2019 级

《物联网数据存储与管理》课程

实 验 报 告

姓　名　薛海波

学　号　U201914971

班　号　物联网 1901 班

日　期　2022.04.18

目　　录

一、选题背景

局部敏感哈希，英文 locality-sensetive hashing，常简称为 LSH。局部敏感哈希在部分中文文献中也会被称做位置敏感哈希。LSH 是一种哈希算法，最早在 1998 年由 Indyk 在上提出。不同于我们在数据结构教材中对哈希算法的认识，哈希最开始是为了减少冲突方便快速增删改查，在这里 LSH 恰恰相反，它利用的正是哈希冲突加速检索，并且效果极其明显。

LSH 主要运用到高维海量数据的快速近似查找。近似查找便是比较数据点之间的距离或者是相似度。因此，很明显，LSH 是向量空间模型下的东西。一切数据都是以点或者说以向量的形式表现出来的。

二、理论分析

2.1 定义

LSH 不像树形结构的方法可以得到精确的结果，LSH 所得到的是一个近似的结果，因为在很多领域中并不需非常高的精确度。即使是近似解，但有时候这个近似程度几乎和精准解一致。

LSH 的主要思想是，高维空间的两点若距离很近，那么设计一种哈希函数对这两点进行哈希值计算，使得他们哈希值有很大的概率是一样的。同时若两点之间的距离较远，他们哈希值相同的概率会很小。给出 LSH 的定义如下：

给定一族哈希函数 H，H 是一个从欧式空间 S 到哈希编码空间 U 的映射。如果以下两个条件都满足，则称此哈希函数满足 (r1, r2, p1, p2) 性。

若 p∈B (q, r1)则 Pr [h(q) = h(p)]≥p1

若 p∉B (q, r2)则 Pr [h(q) = h(p)]≤p2

定义中 B 表示的是以 q 为中心，r1 或 r2 为半径的空间。p、q 表示两个具有多维属性的数据对象，Pr [h(q) = h(p)] 为 2 个对象的相异程度，也就是 1/相似度。

简单来说，就是当足够相似时，映射为同一 hash 值的概率足够大；而足够不相似时，映射为同一 hash 值的概率足够小。图 1 可以更好的表示这个定义。

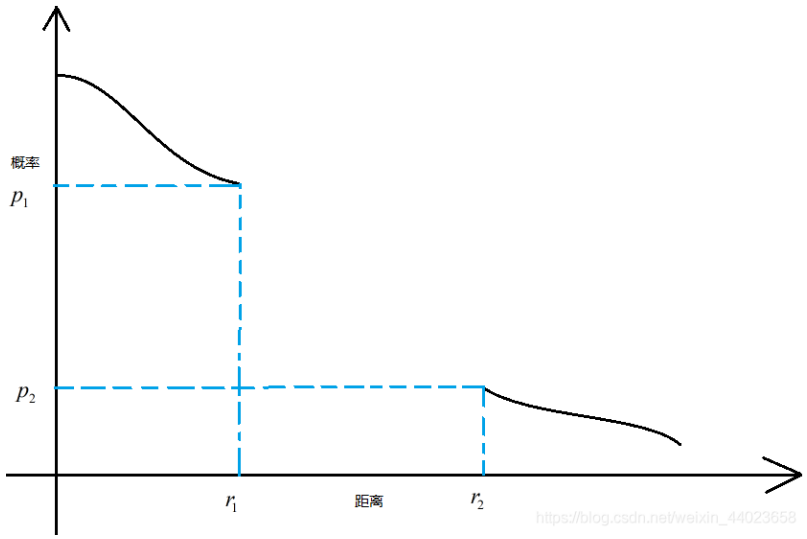

图 1 距离-概率示意图

2.2 分类

上文中相似度的定义根据实际情况决定。针对不同的相似度测量方法，局部敏感哈希的算法设计也不同。一般来说，在两种最常用的相似度下，有两种不同

的 LSH：

1. 使用 Jaccard 系数度量数据相似度时的 min-hash

2. 使用欧氏距离度量数据相似度时的 P-stable hash

当然，无论是哪种 LSH，其实说白了，都是将高维数据降维到低维数据，同时，还能在一定程度上，保持原始数据的相似度不变。LSH 不是确定性的，而是概率性的，也就是说有一定的概率导致原本很相似的数据映射成 2 个不同的 hash 值，或者原本不相似的数据映射成同一 hash 值。这是高维数据降维过程中所不能避免的（因为降维势必会造成某种程度上数据的失真），不过好在 LSH 的设计能够通过相应的参数控制出现这种错误的概率，这也是 LSH 为什么被广泛应用的原因。

本次实验主要使用的是 min-hash。

2.3 Jaccard 系数

上文的分类中提到，min-hash 是使用 Jaccard 系数度量数据相似度时的。

Jaccard 系数主要用来解决的是非对称二元属性相似度的度量问题，常用的场景是度量 2 个集合之间的相似度，具体公式是 2 个集合的交比 2 个集合的并。

比如，现在有如**表 1** 所示的 4 个文档对象，每个文档有相应的单词。若某个文档存在这个单词，则标为 1，否则标 0。

表 1 文档对象

	D1	D2	D3	D4
w1	1	0	0	1
w2	0	0	1	1
w3	1	0	1	1
w4	0	1	1	0
w5	0	1	0	0

首先，我们现在将上面这个单词和文档组成的矩阵按行置换，比如可以置换成如**表 2** 所示的形式：

表 2 置换后文档

	D1	D2	D3	D4
w3	1	0	1	1
w1	1	0	0	1
w5	0	1	0	0
w4	0	1	1	0
w2	0	0	1	1

可以确定的是，这没有改变文档与词项的关系。现在做这样一件事：对这个

矩阵按行进行多次置换，每次置换之后，统计每一列（其实对应的就是每个文档）第一个不为 0 的位置（行号），这样每次统计的结果能构成一个与文档数等大的向量，这个向量，我们称之为签名向量。

比如，如果对最上面的矩阵做这样的统计，得到[1, 4, 2, 1]，对于下面的矩阵做统计，得到[1, 3, 1, 1]。

简单来想这个问题，就拿上面的文档来说，如果两个文档足够相似，那也就是说这两个文档中有很多元素是共有的，换句话说，这样置换之后统计出来的签名向量，如果其中有一些文档的相似度很高，那么这些文档所对应的签名向量的相应的元素，值相同的概率就很高。

比如上面的文档中文档 D1 与 D4 仅有 w2 不同，所以两次得到的签名向量均相同。

实际上，置换原矩阵的行，取每列第一个非 0 元的做法，就是一个 hash 函数。这个 hash 函数成功地将多维数据映射成了一维数据。并且这样的映射没有改变数据相似度。

2.4 构造 LSH 函数族

我们把最初始时的矩阵叫做 input matrix，由个文档，个词项组成。而把由次置换后得到的一个的矩阵叫做 signature matrix。

为了能实现前面 LSH 定义中的 2 个条件的要求，我们通过多次置换，求取向量，构建了一组 hash 函数。也就是最终得到了一个 signature matrix. 为了控制相似度与映射概率之间的关系，我们需要按下面的操作进行，一共三步。

(1) 将 signature matrix 水平分割成一些区块（记为 band），每个 band 包含了 signature matrix 中的行。需要注意的是，同一列的每个 band 都是属于同一个文档的。如**图 2** 所示。

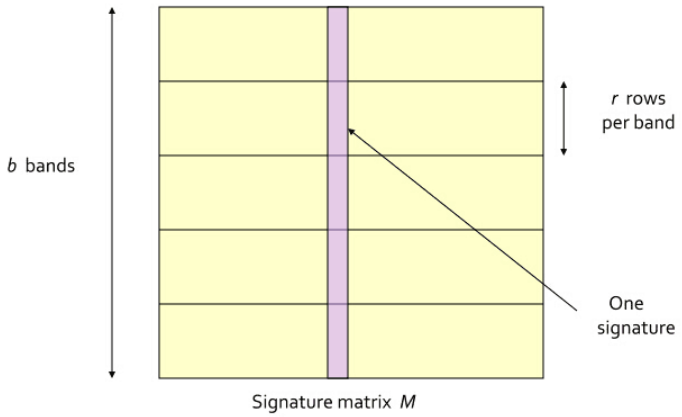

图 2 矩阵分割

(2) 对每个 band 计算 hash 值，这里的 hash 算法没有特殊要求，MD5，SHA1 等等均可。一般情况下，我们需要将这些 hash 值做处理，使之成为事先设定好的 hash 桶的 tag，然后把这些 band“扔”进 hash 桶中。如**图 3** 所示。但是这里，我们只是关注算法原理，不考虑实际操作的效率问题。所以，省略处理 hash 值得这一项，得到每个 band 的 hash 值就 OK 了，这个 hash 值也就作为每个 hash bucket 的。

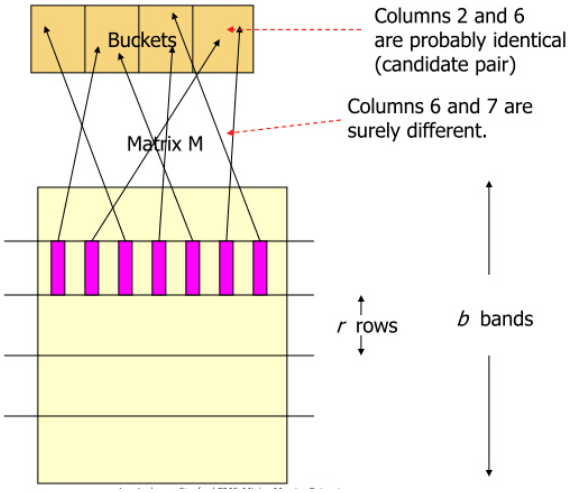

图 3 计算 hash

(3) 如果某两个文档的，同一水平方向上的 band，映射成了同一 hash 值（如果你选的 hash 函数比较安全，抗碰撞性好，那这基本说明这两个 band 是一样的），我们就将这两个文档映射到同一个 hash bucket 中，也就是认为这两个文档是足够相近的。

好了，既然执行的是上面三步的操作，那不难计算出两个文档被映射到同一个 hash bucket 中的概率：

（1）. 对于两个文档的任意一个 band 来说，这两个 band 值相同的概率是：

s^r，其中 s∈[0,1]是这两个文档的相似度。

（2）. 也就是说，这两个 band 不相同的概率是 1−s^r

（3）. 这两个文档一共存在 b 个 band，这 b 个 band 都不相同的概率是(1−s^r)^b

（4）. 所以说，这 b 个 band 至少有一个相同的概率是 1−(1−s^r)^b

这个方法先要求每个 band 的所有对应元素必须都相同，再要求多个 band 中至少有一个相同。符合这两条，才能发生 hash 碰撞。概率就是最终两个文档被映射到同一个 hash bucket 中的概率。

我们发现，这样一来，实际上可以通过控制参数的值来控制两个文档被映射到同一个哈希桶的概率。而且效果非常好。比如，令 r = b = 5

（1）. 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−(1−0.8^5)^5 = 0.862633715

（2）. 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−(1−0.2^5)^5 = 0.001598976

不难看出，这样的设计通过调节参数值，达到了“越相似，越容易在一个哈希桶；越不相似，越不容易在一个哈希桶”的效果。这也就能实现我们上边说的 LSH 的两个性质。

图 4、图 5、图 6 分别是在参数为（r = b = 5），（r = 5，b = 10），（r = 10，b = 5）时参数 s 与概率之间的关系。

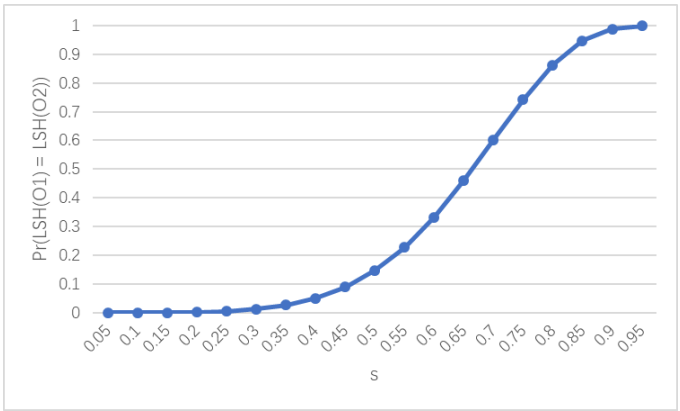

图 4 （r = b = 5）的关系

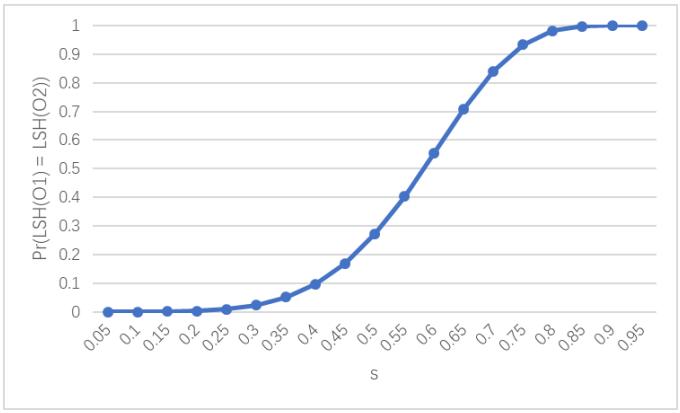

图 5 （r = 5，b = 10）的关系

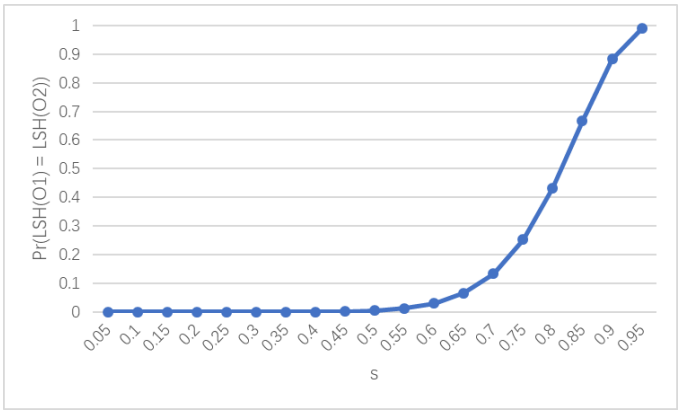

图 6 （r = 10，b = 5）的关系

可见，在参数合适的情况下，当相似度高于某个值的时候，概率会变得非常大，并且快速靠近 1，而当相似度低于某个值的时候，概率会变得非常小，并且快速靠近 0。

三、实验设计

本次实验使用的数据集为来自 kaggle 网站的一个有关心脏病相关特征的统计数据集，该数据集的具体网址为：

https://www.kaggle.com/datasets/kamilpytlak/personal-key-indicators-of-heart-disease

该数据集主要包含两部分：一部分表示是否患有心脏病，标签为 HeartDisease，值为 Yes 或 No；另一部分表示病人的其它生理、心理特征，共计 17 种不同特征，这些特征的标签和值的类型如**图 7** 所示。

B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
BMI	Smoking	AlcoholDr	Stroke	PhysicalHe	MentalHe	DiffWalkin	Sex	AgeCateg	Race	Diabetic	PhysicalAc	GenHealth	SleepTime	Asthma	KidneyDis	SkinCancer
16.6	Yes	No	No	3	30	No	Female	55-59	White	Yes	Yes	Very gooc	5	Yes	No	Yes
20.34	No	No	Yes	0	0	No	Female	80 or olde	White	No	Yes	Very gooc	7	No	No	No
26.58	Yes	No	No	20	30	No	Male	65-69	White	Yes	Yes	Fair	8	Yes	No	No
24.21	No	No	No	0	0	No	Female	75-79	White	No	No	Good	6	No	No	Yes
23.71	No	No	No	28	0	Yes	Female	40-44	White	No	Yes	Very gooc	8	No	No	No
28.87	Yes	No	No	6	0	Yes	Female	75-79	Black	No	No	Fair	12	No	No	No
21.63	No	No	No	15	0	No	Female	70-74	White	No	Yes	Fair	4	Yes	No	Yes
31.64	Yes	No	No	5	0	Yes	Female	80 or olde	White	Yes	No	Good	9	Yes	No	No
26.45	No	No	No	0	0	No	Female	80 or olde	White	No, borde	No	Fair	5	No	Yes	No

图 7 各项特征

现在，我希望可以借助上文提到的数据集和局部敏感哈希算法（LSH）完成一次简单的大数据处理。即根据给出的病人特征，预测该病人是否有可能患有心脏病。

由于我没有其它数据，因此首先对该数据集进行划分。我事先已经知道，该数据集共包含超过 30000 条数据，所以将数据集的 95%随机划分为训练集，剩下的数据作为验证集。该部分代码如下。

```
# 数据集划分训练集、验证集
train_data = original_data.sample(frac=0.95)
print("Size of train data:", train_data.shape[0], train_data.shape[1])
test_data = original_data[~original_data.index.isin(train_data.index)]
print("Size of test data:", test_data.shape[0], test_data.shape[1])
```

之后对数据集做进一步处理：去除 HeartDisease 项，仅保留其它特征；数据重新编号；字符串编码。该部分代码如下。

```
# 整理数据
train_data = train_data.reset_index(drop=True)
train_feature = getFeatures(train_data)
print("Size of new train data:", train_feature.shape[0],
      train_feature.shape[1])
test_data = test_data.reset_index(drop=True)
test_feature = getFeatures(test_data)
print("Size of new test data:", test_feature.shape[0], test_feature.shape[1])
```

接下来，创建 lsh 对象，在创建的同时，设定 lsh 算法的各项参数。该部分代码如下。

```
# 创建 lsh 对象
lsh = MinHashLSH(threshold=0.8, num_perm=128)
```

其中，参数 threshold 表示算法的 Jaccard 距离阈值，带入当前数据集和选择的 0.8 的值，可以大致理解为：当新数据与现有数据有超过 80%的特征相同时，就认为二者相似。参数 num_perm 表示哈希置换函数设定个数，根据定义可知，该参数越大，预测效果越好，但对资源消耗越大。

然后，分别初始化训练集和验证集，并构建 lsh 索引，用于接下来的实际预测。该部分代码如下。

```
# 初始化训练集，创建 lsh 索引
for i in range(train_feature.shape[0]):
    minhash = MinHash(num_perm=128)
    for element in seriesEncode(train_feature.loc[i]):
        minhash.update(element)
    lsh.insert(i, minhash)

# 初始化验证集
minhashes = {}
for i in range(test_feature.shape[0]):
    minhash = MinHash(num_perm=128)
    for element in seriesEncode(test_feature.loc[i]):
        minhash.update(element)
    minhashes[i] = minhash
```

最后，预测结果。由于经过上述步骤后，python 的 datasketch 库中的 lsh 相关的方法返回的是相似结果的索引。所以需要根据索引进一步与包含 HeartDisease 项的训练集对比，再根据“少数服从多数”原则得出最终预测。对于 HeartDisease 项中“Yes”和“No”数量相同的情况，这里取“Yes”。该部分代码如下。

```
# 根据相似结果完成预测
def getFinalResult(dataSet, dataIndex):
    num_Yes = 0
    num_No = 0
    for i in dataIndex:
        if (dataSet.loc[i, 'HeartDisease'] == 'Yes'):
            num_Yes += 1
        else:
            num_No += 1
    if num_Yes >= num_No:
```

```
        return 'Yes'
    else:
        return 'No'
```

到目前为止，我已经得到了验证集的预测结果，但要知道预测的准确率，还需要将预测结果与原带有 HeartDisease 项的验证集对比，从而得到各项预测准确率。该部分代码如下。

```
# 计算准确率
def accuracyRate(real_data, test_data):
    num_true = 0
    num_falsePositive = 0
    num_falseNegative = 0
    length = real_data.shape[0]
    for i in range(length):
        if (real_data.loc[i, 'HeartDisease'] == 'Yes'):
            if (test_data [i] == 'Yes'):
                num_true += 1
            else:
                num_falsePositive += 1
        else:
            if (test_data [i] == 'No'):
                num_true += 1
            else:
                num_falseNegative += 1
    result_list = [
        num_true / length, num_falsePositive / length,
        num_falseNegative / length
    ]
    return result_list
```

四、实验结果

数据读取的结果如**图 8** 所示。

```
Size of original data: 319795 18
Size of train data: 303805 18
Size of test data: 15990 18
Size of new train data: 303805 17
Size of new test data: 15990 17
```

图 8 数据读取的结果

连续测试 3 次，3 次测试的结果分别如**图 9**、**图 10**、**图 11**、**图 12**、**图 13**、**图 14** 所示。

```
The result is: ['No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No',
Yes', 'No', 'No', 'Yes', 'No', 'No', 'Yes', 'No', 'No', 'No', 'No', '
'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No'
o', 'No', 'No', 'No', 'No', 'No', 'Yes', 'No', 'No', 'No', 'No', 'No',
', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', '
, 'No', 'No', 'No', 'No', 'No', 'Yes', 'No', 'No', 'No', 'No', 'No', '
'No', 'No', 'No', 'Yes', 'No', 'No', 'No', 'No', 'Yes', 'No', 'Yes', '
'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'Yes', 'No', 'N
'No', 'No', 'Yes', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'N
 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'N
No', 'No', 'Yes', 'No', 'No', 'No', 'No', 'No', 'No', 'Yes', 'No', 'N
```

图 9 第一次测试的部分结果

```
The true rate is: 0.8514696685428392
The falsePositive rate is: 0.0697936210131332
The falseNegative rate is: 0.07873671044402751
```

图 10 第一次测试的准确率

```
The result is: ['No', 'No', 'No', 'Yes', 'No', 'No', 'N
', 'Yes', 'No', 'No', 'Yes', 'No', 'Yes', 'Yes', 'No',
 'No', 'No', 'Yes', 'No', 'Yes', 'No', 'No', 'No', 'No'
 'No', 'No', 'No', 'Yes', 'No', 'Yes', 'No', 'No', 'Yes
', 'No', 'No', 'No', 'No', 'No', 'No', 'Yes', 'No', 'Ye
', 'No', 'No', 'Yes', 'No', 'No', 'No', 'Yes', 'No', 'N
'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', '
No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'Yes', '
 'Yes', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No',
 'No', 'No', 'No', 'No', 'No', 'Yes', 'No', 'No', 'No',
'No', 'Yes', 'No', 'No', 'No', 'No', 'No', 'No', 'No',
No', 'No', 'No', 'Yes', 'No', 'Yes', 'No', 'No', 'No',
```

图 11 第二次测试的部分结果

```
The true rate is: 0.8470293933708568
The falsePositive rate is: 0.07029393370856786
The falseNegative rate is: 0.08267667292057536
```

图 12 第二次测试的准确率

```
The result is: ['No', 'No', 'No', 'Yes', 'No', 'No',
es', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No',
', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', '
, 'No', 'No', 'No', 'Yes', 'No', 'Yes', 'No', 'No', '
, 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'N
, 'No', 'No', 'Yes', 'No', 'No', 'No', 'No', 'No', '
o', 'No', 'No', 'No', 'No', 'Yes', 'No', 'No', 'No',
', 'No', 'Yes', 'Yes', 'No', 'No', 'No', 'No', 'No',
'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes',
o', 'Yes', 'No', 'No', 'No', 'No', 'No', 'No', 'No',
```

图 13 第三次测试的部分结果

```
The true rate is: 0.8473420888055034
The falsePositive rate is: 0.07567229518449031
The falseNegative rate is: 0.07698561601000625
```

图 14 第三次测试的准确率

当 Visual Studio Code 在运行该 python 代码时，系统消耗的资源如**图 15** 所示。

图 15 资源消耗

根据测试结果，可以看出仅就该数据集而言，LSH 算法准确率较高。根据系统消耗的资源可以看出，LSH 算法对空间资源的消耗更大。

五、总结

本文首先分析了局部敏感哈希算法（LSH）出现的意义，并解释了 LSH 的定义及分类。之后重点分析了一种常用 LSH 算法——使用 Jaccard 系数度量数据相似度时的 min-hash 算法。

然后，我借助 python 已有的 LSH 库，用 min-hash-LSH 算法完成了一个心脏病预测实验。仅从预测成功率来看，实验较为成功。

不过，从 LSH 的原理和实际测试可以看出，LSH 较为消耗系统的空间资源。但我本人能力有限，无法在现有的算法基础上对其进行进一步的改进。

参考文献

[1] "Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions" (by Alexandr Andoni and Piotr Indyk). Communications of the ACM, vol. 51, no. 1, 2008, pp. 117-122.
[2] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, “Multi- Probe LSH: Efficient Indexing for High-Dimensional Similarity Search,” Proc. 33rd Int’l Conf. Very Large Data Bases (VLDB ’07), pp. 950-961, 2007.
[3] Yu Hua, Bin Xiao, Bharadwaj Veeravalli, Dan Feng. "Locality-Sensitive Bloom Filter for Approximate Membership Query", IEEE Transactions on Computers (TC), Vol. 61, No. 6, June 2012, pages: 817-830.
[4] Yu Hua, Xue Liu, Dan Feng, "Data Similarity-aware Computation Infrastructure for the Cloud", IEEE Transactions on Computers (TC), Vol.63, No.1, January 2014, pages: 3-16.