649696261559032| 0.6213930153243914| 0.7931813207401472|
+------+--------+-------+------------------+------------------+------------------+
```

In [3]:
```
from pyspark.ml.feature import Word2VecModel
word2vec_model = Word2VecModel.load('/meiduo_mall/models/电子产品.word2vec_model')
# w2v训练后 保留的关键词
vectors = word2vec_model.getVectors()
```

In [16]:
```
vectors.show()
```

```
+----------+--------------------+
|      word|              vector|
+----------+--------------------+
|        钟爱|[0.02725963108241...|
|        伙伴|[0.01822851598262...|
|    uhd566|[-0.0026338319294...|
|       体重秤|[0.41525700688362...|
|        咪头|[0.18375480175018...|
|        人物|[0.13888520002365...|
|mini25star|[0.10265478491783...|
|    iP7280|[-0.3177461028099...|
|        红龙|[0.08126184344291...|
|    ALW17C|[0.02772485837340...|
|       S18|[-0.3153381943702...|
|        淡粉|[-0.0277402922511...|
|      A550|[0.10420309007167...|
|        体脂|[0.39589273929595...|
|       R80|[0.01038890704512...|
|        环球|[-0.1373830288648...|
|        傲石|[0.11340291798114...|
|        热水|[-0.0937786400318...|
|    iP4980|[-0.1033011823892...|
|        变光|[0.00825320743024...|
+----------+--------------------+
only showing top 20 rows
```

In [ ]:
```
# 每个关键词的embedding向量 列 拼接到'sku_id-权重表sku_tag_merge_weights'上
```

In [14]:
```
print('w2v训练前电子产品所有的关键词总数(不重复)是：',sku_tag_weights.groupBy('tag').c
print('w2v训练后得到的关键词(此时关键词已被用embedding表示)总数：',vectors.count())
# 可见：部分词在wrod2vec模型中不存在，因此必须使用inner join，舍弃掉这次，他们数量较少
# 因此inner join
```

```
w2v训练前电子产品所有的关键词总数(不重复)是： 38328
w2v训练后得到的关键词(此时关键词已被用embedding表示)总数： 18121
```

In [4]:
```
_ = sku_tag_weights.join(vectors,[sku_tag_weights.tag==vectors.word],'inner')
_.show(1)#带有embedding词向量
```

In [18]:
```
print(_.count())
sku_tag_weights.count()
```

```
1139798
```

Out[18]:
```
1171863
```

In [8]:
```
#使用每个词的综合权重乘以每个词的词向量
sku_tag_vector = _.rdd.map(lambda r:(r.sku_id, r.tag, r.weights*r.vector)).toDF(["sku
sku_tag_vector.show()
```

```
+------+-------+--------------------+
|sku_id|    tag|              vector|
+------+-------+--------------------+
|    85|     数码|[0.01333410454927...|
|   172|   USB2|[-0.0582328234302...|
|   182|     型号|[-0.5986965058535...|
|   190|    雷克沙|[-0.0290172319018...|
|   271|   数码配件|[-0.0378638870371...|
|   282|    香槟色|[-0.0372031283570...|
|   305|     星空|[0.08105157660438...|
|   312|   SONY|[-0.0988841883341...|
|   326|     川宇|[-0.1222147164901...|
|   334| TOPSSD|[-0.0103135671827...|
|   334|     天硕|[-0.0202074556195...|
|   351|   数码配件|[-0.0820574445237...|
|   370|     功能|[0.03261207532459...|
|   403|   数码配件|[-0.0648847702723...|
|   410|MicroSD|[-0.0169194095194...|
|   414|     颜色|[4.06892799643887...|
|   441|    手机卡|[-0.0184025048013...|
|   441|     数码|[0.00417148979608...|
|   450|   数码配件|[-0.0678617514344...|
|   456|     数码|[0.00961656805449...|
+------+-------+--------------------+
only showing top 20 rows
```

注意向量与列表的相加是不同的
DenseVector([1,1])+DenseVector([2,2])=DenseVector([3,3])
但是[1,1]+[2,2]=[1,1,2,2]

In [ ]:
```
# 每个sku的 embedding词向量表示
```

In [9]:
```
sku_tag_vector.registerTempTable("tempTable")

def map(row):
    x = 0
    for v in row.vectors:
        x += v
    #  将平均向量作为sku的向量
    return row.sku_id, x/len(row.vectors)

sku_vector = spark.sql("select sku_id, collect_set(vector) vectors from tempTable grou
sku_vector.show()
```

```
+------+--------------------+
|sku_id|              vector|
+------+--------------------+
|    26|[0.01235398195028...|
|    29|[0.21760844687031...|
|   474|[-0.0423007059418...|
|   964|[0.08240807672413...|
|  1677|[0.02065021965251...|
|  1697|[0.39568690077127...|
|  1806|[0.05530353810490...|
|  1950|[0.13603717113579...|
|  2040|[-0.0108802742841...|
|  2214|[-0.0480597909252...|
|  2250|[0.03186294420201...|
|  2453|[-0.0384253501587...|
|  2509|[-0.0221520914506...|
|  2529|[-0.0486537793846...|
|  2927|[-0.0213679434397...|
|  3091|[-0.0477536702739...|
|  3506|[-0.1093108035922...|
|  3764|[0.01326956604482...|
|  4590|[0.16880469210238...|
|  4823|[0.13785943123954...|
+------+--------------------+
only showing top 20 rows
```

In [ ]:
```

```

In [ ]:
```
# 计算皮尔逊相似度
```

In [33]:
```
from pyspark.mllib.stat import Statistics
v1 = sku_vector.where('sku_id=1').select('vector').first().vector # DenseVector
v2 = sku_vector.where('sku_id=2').select('vector').first().vector
sc=spark.sparkContext
x = sc.parallelize(v1) # rdd
y = sc.parallelize(v2)
Statistics.corr(x,y,method='pearson')
```

Out[33]: 0.9739711347735766

In [35]:

```
# 查看DenseVector
# sku_vector.where('sku_id=1').select('vector').first().vector
# DenseVector([...])
```

In [52]:

```
# 延申学习：spark.createDataFrame的建立方式之一；内外连接
# test1=spark.createDataFrame([
#     (1,'ni'),
#     (2,'hao'),
#     (3,'ma')
# ],['id','content'])
# test1.show()
# test2=spark.createDataFrame([
#     (1,'ni'),
#     (2,'hao'),
#     (3,'ma')
# ],['id2','content2'])
# test2.show()
# test1.join(test2,[test1.id!=test2.id2],'outer').show()
# test1.join(test2,[test1.id!=test2.id2],'inner').show()
# +---+-------+
# | id|content|
# +---+-------+
# |  1|     ni|
# |  2|    hao|
# |  3|     ma|
# +---+-------+

# +---+--------+
# |id2|content2|
# +---+--------+
# |  1|      ni|
# |  2|     hao|
# |  3|      ma|
# +---+--------+

# +---+-------+---+--------+
# | id|content|id2|content2|
# +---+-------+---+--------+
# |  1|     ni|  2|     hao|
# |  1|     ni|  3|      ma|
# |  2|    hao|  1|      ni|
# |  2|    hao|  3|      ma|
# |  3|     ma|  1|      ni|
# |  3|     ma|  2|     hao|
# +---+-------+---+--------+

# +---+-------+---+--------+
# | id|content|id2|content2|
# +---+-------+---+--------+
# |  1|     ni|  2|     hao|
# |  1|     ni|  3|      ma|
# |  2|    hao|  1|      ni|
# |  2|    hao|  3|      ma|
# |  3|     ma|  1|      ni|
# |  3|     ma|  2|     hao|
# +---+-------+---+--------+
```

In [55]:
```
#=========================很慢==========================
# temp_df = sku_vector.withColumnRenamed("sku_id", "sku_id2").withColumnRenamed("vecto
# import time
# start = time.time()
# ### 注意注意注意：这里一定不要用inner join，因为内连接由于会剔除条左右表中不存在的条
# ### 因此它会有过滤操作，在数据量极其大的情况下非常慢，注意是非常慢
# print(sku_vector.join(temp_df, sku_vector.sku_id!=temp_df.sku_id2, how="inner").cou
# print(time.time()-start)
```

...

In [61]:
```
# 这里其实本身是一个自连接，不会有不存在的条目，所以直接用outer，加快速度
temp_df = sku_vector.withColumnRenamed("sku_id", "sku_id2").withColumnRenamed("vector
import time
start = time.time()
print(sku_vector.join(temp_df, sku_vector.sku_id!=temp_df.sku_id2, how="outer").count
print(time.time()-start)
# sku_vector.count()#有个6w6个sku，两两sku计算，共44亿次
# 66651 * 66651 约44亿条目
```

```
4442289150
426.5052945613861
```

In [57]:
```
66651 * 66651 - 66651
```

Out[57]:
```
4442289150
```

In [62]:
```
sku_vector.join(temp_df, sku_vector.sku_id!=temp_df.sku_id2, how="outer").show(100)
```

```
+------+--------------------+-------+--------------------+
|sku_id|              vector|sku_id2|             vector2|
+------+--------------------+-------+--------------------+
|    26|[-0.0012209568107...|     29|[0.04843440780716...|
|    26|[-0.0012209568107...|    474|[-0.0435095790902...|
|    26|[-0.0012209568107...|    964|[-0.0222628132554...|
|    26|[-0.0012209568107...|   1677|[-0.0038834392417...|
|    26|[-0.0012209568107...|   1697|[0.11111611819181...|
|    26|[-0.0012209568107...|   1806|[0.01085346123940...|
|    26|[-0.0012209568107...|   1950|[0.01838842107055...|
|    26|[-0.0012209568107...|   2040|[0.09027117856091...|
|    26|[-0.0012209568107...|   2214|[0.03547335331710...|
|    26|[-0.0012209568107...|   2250|[0.00117475391754...|
|    26|[-0.0012209568107...|   2453|[0.07669576251528...|
|    26|[-0.0012209568107...|   2509|[0.00907962355502...|
|    26|[-0.0012209568107...|   2529|[0.01184689227830...|
|    26|[-0.0012209568107...|   2927|[-0.0012751391840...|
|    26|[-0.0012209568107...|   3091|[-0.0032631794437...|
|    26|[-0.0012209568107...|   3506|[-0.0658730276938...|
```

In [ ]:
```
import numpy as np
# np.linalg = linear algebra
temp_df = sku_vector.withColumnRenamed('sku_id','sku_id2').withColumnRenamed('vector',
sku_vector_join = sku_vector.join(temp_df,[sku_vector.sku_id!=temp_df.sku_id2],'outer

def mapPartitions(partition):
    import numpy as np
    for row in partition:
        sim = np.dot(row.vector,row.vector2)/(np.linalg.norm(vector1)*(np.linalg.norm
        yield row.sku_id,row.sku_id2,float(sim)

similarity = sku_vector_join.rdd.mapPartitions(mapPartitions).toDF(['sku_id','sku_id2'
'''
# =======跳过，延伸学习：如何求范数-模=====
import numpy as np
vector1 = v1
vector2 = v2
np.dot(vector1,vector2)/(np.linalg.norm(vector1)*np.linalg.norm(vector2))
# 0.9737596995475859
'''
```

In [67]:
```
similarity.show()
```

```
+------+-------+--------------------+
|sku_id|sku_id2|                 sim|
+------+-------+--------------------+
|    26|     29|  0.3266431231283569|
|    26|    474| 0.08736009761829149|
|    26|    964| 0.14403582542295462|
|    26|   1677| 0.05289843618892875|
|    26|   1697| 0.09706387852546618|
|    26|   1806| 0.04812759823324625|
|    26|   1950| 0.06598719786681272|
|    26|   2040| 0.23167814841316745|
|    26|   2214|     0.16516129321389|
|    26|   2250| 0.07333314880543064|
|    26|   2453| 0.10024709344450417|
|    26|   2509| 0.14559026351877366|
|    26|   2529| 0.03967589505832165|
|    26|   2927| 0.09751409303471345|
|    26|   3091|0.028248762853119336|
|    26|   3506|-0.03034581124060582|
|    26|   3764| 0.15092135024451372|
|    26|   4590| 0.25749362940782733|
|    26|   4823| 0.21968089633654198|
|    26|   4894|  0.3175731390319865|
+------+-------+--------------------+
only showing top 20 rows
```

In [70]:
```
# 由于条目数实在太多，如果这里进行如查询、分组等操作需要大量的计算，对硬件要求很高，所
similarity.where("sku_id=1").show()
```

...

In [78]:
```
1 sku_vector_join
```

Out[78]:
```
DataFrame[sku_id: bigint, vector: vector, sku_id2: bigint, vector2: vector]
```

In [9]:

```
# 这里考虑使用foreachPartition方法，遍历每一行数据，进行运算
# 将结果直接存储进入redis，并且每一个商品只保留与它相似的TOP100个商品的sku_id和对应的
# 那么这里的结果其实就是一个基于商品相似度的一个召回结果集

#与sku_id相似的元素存够100个了，就取出相似度最小的score,member，与当前计算出来的相似度
# 当前大，就先删除取出来的，再将当前加进去；否则，直接添加
import numpy as np
import gc
import redis


temp_df = sku_vector.withColumnRenamed("sku_id", "sku_id2").withColumnRenamed("vector
sku_vector_join = sku_vector.join(temp_df, sku_vector.sku_id!=temp_df.sku_id2, how="o

def foreachPartition(partition):
    import redis

    client = redis.StrictRedis(host="192.168.58.100", port=6379, db=0)#默认"host='loc

    for row in partition:
        vector1 = row.vector
        vector2 = row.vector2

        # 余弦相似度计算公式
        sim = np.dot(vector1,vector2)/(np.linalg.norm(vector1)*(np.linalg.norm(vector

        # 有序集合(sorted set)：按照分数排序，默认升序
        if client.zcard(row.sku_id) < 100:# 返回有序集合sku_id中元素的个数
            # redis.zadd('my-key', 'name1', 1.1, 'name2', 2.2, name3=3.3, name4=4.4)
            # 交互式中 的提示是：key, score, member,...
            client.zadd(row.sku_id, float(sim), row.sku_id2)# 给sku_id中对应的集合添加
        else:#与sku_id相似的元素存够100个了，就取出相似度最小的score,member，与当前计
            # 取出当前redis中与sku_id相似度最小的sku_id2，即add中的member
            key = client.zrange(row.sku_id, 0, 0)# 返回指定score区间（全闭区间[start,
            # 返回某个member的score
            min_sim = client.zscore(row.sku_id, key[0]) if len(key) == 1 else None#l

            if min_sim is None:
                client.zadd(row.sku_id, float(sim), row.sku_id2)
            else:
                if sim > min_sim:
                    client.zrem(row.sku_id, key[0])#Remove member ``values`` from sort
                    client.zadd(row.sku_id, float(sim), row.sku_id2)
        #
            del key
            del min_sim
        del vector1
        del vector2
        del sim
        del row
        gc.collect()
        # 这里为了节约内存开销，手动进行内存回收，避免内存泄漏等问题

sku_vector_join.foreachPartition(foreachPartition)

# 但注意这里，由于计算量比较大， 非常耗时
```

```
# 光是计算6w条电子产品的相似就要这这么久时间，那么如果全部大类都要计算完，或者说数量量
# 对于商品之间的相似度计算，其实往往只需要一开始对所有的商品进行两两计算，比如6w条，那
# 但是其实后面如果有新增商品，只需要对新增商品与其他商品的相似度进行计算就可以，那么新
# 也就是 只需要在一开始把所有的都算一遍，之后更新的时候，采取的是增量更新
# 或者根据业务发展情况，每隔几个月做一次全量更新，期间就只做增量更新


# 一下报错，是因为手动关闭程序导致的，因为这里没有等全部算出结果，只算了一部分，就关闭
```

延伸学习：spark中也有多种相似度计算的API

如pyspark.ml.stat.Correlation、pyspark.mllib.stat.Statistics 但这些的相似度计算都只能完成所有列两两之间的相似度计算，如果是计算所有行两两之间的相似度，就需要自行去实现

In [3]:
```
# 使用了稠密向量稀疏向量
from pyspark.ml.linalg import Vectors#lin+alg 线性代数
from pyspark.ml.stat import Correlation
dataset = [
    [Vectors.dense([1,0,0,-2])],
    [Vectors.dense([4,5,0,3])],
    [Vectors.dense([6,7,0,8])],
    [Vectors.dense([9,0,0,1])],
    [Vectors.dense([1,0,0,-2])]
]
dataset = spark.createDataFrame(dataset,['features'])
dataset.show()
pearsonCorr = Correlation.corr(dataset,'features','pearson').show(truncate=False)

spearmanCorr = Correlation.corr(dataset,'features',method='spearman').show(truncate=F
```

```
+-----------------+
|         features|
+-----------------+
|[1.0,0.0,0.0,-2.0]|
| [4.0,5.0,0.0,3.0]|
| [6.0,7.0,0.0,8.0]|
| [9.0,0.0,0.0,1.0]|
|[1.0,0.0,0.0,-2.0]|
+-----------------+

+--------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------
---+
|pearson(features)
|
+--------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------
---+
|1.0                  0.2522120576380154  NaN  0.5517643975445568
0.2522120576380154  1.0                 NaN  0.9262057975392757
NaN                 NaN                 1.0  NaN
0.5517643975445568  0.9262057975392757  NaN  1.0                 |
+--------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------
---+

+--------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------
---+
|spearman(features)
|
+--------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------
---+
|1.0                  0.3441236008058426  NaN  0.6842105263157896
0.3441236008058426  1.0                 NaN  0.917662935482247
```

```
NaN                 NaN                 1.0  NaN
0.6842105263157896  0.917662935482247   NaN  1.0                 |
+-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
---+
```

In [20]:
```
1 Vectors.dense([1, 0, 0, -2])/2
```

Out[20]:
```
DenseVector([0.5, 0.0, 0.0, -1.0])
```

In [21]:
```
1  from __future__ import print_function
2
3  import numpy as np
4
5  from pyspark import SparkContext
6  # $example on$
7  from pyspark.mllib.stat import Statistics
8  # $example off$
9
10 if __name__ == "__main__":
11     sc = spark.sparkContext
12
13     # $example on$
14     seriesX = sc.parallelize([1.0, 2.0, 3.0, 3.0, 5.0])  # a series
15     # seriesY must have the same number of partitions and cardinality as seriesX
16     seriesY = sc.parallelize([11.0, 22.0, 33.0, 33.0, 555.0])
17
18     # Compute the correlation using Pearson's method. Enter "spearman" for Spearman's
19     # If a method is not specified, Pearson's method will be used by default.
20     print("Correlation is: " + str(Statistics.corr(seriesX, seriesY, method="pearson"
21
22     data = sc.parallelize(
23         [np.array([1.0, 10.0, 100.0]), np.array([2.0, 20.0, 200.0]), np.array([5.0, 33
24     )  # an RDD of Vectors
25
26     # calculate the correlation matrix using Pearson's method. Use "spearman" for Spea
27     # If a method is not specified, Pearson's method will be used by default.
28     print(Statistics.corr(data, method="pearson"))
29     # $example off$
```

```
Correlation is: 0.8500286768773001
[[1.         0.98473193 0.99316078]
 [0.98473193 1.         0.99832152]
 [0.99316078 0.99832152 1.        ]]
```

In [22]:
```
from pyspark.mllib.stat import Statistics
from pyspark.mllib.linalg import Vectors
rdd = sc.parallelize([Vectors.dense([1, 0, 0, -2]), Vectors.dense([4, 5, 0, 3]),Vecto
pearsonCorr = Statistics.corr(rdd)
pearsonCorr
```

Out[22]:
```
array([[ 1.        , -0.16524238,         nan,  0.24090545],
       [-0.16524238,  1.        ,         nan,  0.89613897],
       [        nan,         nan,  1.        ,         nan],
       [ 0.24090545,  0.89613897,         nan,  1.        ]])
```