華中科技大學

课 程 报 告

报告名称：基于 LSH 的系统设计

专业班级：物联网工程 1601

学　　号：U2016148989

姓　　名：潘翔

指导教师：华宇

报告日期：2019.5

计算机科学与技术学院

摘要

局部敏感哈希算法(LSH)，作为一种被广泛采用的 Hash 算法，能够很好的对数据进行降维，从而作为 KNN 问题的粗解，在实际的应用系统(如推荐系统中),通常采用 LSH 进行候选集的获取，采用序列，属性的精确特征模型获取最终的召回集。本文分析了现存 LSH 的相关算法，尝试提出了结合主题，社群等**语义信息**的**分层次 LSH**，利用少种群，高语义的粗划分减少 LSH 的存储空间，从而提升 LSH 算法的性能，并进行了相关的实验验证，最终证明了相关算法性能的提升。结合语义的 LSH 算法能够更好的适合几何级增长的复杂网络和互联网数据，从而适应网络大数据系统部署。

关键词： LSH 性能分析 层次降维 语义模型 推荐系统

Abstract

The Local Sensitive Hash Algorithm (LSH), as a widely used Hash algorithm, can well reduce the dimension of data, thus serving as a rough solution to the KNN problem, in practical applications (such as recommendation systems). The LSH is used to acquire the candidate set, and the final recall set is obtained by using the sequence and the precise feature model of the attribute. This paper analyzes the existing LSH related algorithms, and attempts to propose a **Hierarchical LSH** that combines the S**emantic Information** of topics and communities, and reduces the storage space of LSH by using coarse population with low population and high semantics, thus improving the performance of LSH algorithm. Relevant experimental verification finally proved the performance improvement of related algorithms. The semantic LSH algorithm can be better suited to complex network and Internet data with geometric growth, thus adapting to the deployment of large data systems.

Keywords: LSH,Performance Analysis,Hierarchical Embedding,Semantic model,Recommendation System

目　录

1 LSH 综述

局部敏感哈希（LSH），它将相似的输入项强加到相同的“桶”中，具有很高的概率。（桶的数量远小于可能的输入项的范围）

由于类似的项目最终存在于相同的桶中，因此该技术可用于数据聚类和最近邻搜索。 它与传统的散列技术的不同之处在于散列冲突最大化，而不是最小化。作为一种降维的方法，在推荐系统和相应的匹配算法中有着重要的应用。

基于散列的近似最近邻搜索算法通常使用两种主要类别的散列方法之一：与数据无关的方法，例如局部敏感散列（LSH）；或依赖于数据的方法，如局部保留散列（LPH）。

对于不同的实体，采用 LSH 等 Hash 算法和通常进行系统处理进行建模的 Embedding 有着相似之处，其本质是进行实体表示和降维，从而方便后序搜索排序算法的实行。

通常可以将相似性分为以下两类：

1) 机械相似性

两个文本内容上的相关程度, 文本级别的相似(重复程度或者共现率)，此处应用场景为文本去重(搜索引擎)，相同实体合并(NLP)

2) 语义相似性。

两个文本语义上的相似程度

对于 Embedding，由于约束较弱，从而能够较快的实现定位和检索，通常使用 LSH 算法进行粗排序。对于相似性的处理，此时综合考虑到机械相似性的高性能和语义相似性的高准确度，能够在性能和准确度之间做出权衡。

2 公式化定义

2.1.1 KNN

最近邻查找（NN，Nearest Neighbor）,给定一拥有 nn 个点的点集 P，在此集合中寻找距离 q 点最近的一个点。

2.1.2 C-NN

给定一拥有 n 个点的点集 PP，在点集中寻找点 p，这个 p 满足

$$d(q,p) <= (1+c)d(q,P)d(q,p)$$

其中 d(q,P)d(q,P)是 P 中距离 q 点最近一点到 q 的距离。

2.1.3 LSH

给定一族哈希函数 HH，HH 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)(r1,r2,p1,p2)。

若 p∈B(q,r1)

$$p_{r_H}[h(q)=h(p)] \geq p_1$$

若 p∉B(q,r2)

$$p_{r_H}[h(q)=h(p)] \leq p_2$$

其中 B 为以 q 为中心，r1 或 r2 为半径的空间，此处理解为 hash 空间，其维度等于 hash 之后 embedding 的维度。

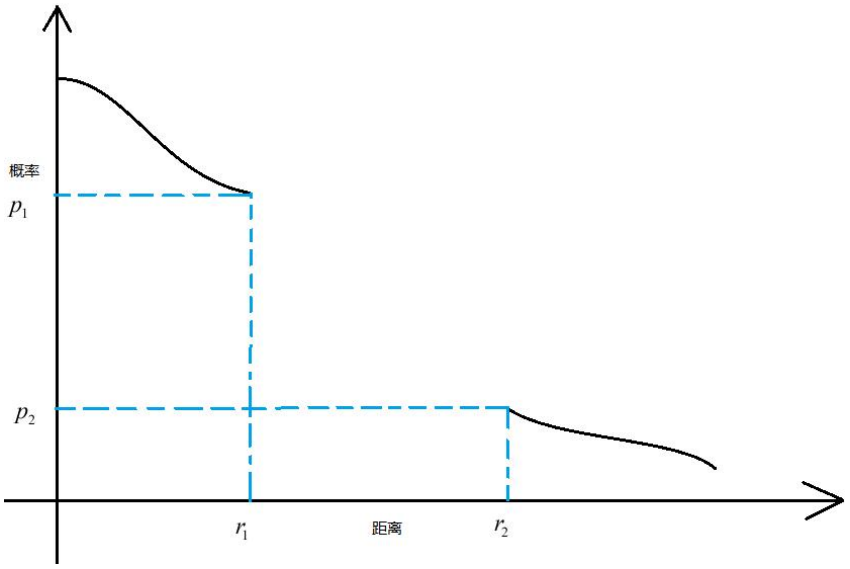


图 2.1 Hash 空间距离表示图

对于 LSH 函数的设计，关心的是其约束条件，具体的中间部分不一定为单调函数。

3 LSH 基础算法

3.1 LSH Pipeline

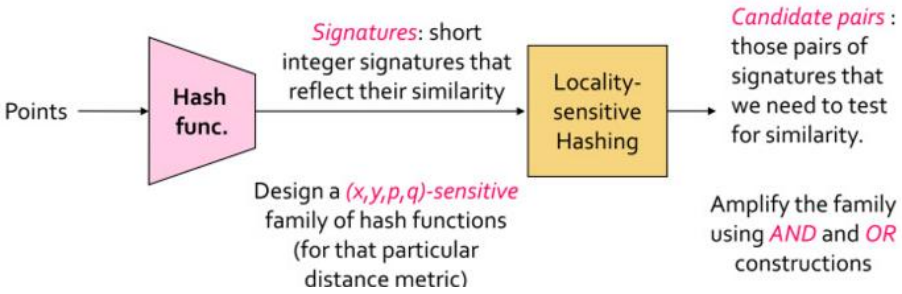


图 3.1 LSH 算法流程

1) 数据点（可以是原始数据，或者提取到的特征向量）组成矩阵；
2) 第一次 hash functions（有多个哈希函数，是从某个哈希函数族中选出来的）哈希成一个叫“签名矩阵（Signature Matrix）”的东西，这个矩阵可以直接理解为是降维后的数据，此时用 simhash、minhash 来做，第一步的 hash 过程可以使用不同的 functions 来做；
3) 第二次 LSH 把 Signature Matrix 哈希一下，就得到了每个数据点最终被 hash 到了哪个 bucket 里。

本质上为先进行训练(train)然后测试(test/predict)的过程，其中，后期的结果不会对模型产生影响。

LSH 的过程为

相比两两比较，LSH 可以实现再降维+局部寻找匹配对。

例如：

在进行文本相似性检测的时候，进行 word-vector hash 进桶

3.2 基于不同距离表示

通常来说，基于距离的 LSH 设计可以考虑到不同情况下的距离表示能力，从而能够更好度量 embedding 之后的距离，此处的 hash 空间可以采用不同的距离进行表示。

下述分别介绍了不同的距离的相应表示和适用场景，在进行相应的系统构建的时候，如果能够很好的选择相应的距离函数，其 LSH 能够使用更小的编码距离，从而能够缩小存储的空间。

3.2.1 欧式距离

$$d(x,y) := \sqrt{(x_1-y_1)^2+(x_2-y_2)^2+\cdots+(x_n-y_n)^2} = \sqrt{\sum_{i=1}^{n}(x_i-y_i)^2}$$

最为通常的距离表示，因为一个好的降维方法通常考虑到了不同权值的影响，从而进行了维度上的归一化处理，故可以直接采用欧式距离进行距离定义。

3.2.2 曼哈顿距离

可以定义曼哈顿距离的正式意义为 L1-距离或城市区块距离，也就是在欧几里得空间的固定直角坐标系上两点所形成的线段对轴产生的投影的距离总和。例如在平面上，坐标（x1, y1）的点 P1 与坐标（x2, y2）的点 P2 的曼哈顿距离为：

$$d_{12} = \sum_{k=1}^{n} |x_{1k} - x_{2k}|$$

要注意的是，曼哈顿距离依赖座标系统的转度，而非系统在座标轴上的平移或映射。

3.2.3 汉明距离(simhash)

google 对于网页去重使用的是 simhash(hamming LSH)，每天需要处理的文档在亿级别。

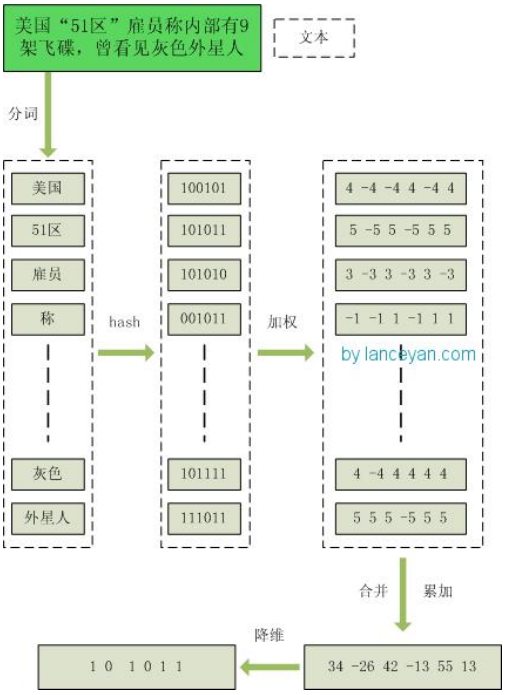


图 3.2 Google Simhash 处理流程实例

由于需要保留相似性约束，相对于普通 hash，先进行词语的 embedding，然后进行加权处理，最后平均归一化，采用的方法是使用 word-embedding 的平均值作为 sentence-embedding 的方法。

1) 处理文本长度

实验表明，simhash(Hamming LSH)用于比较大文本，比如 500 字以上效果都还蛮好，距离小于 3 的基本都是相似，误判率也比较低。但是如果处理的是中等长度的文本，使用 simhash 的效果并不那么理想。

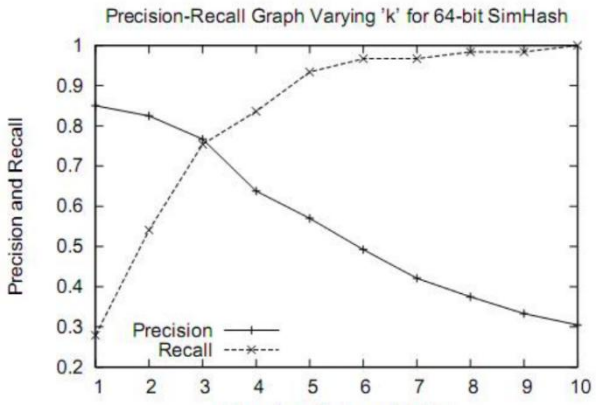


图 3.3 Hamming LSH 处理文本准确度

2) 处理文本速度

目前速度提升了但是数据是不断增量的，如果未来数据发展到一个小时 100w，按现在一次 100ms，一个线程处理一秒钟 10 次，一分钟 60 * 10 次，一个小时 60*10 *60 次 = 36000 次，一天 60*10*60*24 = 864000 次。

目标是一天 100w 次，通过增加两个线程就可以完成。但是如果要一个小时 100w 次呢？则需要增加 30 个线程和相应的硬件资源保证速度能够达到。

线性的算法复杂度对于指数级增长的网络数据集来说降维效果并不友好。

3.2.4 切比雪夫距离

$$D_{\text{Chebyshev}}(p,q) := \max_i(|p_i - q_i|).$$

这也等于以下 Lp 度量的极值：

$$\lim_{k\to\infty}\left(\sum_{i=1}^{n}|p_i - q_i|^k\right)^{1/k},$$

以数学的观点来看，切比雪夫距离是由一致范数（uniform norm）（或称为上确界范数）所衍生的度量，也是超凸度量（injective metric space）的一种。

在最近的复杂网络研究中，基于网络表示学的切比雪夫距离可以进行蛋白质网络中的实体表达。

3.2.5 Jaccard 距离(minhash)

Jaccard Coefficient 用来度量两个集合的相似度，设有两个集合和，它们之间的 Jaccard Coefficient 定义为

$$s = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

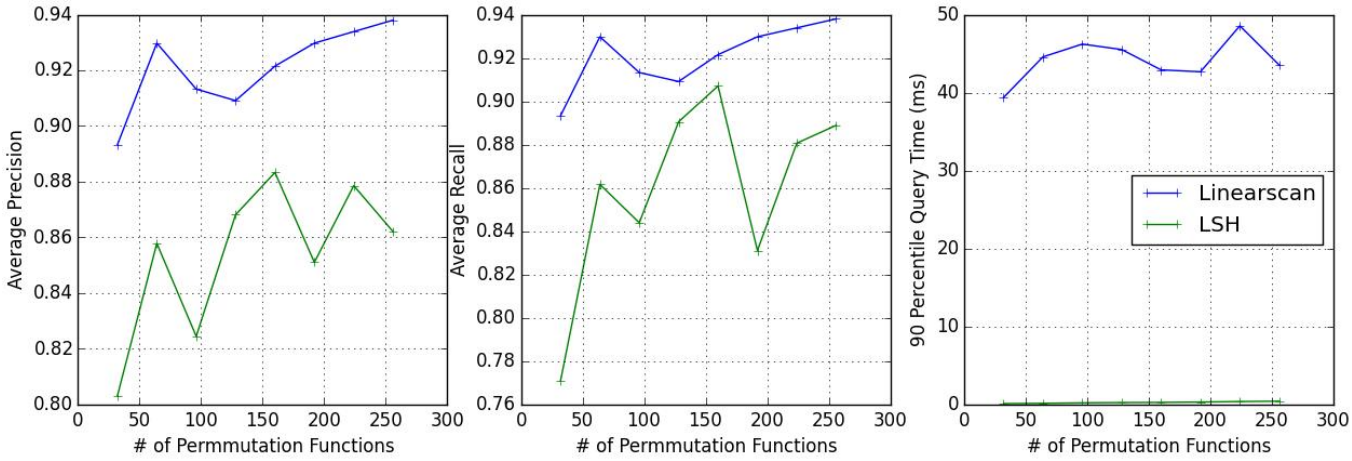


图 3.4 minhash 性能曲线图

3.3 基于 Ensemble 的算法(随机投影森林)

采用多个 Hash 进行定位：利用随机投影树，对所有的数据进行划分，将每次搜索与计算的点的数目减小到一个可接受的范围，然后建立多个随机投影树构成随机投影森林，将森林的综合结果作为最终的结果。

建立一棵随机投影树的过程如下：

1) 随机选取一个从原点出发的向量
2) 与这个向量垂直的直线将平面内的点划分为了两部分
3) 将属于这两部分的点分别划分给左子树和右子树

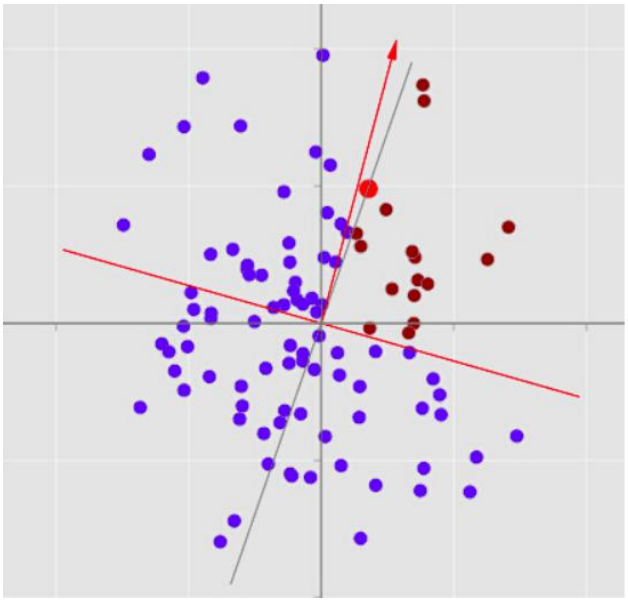


图 3.5 LSH Forest

3.4 机器学习(ML-Based)

在通常的系统中，通过监督学习学习图片的相关表示，如采用 CNN 将图片 Embedding 之后使用 MLP 之后 Softmax 进行输出，此时网络表示 Target-Based 学习。

此处可以考虑将 LSH 和 ML 进行 End-End 组合，同时由于 AutoML 等技术，可以进行自动化的参数调整，从而能够找到性能和开销的拐点，尽可能降低 Embedding 的维度。

f(A)为 CNN 输出的向量表示，可以将其更改为 H(f(A))进一步降维。

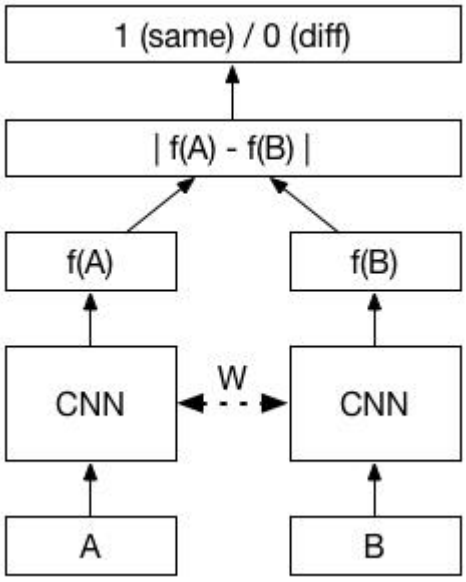


图 3.6 ML-Based LSH 算法

4 LSH 系统设计

4.1 系统理论分析

4.1.1 Simhash 生成(Train)

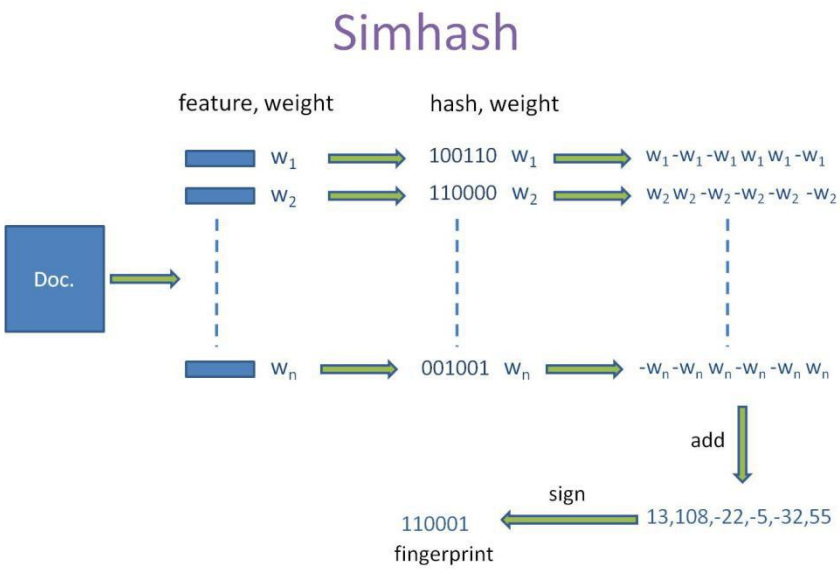


图 4.1 Google Simhash 处理流程

1) 选择 simhash 的位数，请综合考虑存储成本以及数据集的大小，比如说 32 位
将 simhash 的各位初始化为 0
2) 提取原始文本中的特征，一般采用各种分词的方式。比如对于"the cat sat on the mat"，采用两两分词的方式得到如下结果：{"th", "he", "e ", " c", "ca", "at", "t ", " s", "sa", " o", "on", "n ", " t", " m", "ma"}
3) 使用传统的 32 位 hash 函数计算各个 word 的 hashcode，比如："th".hash = -502157718 ，"he".hash = -369049682，……
4) 对各 word 的 hashcode 的每一位，如果该位为 1，则 simhash 相应位的值加 1；否则减 1
5) 对最后得到的 32 位的 simhash，如果该位大于 1，则设为 1；否则设为 0

4.1.2 Simhash 预测(Predict)

1) 遍历数据集

对于 64 位 simhash，海明距离在 3 以内的文本都可以认为是近重复文本，查找待查询文本的 64 位 simhash code 的所有 3 位以内变化的组合，大约需要四万多次的查询

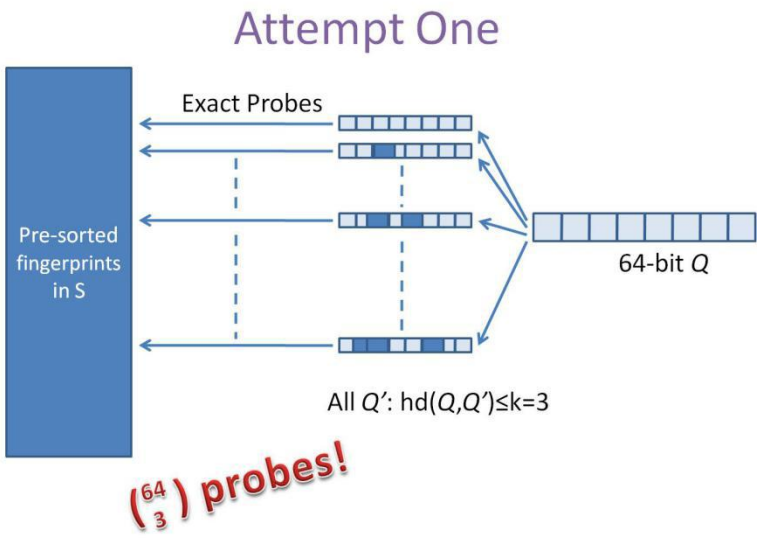


图 4.2 时间遍历算法

时间复杂度高，空间复杂度小

2) 空间预存储

预生成库中所有样本 simhash code 的 3 位变化以内的组合，大约需要占据 4 万多倍的原始空间

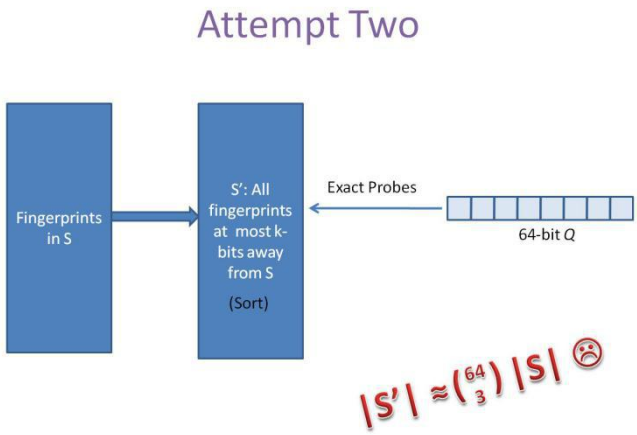


图 4.3 空间预存储算法

4.2 系统 Baseline

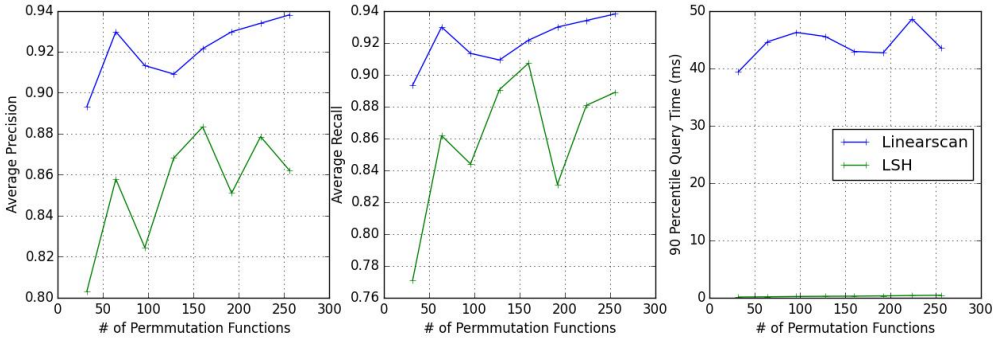
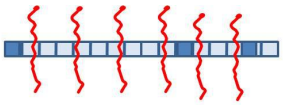


图 4.4 Baseline 系统性能

4.3 改进算法 1-Spilit

要寻找海明距离 3 以内的数值，根据抽屉原理，只要我们将整个 64 位的二进制串划分为 4 块，无论如何，匹配的两个 simhash code 之间至少有一块区域是完全相同的，如下图所示：



- *f*-bits split into *r* pieces
- $\binom{r}{k}$ tables with permuted fingerprints
- Exact search on *f(1-k/r)* bits
- With 2^d existing fingerprints, each probe yields $2^{d-f(1-k/r)}$ fingerprints

f=64,k=3,d=34

r	#tables	#match per probe
4	4	2^{18}
5	10	2^6
6	20	8

图 4.5 改进算法

由于无法事先得知完全相同的是哪一块区域，因此必须采用存储多份 table 的方式。在本例的情况下，需要存储 4 份 table，并将 64 位的 simhash code 等分成 4 份；

对于每一个输入的 code，通过精确匹配的方式，查找前 16 位相同的记录作为候选记录。

1) 将 64 位的二进制串等分成四块

2) 调整上述 64 位二进制，将任意一块作为前 16 位，总共有四种组合，生成四份 table

3) 采用精确匹配的方式查找前 16 位

4) 如果样本库中存有 2^34（差不多 10 亿）的哈希指纹，则每个 table 返回 2^(34-16)=262144 个候选结果，大大减少了海明距离的计算成本

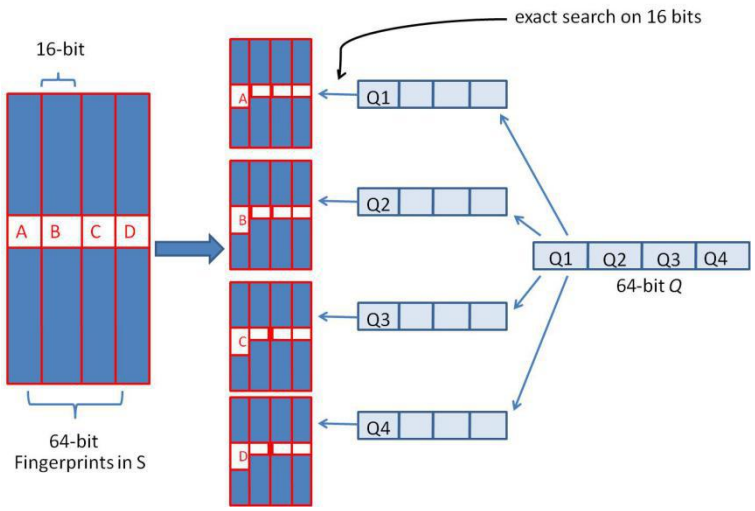
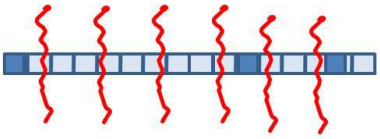


图 4.6 table 预存储

将 f-bits 的 hash code 进行划分 r pieces



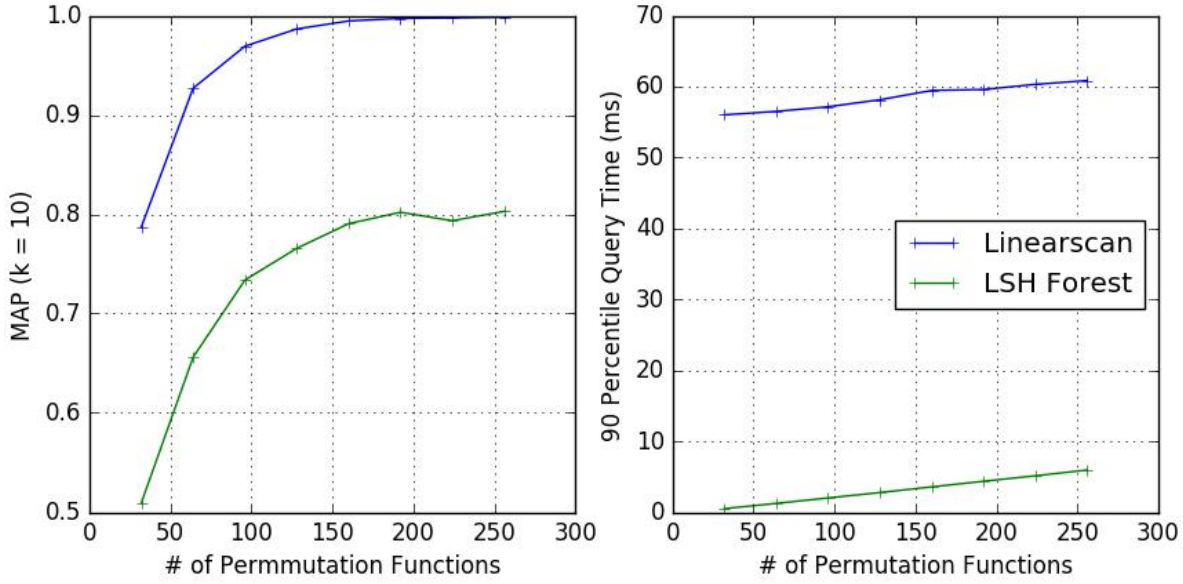
- *f*-bits split into *r* pieces
- $\binom{r}{k}$ tables with permuted fingerprints
- Exact search on *f(1-k/r)* bits
- With 2^d existing fingerprints, each probe yields $2^{d-f(1-k/r)}$ fingerprints

f=64,k=3,d=34

r	#tables	#match per probe
4	4	2^{18}
5	10	2^6
6	20	8

图 4.7 table 预存储对应

4.4 改进算法 2-采用 Forest for Top-K



MinHash LSH 对半径（或阈值）查询很有用。 但是，在某些情况下，top-k 查询通常更有用。是一种通用的 LSH 数据结构，它使 top-k 查询可用于许多不同类型的 LSH 索引，包括 MinHash LSH。 利用前缀树进行 forest 聚合。可调节参数为树的深度和数量。

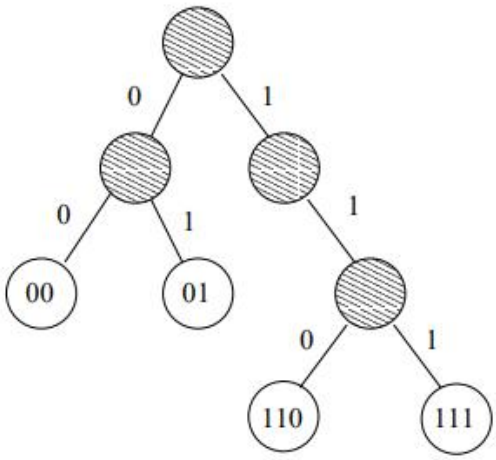
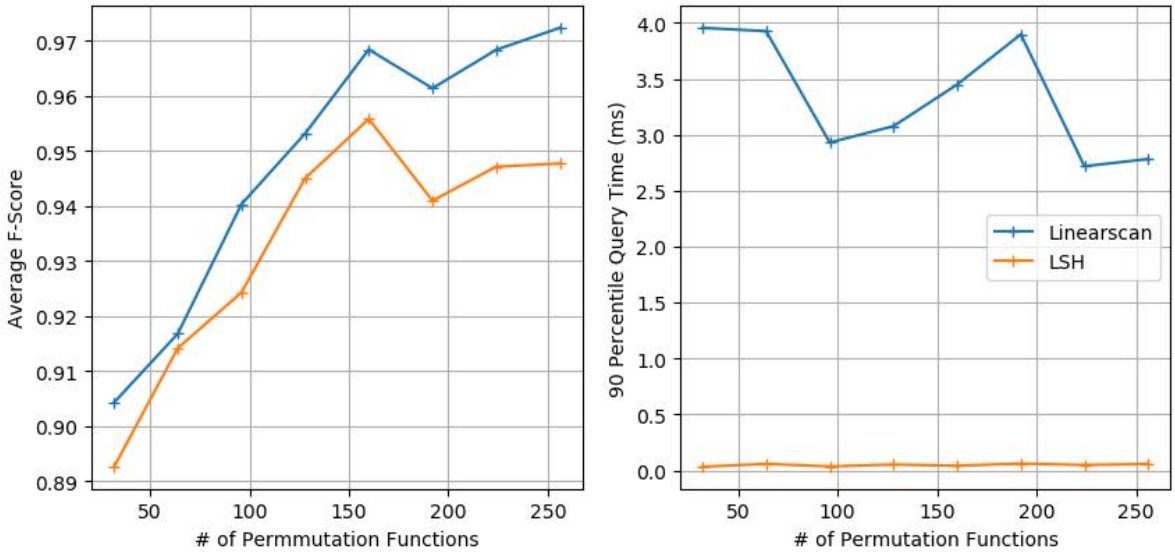


图 4.8 LSH 前缀树

利用列斯与哈夫曼树的前缀树进行查询，从而将查询性能从 O(N)， k 为桶中样本数，变成

$$O(l * log_B N)$$

4.5 改进算法 3-采用 Ensemble



与 MinHash LSH 类似，包含搜索的 LSH 索引,给定查询集，查找包含高于特定阈值的集合而非采用 Forest 进行 Bagging 结果聚合。

$$Containment = \frac{|Q \cap X|}{|Q|}$$

假设有一大堆集合，给定一个查询，也是一个集合，你想在你的集合中找到有交集的集合查询高于某个阈值。我们可以将查询视为集合 Q，并将集合中的任意集合视为 X.由于此处的查询集是固定的（对于特定的搜索问题），分母|Q|是一个常数，交叉点大小完全由包含确定。因此，我们可以搜索集合中具有高于特定阈值的集合的集合，并且可以通过从交集阈值轻松推导出包含阈值|Q|。

简而言之，由于通常的数据或者网络为无标度，其访问频率为幂率分布，从而能够加入访问频次信息，统计频次提供召回概率，从而优化算法性能。

Symbol	Description
X, Q	Indexed domain, Query domain
x, q	Domain size of X, Domain size of Q
b, r	Number of bands in LSH, Number of hash functions in each band
$s(Q,X)$, $t(Q,X)$	Jaccard similarity and containment score between sets Q and X
s^*, t^*	Jaccard similarity and containment thresholds
$\hat{s}_{x,q}(t)$, $\hat{t}_{x,q}(s)$	functions that convert $t(Q,X)$ to $s(Q,X)$ and vice versa, given the domain sizes $x = \|X\|$ and $q = \|Q\|$
FP, FN	False positive & negative probabilities
N^{FP}	Count of false positive domains
l, u	Lower, Upper bound of a partition
m	Num. hash functions in MinHash
n	Num. of partitions in LSH Ensemble

给定两个集合的相似度定义：

$$P(h_{\min}(X) = h_{\min}(Y)) = s(X,Y)$$

给定查询的概率设定：

$$P(s|b,r) = 1 - (1 - s^r)^b$$

给定相应的相似度和阈值估计：

$$\hat{s}_{x,q}(t) = \frac{t}{\frac{x}{q} + 1 - t},\ \hat{t}_{x,q}(s) = \frac{(\frac{x}{q} + 1)s}{1 + s}$$

给定最终的相似度定义：

$$s^* = \hat{s}_{u,q}(t^*) = \frac{t^*}{\frac{u}{q} + 1 - t^*}$$

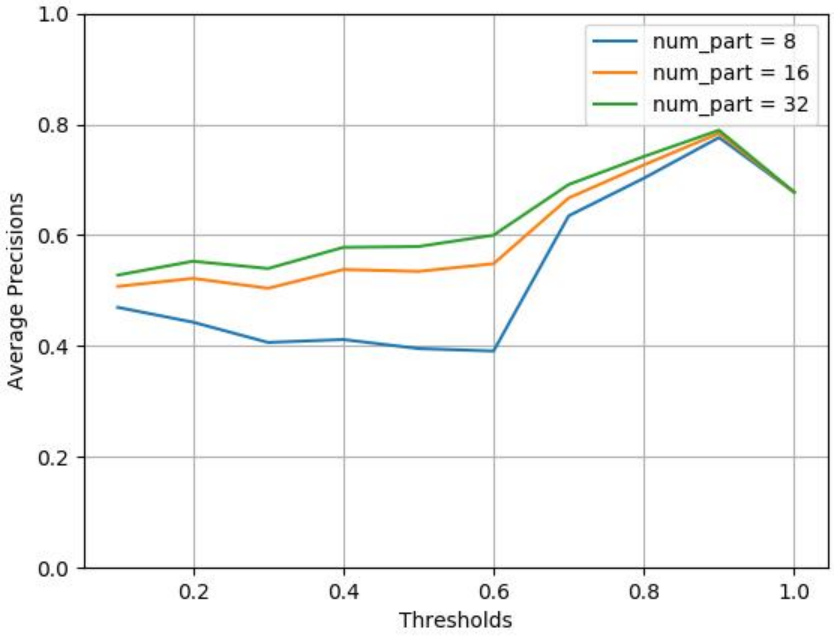


图 4.9 Ensemble Avg Precisions

5 基于 LSH 的推荐系统设计

此处使用 NIPS-Paper 文档数据集进行相应的基于 LSH 的推荐系统设计和测试。

5.1 系统构建

5.1.1 Shingles

Shingles 是一个非常基本的广泛概念。 对于文本，它们可以是字符，unigrams 或 bigrams。 可以使用一组类别作为 Shingles。Shingles 只是将我们的文档简化为元素集，以便我们可以计算集合之间的相似性。

对于论文推荐，我们将使用从论文标题和摘要中提取的 unigrams。 如果我们有关于一组用户喜欢哪些论文的数据。

5.1.2 词袋分词

	paper 1	paper 2	paper 3	...	paper n
word 1	1	0	1	...	0
word 2	0	0	0	...	1
word 3	0	1	1	...	1
...					
word n	1	0	1	...	0

图 5.1 词袋分词

使用 word-bag 作为初始 hash,不考虑词语之间的序列关系。

5.1.3 构建 LSH Forest

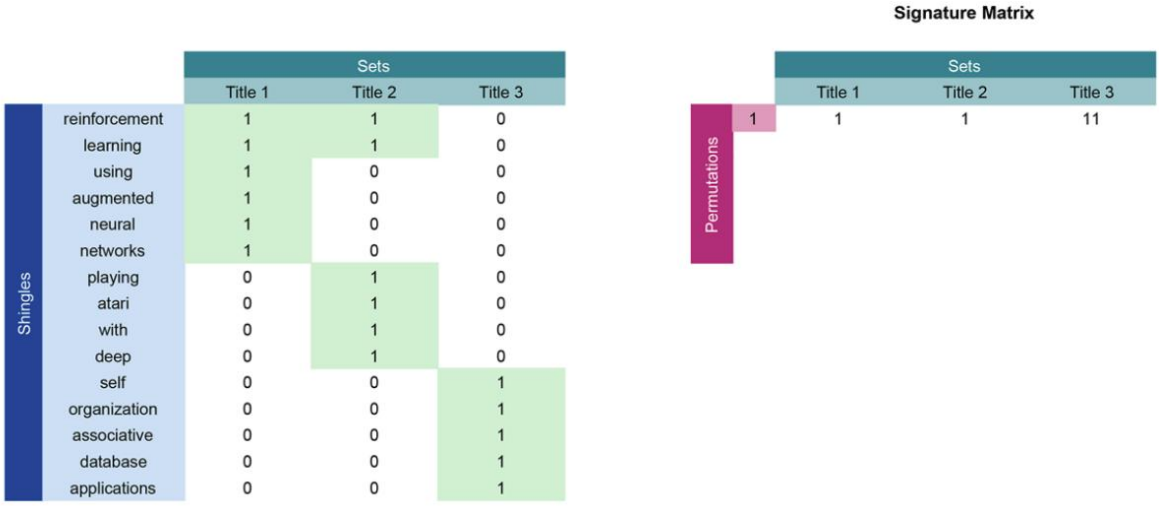


图 5.2 LSH-1

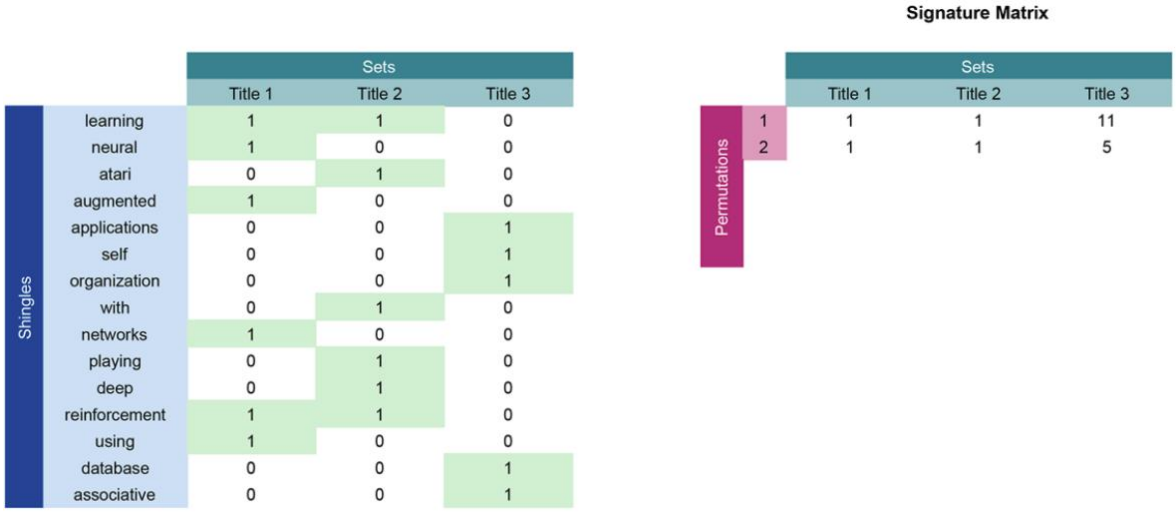


图 5.3 LSH-2

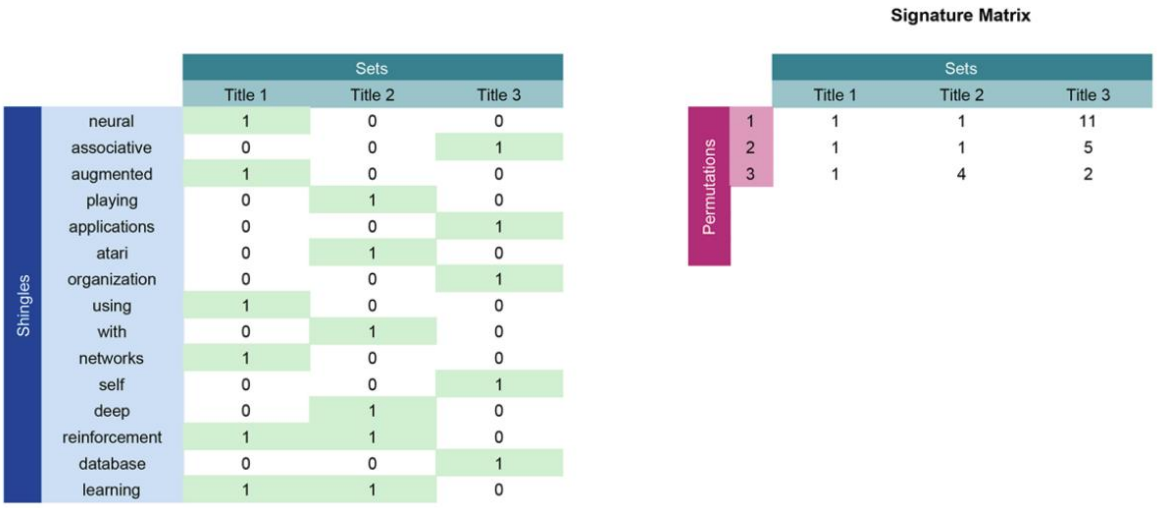


图 5.4 LSH-3

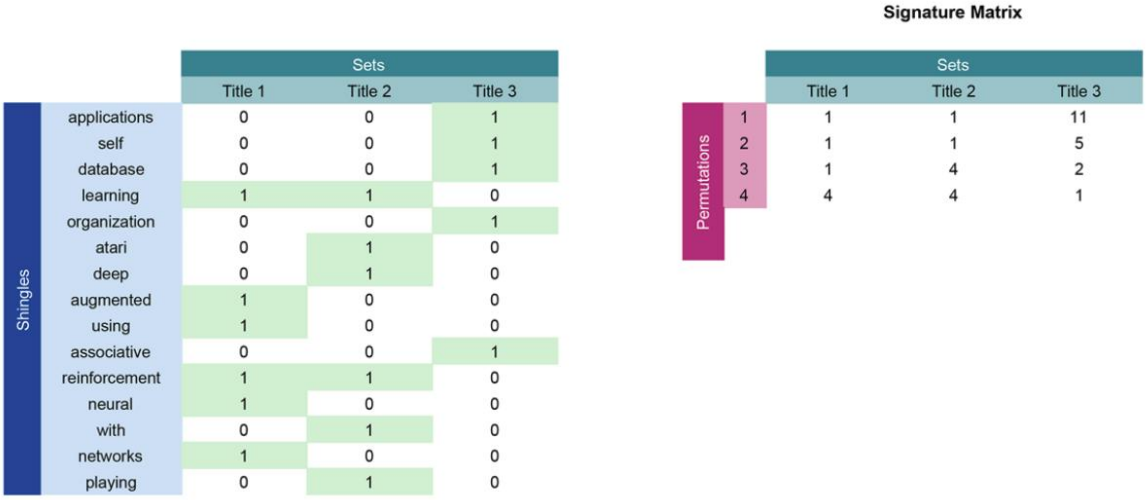


图 5.5 LSH-4

5.2 系统构建时间测试

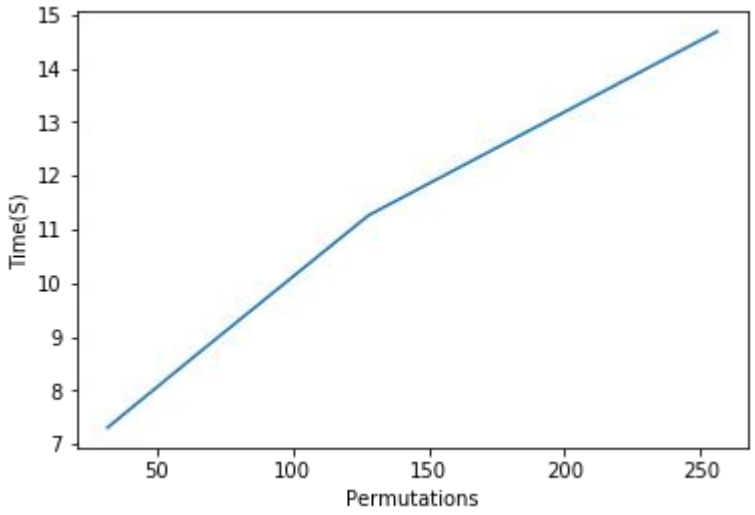


图 5.6 系统预测构建时间测试图(Forest)

可以看到系统的构建时间基本为线性

5.3 系统构建空间测试

由于 python 自带的 LSH 无法获知准确的 object 大小，采用观察 python 线程 memory 进行内存占用估算，其空间占与理论相符。

因为 jupyter notebook 的自身内存管理原因，不容易

将 jupyter notebook 中代码整合进入 python 脚本，利用 memory_profiler 进行装饰器装饰需要检查的函数

代码 5.1 检查内存占用

```
@profile
def get_forest(data, perms):
    start_time = time.time()

    minhash = []

    for text in data['text']:
        tokens = preprocess(text)
        m = MinHash(num_perm=perms)
        for s in tokens:
            m.update(s.encode('utf8'))
        minhash.append(m)

    forest = MinHashLSHForest(num_perm=perms)

    for i,m in enumerate(minhash):
        forest.add(i,m)

    forest.index()

    print('It took %s seconds to build forest.' %(time.time()-start_time))
```

```
    return forest,(time.time()-start_time)
```

```
hover@wings ~/Desktop/Labs/HUST_IOTStorage_Labs/LSH/src master ● python -m memory_profiler test.py
It took 118.52210640907288 seconds to build forest.
(995     Neural Network Weight Matrix Synthesis Using O...
5       Using a neural net to instantiate a deformable...
6413    Non-Intrusive Gaze Tracking Using Artificial N...
3056    Proximity Effect Corrections in Electron Beam ...
2457    Inferring Neural Firing Rates from Spike Train...
Name: title, dtype: object, 0.004341602325439453)
Filename: test.py

Line #    Mem usage    Increment   Line Contents
================================================
    20  283.684 MiB  283.684 MiB   @profile
    21                             def get_forest(data, perms):
    22  283.684 MiB    0.000 MiB       start_time = time.time()
    23
    24  283.684 MiB    0.000 MiB       minhash = []
    25
    26  292.016 MiB    0.000 MiB       for text in data['text']:
    27  292.016 MiB    0.258 MiB           tokens = preprocess(text)
    28  292.016 MiB    0.391 MiB           m = MinHash(num_perm=perms)
    29  292.016 MiB    0.051 MiB           for s in tokens:
    30  292.016 MiB    0.258 MiB               m.update(s.encode('utf8'))
    31  292.016 MiB    0.258 MiB           minhash.append(m)
    32
    33  292.016 MiB    0.000 MiB       forest = MinHashLSHForest(num_perm=perms)
    34
    35  304.383 MiB    0.020 MiB       for i,m in enumerate(minhash):
    36  304.383 MiB    0.512 MiB           forest.add(i,m)
    37
    38  304.383 MiB    0.000 MiB       forest.index()
    39
    40  304.383 MiB    0.000 MiB       print('It took %s seconds to build forest.' %(time.time()-start_time))
    41
    42  304.383 MiB    0.000 MiB       return forest,(time.time()-start_time)
```

图 5.7 系统空间占用检测(k=32)

则其中 forest 占用 0.512MB。

```
It took 114.87854027748108 seconds to build forest.
(995     Neural Network Weight Matrix Synthesis Using O...
5       Using a neural net to instantiate a deformable...
5191    A Self-Organizing Integrated Segmentation and ...
2069    Analytic Solutions to the Formation of Feature...
2457    Inferring Neural Firing Rates from Spike Train...
Name: title, dtype: object, 0.003495931625366211)
Filename: test.py

Line #    Mem usage    Increment   Line Contents
================================================
    20  283.512 MiB  283.512 MiB   @profile
    21                             def get_forest(data, perms):
    22  283.512 MiB    0.000 MiB       start_time = time.time()
    23
    24  283.512 MiB    0.000 MiB       minhash = []
    25
    26  296.391 MiB    0.000 MiB       for text in data['text']:
    27  296.391 MiB    0.258 MiB           tokens = preprocess(text)
    28  296.391 MiB    0.348 MiB           m = MinHash(num_perm=perms)
    29  296.391 MiB    0.012 MiB           for s in tokens:
    30  296.391 MiB    0.258 MiB               m.update(s.encode('utf8'))
    31  296.391 MiB    0.258 MiB           minhash.append(m)
    32
    33  296.391 MiB    0.000 MiB       forest = MinHashLSHForest(num_perm=perms)
    34
    35  310.902 MiB    0.250 MiB       for i,m in enumerate(minhash):
    36  310.902 MiB    0.516 MiB           forest.add(i,m)
    37
    38  310.902 MiB    0.000 MiB       forest.index()
    39
    40  310.902 MiB    0.000 MiB       print('It took %s seconds to build forest.' %(time.time()-start_time))
    41
    42  310.902 MiB    0.000 MiB       return forest,(time.time()-start_time)
```

图 5.8 系统空间占用检测(k=64)

则其中 forest 占用 0.512MB。

```
hover@wings ~/Desktop/Labs/HUST_IOTStorage_Labs/LSH/src master ● python -m memory_profiler test.py
It took 106.04870462417603 seconds to build forest.
(995     Neural Network Weight Matrix Synthesis Using O...
5       Using a neural net to instantiate a deformable...
5191    A Self-Organizing Integrated Segmentation and ...
2069    Analytic Solutions to the Formation of Feature...
2457    Inferring Neural Firing Rates from Spike Train...
Name: title, dtype: object, 0.008367061614990234)
Filename: test.py

Line #    Mem usage    Increment   Line Contents
================================================
    20  283.273 MiB  283.273 MiB   @profile
    21                             def get_forest(data, perms):
    22  283.273 MiB    0.000 MiB       start_time = time.time()
    23
    24  283.273 MiB    0.000 MiB       minhash = []
    25
    26  306.645 MiB    0.000 MiB       for text in data['text']:
    27  306.645 MiB    0.258 MiB           tokens = preprocess(text)
    28  306.645 MiB    0.266 MiB           m = MinHash(num_perm=perms)
    29  306.645 MiB    0.023 MiB           for s in tokens:
    30  306.645 MiB    0.258 MiB               m.update(s.encode('utf8'))
    31  306.645 MiB    0.000 MiB           minhash.append(m)
    32
    33  306.645 MiB    0.000 MiB       forest = MinHashLSHForest(num_perm=perms)
    34
    35  324.152 MiB    0.250 MiB       for i,m in enumerate(minhash):
    36  324.152 MiB    1.125 MiB           forest.add(i,m)
    37
    38  324.152 MiB    0.000 MiB       forest.index()
    39
    40  324.152 MiB    0.000 MiB       print('It took %s seconds to build forest.' %(time.time()-start_time))
    41
    42  324.152 MiB    0.000 MiB       return forest,(time.time()-start_time)
```

图 5.9 系统空间占用检测(k=128)

则其中 forest 占用 1.125MB。

由于机器字长的限定，所以空间占用并非严格线性，而是以 64 为单位进行增长。

5.4 系统预测时间测试

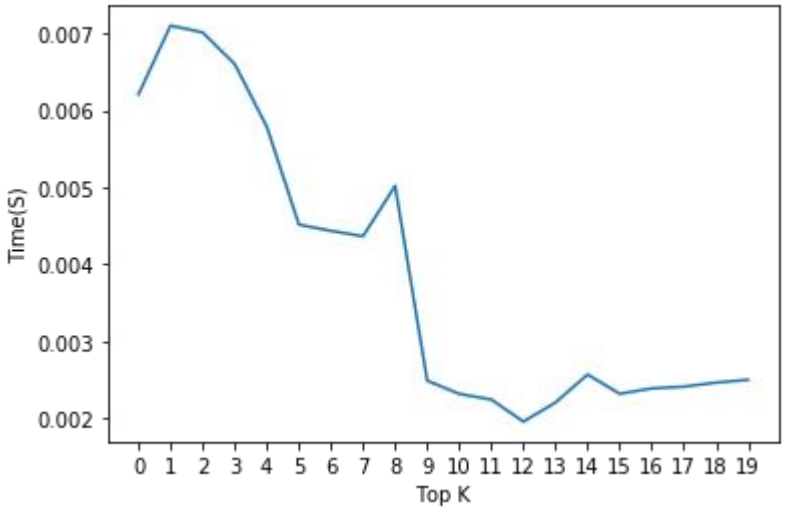


图 5.10 系统预测 temp 延迟时间测试图(Top-K)

在进行系统预测的时候，对于 TOP-K 的预测结果出现了如图的变化，可以有两点原因：

1) 系统进行了一定的缓存，导致载入内存的时间变快。

2) 由于粒度更粗，不需要多次的聚合查询，从而速度更快。

5.5 系统准确度测试

由于系统若需要遍历求出准确最邻近开销过大 O(N^2)，故此处使用 FP 进行准确度测试。

其中 N 为 LSH 的维度，K 为推荐(最邻近)的个数。

代码 5.2 FP 计算

```
fp_rate_list=[]
permutations=128
num_recommendations=3
forest,build_time = get_forest(db, permutations)
for id in db['id'].values:
    print(id)
    title = get_title(id).values[0]
    result,restime = predict(title, db, permutations, num_recommendations, fo
rest)

    right=get_topic_by_id(id)
    fp=0
    total=num_recommendations
    for i in list(result):
        temp=get_topic_by_title(i)
        if right!=temp:
            fp+=1
    fp_rate=fp/total
    fp_rate_list.append(fp_rate)
```

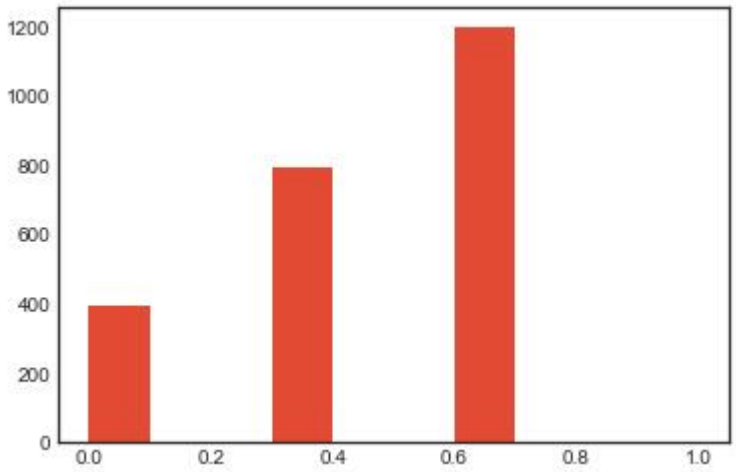


图 5.11 FP(k=3,N=128)

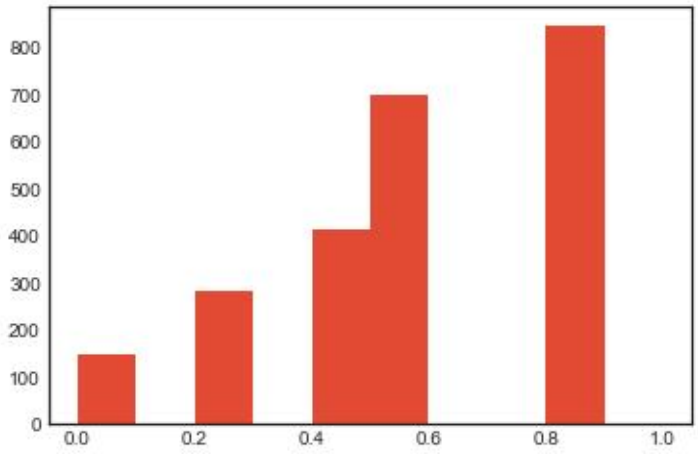


图 5.12 FP(k=5,N=128)

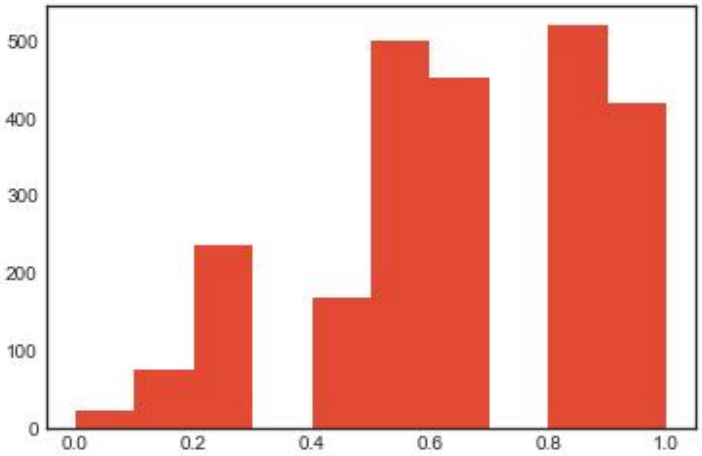


图 5.13 FP(k=10,N=128)

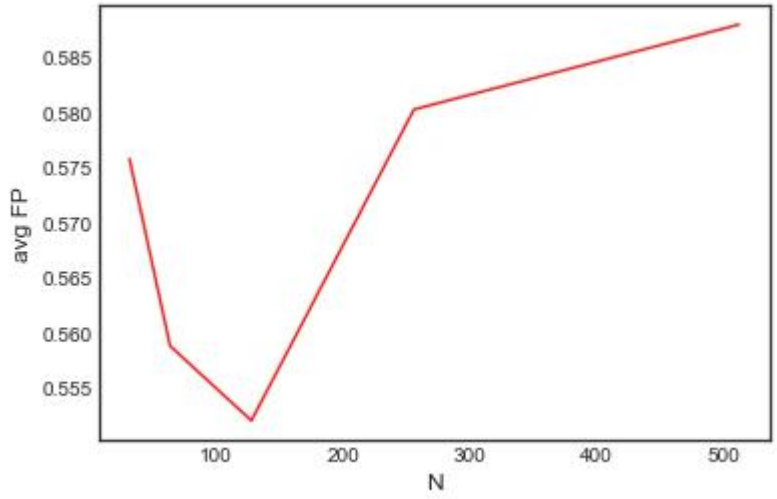


图 5.14 FP(k= 5 ,N=32,64,128,256,512)

可以看到，在提高 K 值之后，系统的 FP(False-Positive)有了先下降后上升，整体性能先上升后下降，体现了 LSH 的局部性特征。

在小样本的情况下，LSH 的准确度基本维持不变，由于 k 的上升，整体 FP 下降。随着 K 值的增大，局部性特征体现，embedding 不能很好的区分样本，从而 FP 升高，系统性能下降。

5.6 基于主题进行聚类后再 Hash

使用 NMF，LDA 等算法进行主题聚类，之后再进行 Hash，类似于层次聚类，在较高的层次采用具有语义的聚类算法，而在较低的维度采用无语义的 LSH 进行比较。从而利用系统进行预测的时候先选择 Topic，再进行 LSH 从而能够大大的缩小 LSH 的存储空间，更好的利用局部性。

代码 5.3 LDA 主题模型

```
n_features = 1000
n_topics = 8
n_top_words = 10



def print_top_words(model, feature_names, n_top_words):
    for topic_idx, topic in enumerate(model.components_):
        print("\nTopic #%d:" % topic_idx)
        print(" ".join([feature_names[i]
                        for i in topic.argsort()[:-n_top_words - 1:-1]]))
    print()



tfidf_vectorizer = TfidfVectorizer(max_df=0.95, min_df=2,max_features=n_featu
res,stop_words='english')

tfidf = tfidf_vectorizer.fit_transform(df['paper_text'])



nmf = NMF(n_components=n_topics, random_state=0,alpha=.1, l1_ratio=.5).fit(tf
idf)

print("Topics found via NMF:")
```

```
tfidf_feature_names = tfidf_vectorizer.get_feature_names()

print_top_words(nmf, tfidf_feature_names, n_top_words)
```

```
Topics found via NMF:

Topic #0:
algorithm matrix convex theorem bound log loss problem optimization function

Topic #1:
network networks units layer training neural input hidden output learning

Topic #2:
policy state action reward agent actions reinforcement learning policies states

Topic #3:
model data models distribution posterior latent bayesian inference gaussian likelihood

Topic #4:
image images object visual features objects model feature recognition pixel

Topic #5:
neurons spike neuron synaptic stimulus firing cells activity cell time

Topic #6:
graph tree node nodes clustering graphs cluster algorithm clusters edges

Topic #7:
kernel data training classification svm learning kernels xi classifier feature
```

图 5.15 获取数据集的 Topic

```
In [109]: tsne = TSNE(random_state=3211)
          tsne_embedding = tsne.fit_transform(nmf_embedding)
          tsne_embedding = pd.DataFrame(tsne_embedding,columns=['x','y'])
          tsne_embedding['hue'] = nmf_embedding.argmax(axis=1)

In [112]: tsne_embedding
```

Out[112]:

	x	y	hue
0	-9.190134	35.413063	4
1	51.162403	34.995338	5
2	44.959511	52.741528	5
3	-3.304435	2.795137	3
4	28.577814	-0.880817	1
5	-0.073279	26.227720	4
6	44.570671	9.655113	1
7	55.025757	16.565563	1
8	57.276543	8.895735	1
9	55.319099	13.732885	1
10	51.395344	6.628024	1
11	-58.266460	-46.929237	6
12	67.768967	-0.389335	1
13	56.018024	-5.948046	1
14	12.298813	19.958035	5
15	17.176563	-6.983132	7
16	-59.586040	-24.148685	6
17	51.711693	43.382500	5
18	30.186480	40.845325	5
19	57.912445	4.758279	1

图 5.16 利用 t-SNE 进行降维划分

代码 5.4 tSNE 降维划分

```
tsne = TSNE(random_state=3211)

tsne_embedding = tsne.fit_transform(nmf_embedding)

tsne_embedding = pd.DataFrame(tsne_embedding,columns=['x','y'])

tsne_embedding['hue'] = nmf_embedding.argmax(axis=1)
```

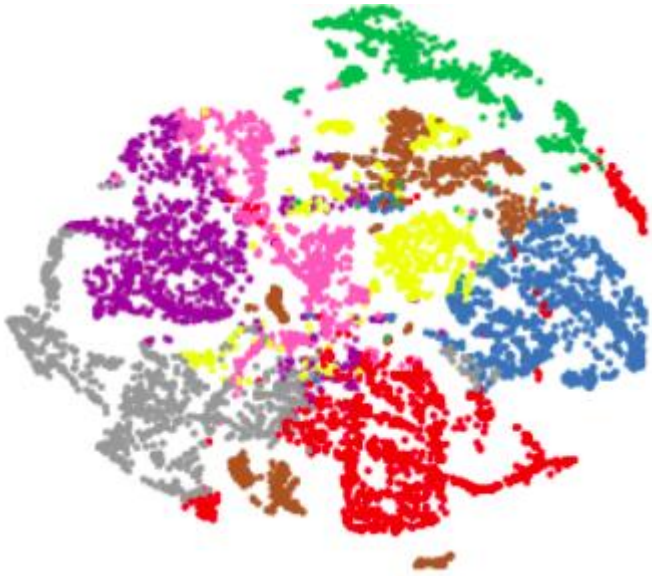


图 5.17 主题可视化

```
In [334]: forest_list=[]
          x=[]
          y=[]
          for i in range(0,7):
              nowdb=db.iloc[l[i]]
              print(nowdb.shape[0])
              x.append(nowdb.shape[0])
              forest,build_time = get_forest(nowdb, permutations)
              y.append(build_time)
              forest_list.append(forest)

          877
          It took 2.034874439239502 seconds to build forest.
          553
          It took 1.9230129718780518 seconds to build forest.
          1052
          It took 3.5753085613250732 seconds to build forest.
          683
          It took 2.285890817642212 seconds to build forest.
          736
          It took 2.0455105304718018 seconds to build forest.
          677
          It took 2.5237605571746826 seconds to build forest.
          839
          It took 2.693091630935669 seconds to build forest.
```

图 5.18 分主题构建 LSH

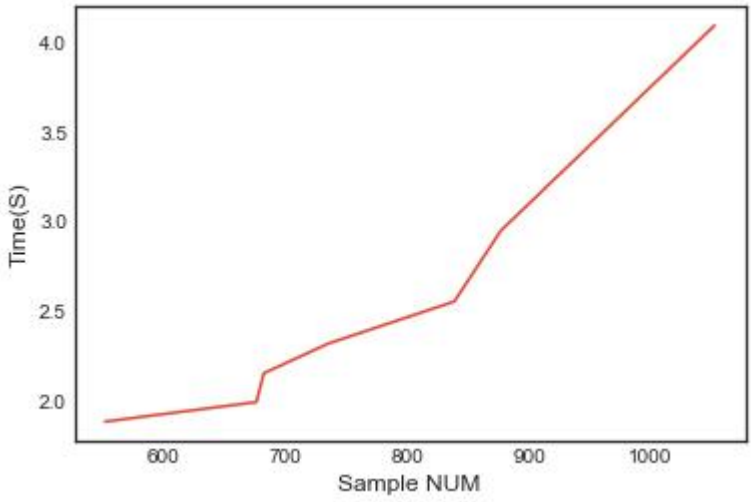


图 5.19 Sample 数量与构建时间关系

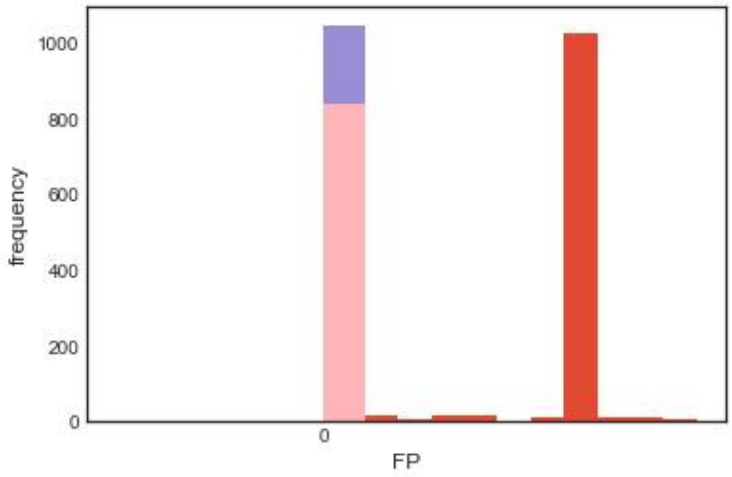


图 5.20 FP 的频率分布(N=128)

可以看到 Hierachy-LSH FP(K=128)相较于 Baseline(N=512) FP 有了明显的降低，此时可以有效的降低维度，同时提升性能。

5.7 基于社群进行聚类后再 Hash

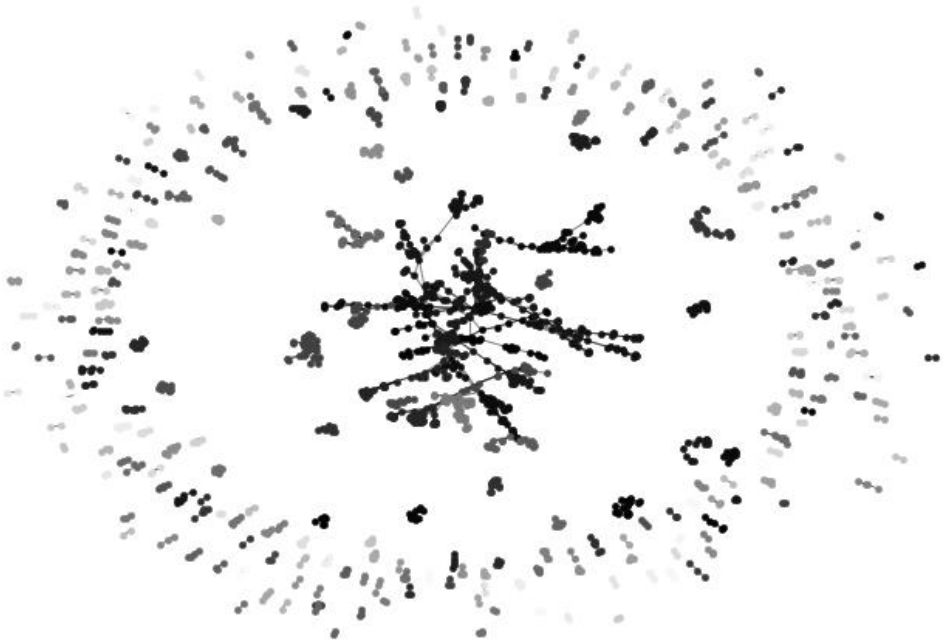


图 5.21 Community 可视化

同样，采用在先进行 Community 聚类，后进行 LSH，过程如上，此处不再赘述。

6 研究总结

因为之前工作的主要集中在网络表示学习，对于常规的降维方法和 embedding 算法有一定了解，故进行 LSH 相应探索的时候结合已有知识(复杂网络，搜索引擎，推荐系统)等，并考虑实际的应用场景，完成 LSH 存储体积优化。

通常来说，在进行 LSH 系统构建的时候，使用 LSH 的顺序为先 LSH 进行粗选，然后使用相应的精确算法(如推荐系统中的序列建模，搜索引擎中的图关联分析)进行候选集的召回，但此时为了候选集的精确度，需要占用一定的系统性能和存储空间，正如同上文所分析的，在指数级增长的网络数据面前，需要维持一定的精度，其性能开销是线性的，为了减少系统开销和提升性能就需要其他的解决途径。

在不同的领域有不同的解决方案，通常来说的思路有

1) 基于特征
   a) 图像领域：针对特定的特征进行建模，如模式匹配点，图片利用 CNN 进行 embedding，获取了第一层 hash 之后再进行 hash 的策略
   b) 文本领域：word-bag 模型，Transformer(Pos-Based) Matrix
   c) 网络领域：Deep-walk, Aggregator
2) 基于模型
   a) Split:对于 Embedding 内部进行划分
   b) Forest:采用多个弱分类器进行 Bagging
   c) ML-Based:与 ML 模型一起组成 End-End 模型

本文采用了层次的 LSH，充分考虑了机械相似性和语义相似性，相较于传统的 Forest，加入了相应的语义信息，能够有效的提升性能。

实验过程中，采用 NIPS-PAPER 数据集进行测试，以及多种算法的比较分析，同时给出了一般情况下的系统建模 pipeline。

参考文献

[1] http://web.mit.edu/andoni/www/LSH/

[2] "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.

[3] 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.

[4] 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.

[5] 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

[6] LSH Forest: Self-Tuning Indexes for Similarity Search