# Problem I: Locality-Sensitive Hashing

*1452669, Yang LI, April 7*

## Data Overview & Propressing

Read data as DataFrame, which indexed by pluno and columned by vipno, basic information shows as follows:

```
1  <class 'pandas.core.frame.DataFrame'>
2  Index: 2635 entries, 10000004 to 40000700
3  Columns: 298 entries, 13205496418 to 6222021615015662822
4  dtypes: float64(298)
5  memory usage: 6.0+ MB
```

To have a basic impression by Data Visualization:

| Overview of Trade Data | Distribution of Trade Data Amount |
|---|---|
|  |  |
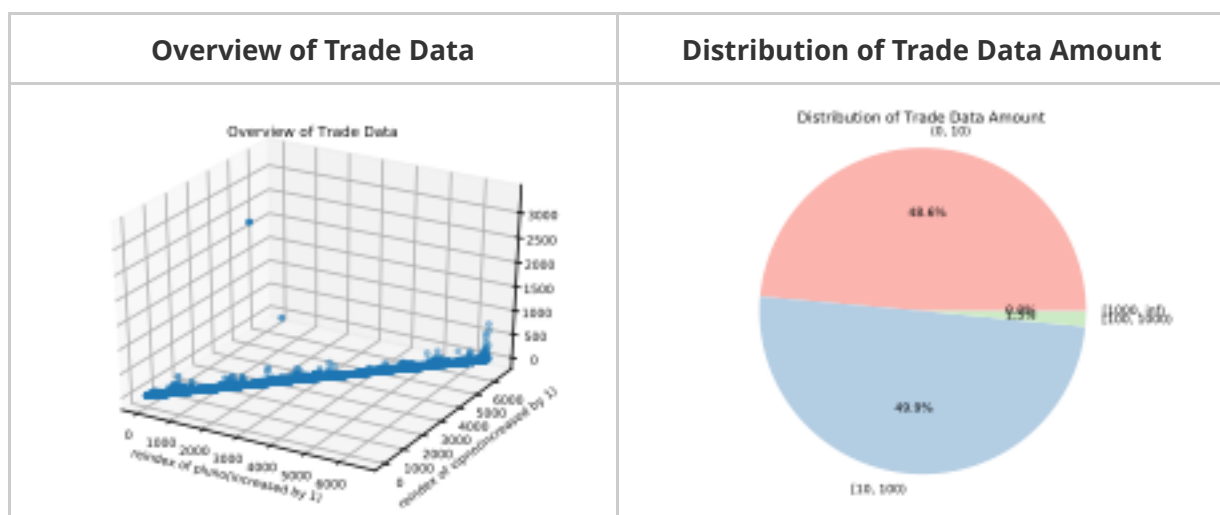
As we can see, it have 2 points above 1k, which lists as follows:

| vipno | pluno | amt |
|---|---|---|
| 2900001356947 | 14842010 | 3303.0 |
| 2900000350175 | 15114031 | 1334.0 |

Thus, the distribution of the data can be represented by the Pie Chart as above (excluding zero).

## Near Neighbor using LSH

Since I use Euclidean Space, following is a brief introduction of Locality-Sensitive Hashing using s-stable distributions.

Let $f_s(t)$ denote the probability density funtion of the absolute value of the s-stable distribution. For the two vectors $p, q$, let $u = ||p - q||_s$ and let $p(u)$ denote the probility (as a function of $u$) that $p, q$ collide for a hash function uniformly chosen from family. For a radom vector $a$ whose entries are drawn from from s-stable distribution, $a.p - a.q$ is distributed as $cX$ where $X$ is a random variable drawn from a s-stable distribution. Since b is drawn uniformly from $[0, w]$ it is easy to see that

$$p(u) = Pr_{a,b}[h_{a,b}(p) = h_{a,b}(q)] = \int_0^w \frac{1}{u} f_s(\frac{t}{u})(1 - \frac{t}{w})dt$$

In practice, all $L$ hash tables use the same primary hash function $t_1$ (used to dermine the index in the hash table) and the same secondary hash function $t_2$. These two hash functions have the form

$$t_1(a_1, a_2, \ldots, a_k) = ((\Sigma_{i=1}^k r'_i a_i) \bmod P) \bmod \text{tablesize}$$

$$t_2(a_1, a_2, \ldots, a_k) = (\Sigma_{i=1}^k r''_i a_i) \bmod P)$$

where $r'$ and $r''$ are random integers, tablesize is the size of the hash tables, and $P$ is a prime.

In the current usage of the LSHash module, I use `sample` to generate random vipno. Since it is a Approximate Near Neighbor, I remove the generated random vipno carefully (rather than remove the first), and the whole code shows as follows:

```
def knn(df, k, coefficient):
    hash_size = int(coefficient * df.shape[1])
    lsh = LSHash(hash_size, input_dim=df.shape[0])
    for vipno in df:
        lsh.index(df[vipno], extra_data=vipno)
    random_column = df[df.columns.to_series().sample(1)]
    random_vip = random_column.columns.values[0]
    logging.info('random vipno: {}'.format(random_vip))
    res = lsh.query(random_column.values.flatten())[0: k + 1]
    print('vipno in ranked order using kNN(k = {}):'.format(k))
    knns = []
    for item in res:
        if item[0][1] != random_vip:
            print(item[0][1])
            knns.append(item[0][1])
    return random_vip, knns[:5]
```

# Performance

Here will analyze the performance both in theory and practice.

**Time & Space Complexity in Theory**

Given the parameters $k$ and $L$, the algorithm has the following performance guarantees:

- preprocessing time: $O(nLkt)$, where $t$ is the time to evaluate a function $h \in \mathcal{F}$ on an input point $p$.
- space: $O(nL)$, plus the space for storing data points.

- query time: $O(L(kt + dnP_2^k)^L)$

**Benchmark in Practice**

As the statistics shows, most time cost in file I/O. In the algorithms part, `index` and `query` cost almost all time, since number of functions $L$ is constant, it takes linear time, which is acceptedable.
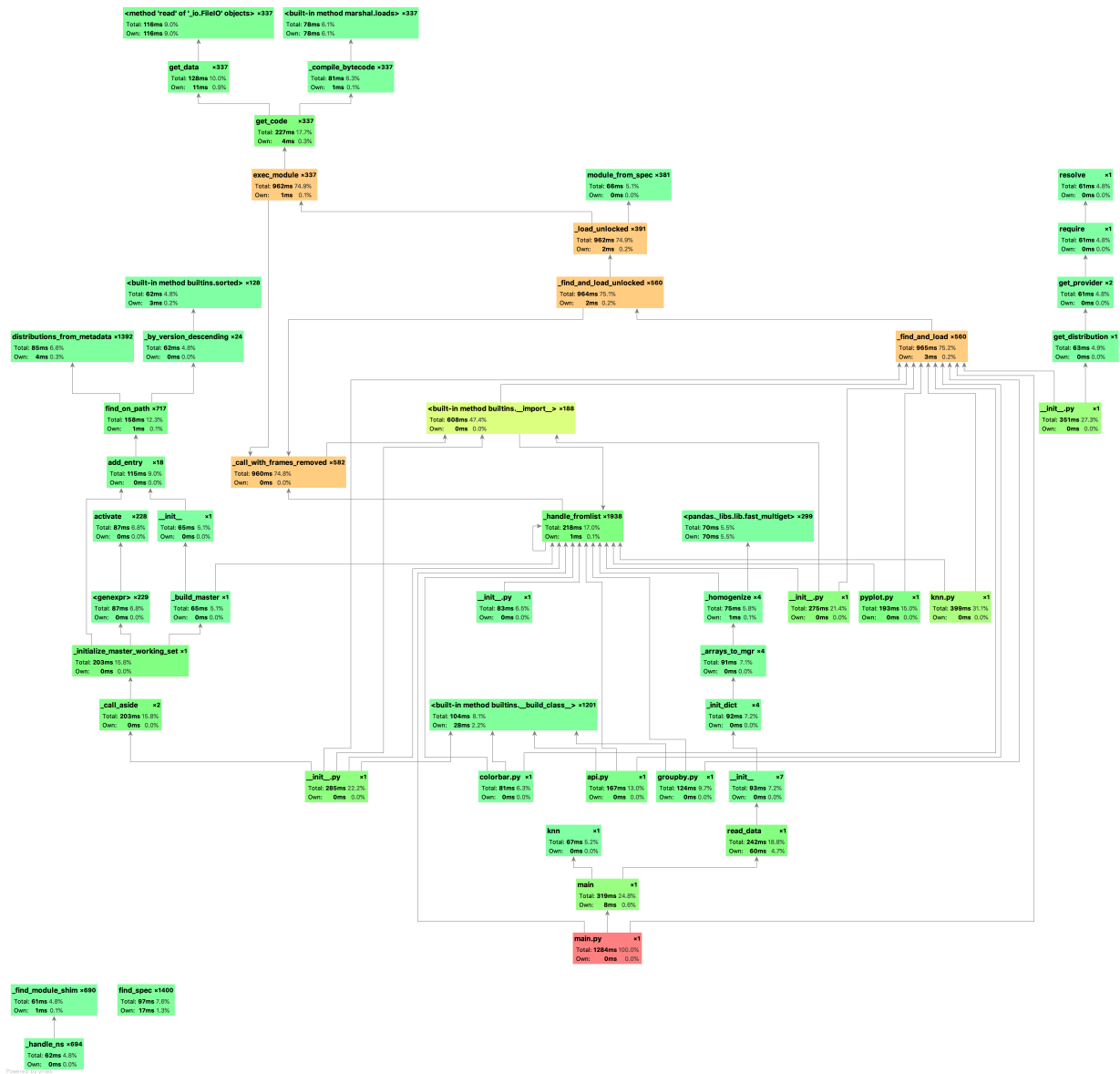
```
Timer unit: 1e-06 s

Total time: 0.073344 s
File: /Users/Yang/Developer/420235DataMining/hw1/q1/knn.py
Function: knn at line 5

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     5                                           @profile
     6                                           def knn(df, k, coefficient):
     7         1         13.0     13.0      0.0       hash_size = int(coefficient * df.shape[1])
     8         1        223.0    223.0      0.3       lsh = LSHash(hash_size, input_dim=df.shape[0])
     9       299        443.0      1.5      0.6       for vipno in df:
    10       298      63103.0    211.8     86.0           lsh.index(df[vipno], extra_data=vipno)
    11         1       2874.0   2874.0      3.9       random_column = df[df.columns.to_series().sample(1)]
    12         1          5.0      5.0      0.0       random_vip = random_column.columns.values[0]
    13         1        203.0    203.0      0.3       logging.info('random vipno: {}'.format(random_vip))
    14         1       6445.0   6445.0      8.8       res = lsh.query(random_column.values.flatten())[0: k + 1]
    15         1         12.0     12.0      0.0       print('vipno in ranked order using kNN(k = {}):'.format(k))
    16         1          1.0      1.0      0.0       knns = []
    17         7          4.0      0.6      0.0       for item in res:
    18         6          4.0      0.7      0.0           if item[0][1] != random_vip:
    19         5          7.0      1.4      0.0               print(item[0][1])
    20         5          6.0      1.2      0.0               knns.append(item[0][1])
    21         1          1.0      1.0      0.0       return random_vip, knns[:5]
```

```
Line #    Mem usage    Increment   Line Contents
================================================
     5  112.758 MiB  112.758 MiB   @profile
     6                             def knn(df, k, coefficient):
     7  112.758 MiB    0.000 MiB       hash_size = int(coefficient * df.shape[1])
     8  112.777 MiB    0.020 MiB       lsh = LSHash(hash_size, input_dim=df.shape[0])
     9  130.707 MiB    0.000 MiB       for vipno in df:
    10  130.707 MiB   17.930 MiB           lsh.index(df[vipno], extra_data=vipno)
    11  130.969 MiB    0.262 MiB       random_column = df[df.columns.to_series().sample(1)]
    12  130.969 MiB    0.000 MiB       random_vip = random_column.columns.values[0]
    13  130.969 MiB    0.000 MiB       logging.info('random vipno: {}'.format(random_vip))
    14  131.016 MiB    0.047 MiB       res = lsh.query(random_column.values.flatten())[0: k + 1]
    15  131.016 MiB    0.000 MiB       print('vipno in ranked order using kNN(k = {}):'.format(k))
    16  131.016 MiB    0.000 MiB       knns = []
    17  131.016 MiB    0.000 MiB       for item in res:
    18  131.016 MiB    0.000 MiB           if item[0][1] != random_vip:
    19  131.016 MiB    0.000 MiB               print(item[0][1])
    20  131.016 MiB    0.000 MiB               knns.append(item[0][1])
    21  131.016 MiB    0.000 MiB       return random_vip, knns[:5]
```

```
<method 'read' of '_io.FileIO' objects>  ×337
Total: 116ms 9.0%
Own:   116ms 9.0%

<built-in method marshal.loads>  ×337
Total: 78ms 6.1%
Own:   78ms 6.1%

get_data   ×337
Total: 128ms 10.0%
Own:   11ms 0.9%

_compile_bytecode  ×337
Total: 81ms 6.3%
Own:   1ms 0.1%

get_code   ×337
Total: 227ms 17.7%
Own:   4ms 0.3%

exec_module ×337
Total: 962ms 74.9%
Own:   1ms 0.1%

module_from_spec  ×381
Total: 66ms 5.1%
Own:   0ms 0.0%

resolve   ×1
Total: 61ms 4.8%
Own:   0ms 0.0%

_load_unlocked  ×391
Total: 962ms 74.9%
Own:   2ms 0.2%

require   ×1
Total: 61ms 4.8%
Own:   0ms 0.0%

<built-in method builtins.sorted>  ×128
Total: 62ms 4.8%
Own:   3ms 0.2%

_find_and_load_unlocked  ×560
Total: 964ms 75.1%
Own:   2ms 0.2%

get_provider  ×2
Total: 61ms 4.8%
Own:   0ms 0.0%

distributions_from_metadata ×1392
Total: 85ms 6.6%
Own:   4ms 0.3%

_by_version_descending  ×24
Total: 62ms 4.8%
Own:   0ms 0.0%

_find_and_load  ×560
Total: 965ms 75.2%
Own:   3ms 0.2%

get_distribution ×1
Total: 63ms 4.9%
Own:   0ms 0.0%

find_on_path  ×717
Total: 168ms 12.3%
Own:   1ms 0.1%

<built-in method builtins.__import__>  ×188
Total: 608ms 47.4%
Own:   0ms 0.0%

__init__.py   ×1
Total: 351ms 27.3%
Own:   0ms 0.0%

add_entry   ×18
Total: 115ms 9.0%
Own:   0ms 0.0%

_call_with_frames_removed  ×582
Total: 960ms 74.8%
Own:   0ms 0.0%

activate   ×228
Total: 87ms 6.8%
Own:   0ms 0.0%

__init__   ×1
Total: 65ms 5.1%
Own:   0ms 0.0%

_handle_fromlist ×1938
Total: 218ms 17.0%
Own:   1ms 0.1%

<pandas._libs.lib.fast_multiget>  ×299
Total: 70ms 5.5%
Own:   70ms 5.5%

<genexpr>  ×229
Total: 87ms 6.8%
Own:   0ms 0.0%

_build_master  ×1
Total: 65ms 5.1%
Own:   0ms 0.0%

__init__.py   ×1
Total: 83ms 6.5%
Own:   0ms 0.0%

_homogenize  ×4
Total: 75ms 5.8%
Own:   0ms 0.0%

__init__.py   ×1
Total: 275ms 21.4%
Own:   1ms 0.1%

pyplot.py   ×1
Total: 193ms 15.0%
Own:   0ms 0.0%

knn.py   ×1
Total: 399ms 31.1%
Own:   0ms 0.0%

_initialize_master_working_set ×1
Total: 203ms 15.8%
Own:   0ms 0.0%

_arrays_to_mgr  ×4
Total: 91ms 7.1%
Own:   0ms 0.0%

_call_aside   ×2
Total: 203ms 15.8%
Own:   0ms 0.0%

<built-in method builtins.__build_class__> ×1201
Total: 104ms 8.1%
Own:   28ms 2.2%

_init_dict   ×4
Total: 92ms 7.2%
Own:   0ms 0.0%

__init__.py   ×1
Total: 285ms 22.2%
Own:   0ms 0.0%

colorbar.py  ×1
Total: 81ms 6.3%
Own:   0ms 0.0%

api.py   ×1
Total: 167ms 13.0%
Own:   0ms 0.0%

groupby.py  ×1
Total: 124ms 9.7%
Own:   0ms 0.0%

__init__   ×7
Total: 93ms 7.2%
Own:   0ms 0.0%

knn   ×1
Total: 67ms 5.2%
Own:   0ms 0.0%

read_data   ×1
Total: 242ms 18.8%
Own:   60ms 4.7%

main   ×1
Total: 319ms 24.8%
Own:   8ms 0.6%

main.py   ×1
Total: 1284ms 100.0%
Own:   0ms   0.0%

_find_module_shim ×690
Total: 61ms 4.8%
Own:   1ms 0.1%

find_spec ×1400
Total: 97ms 7.6%
Own:   17ms 1.3%

_handle_ns  ×694
Total: 62ms 4.8%
Own:   0ms 0.0%
```

Powered by...

## Screenshot

```
/usr/local/bin/python3.6 /Users/Yang/Developer/420235DataMining/hw1/main.py
INFO:root:DataFrame shape: (2635, 298)
DEBUG:root:DataFrame info: None
<class 'pandas.core.frame.DataFrame'>
Index: 2635 entries, 10000004 to 40000700
Columns: 298 entries, 13205496418 to 6222021615015662822
dtypes: float64(298)
memory usage: 6.0+ MB
INFO:root:random vipno: 2900000394193
vipno in ranked order using kNN(k = 5):
1595141299820
1593140598586
1590151544861
1595142205462
1593140967467

Process finished with exit code 0
```