STA 663
Hierarchical Agglomerative Clustering using Locality-Sensitive Hashing

Mélanie Lai Wai
Weisong Zhu

Wednesday May 2, 2018

Link to Github repo: https://github.com/melaiwai/LSH-link

1 Abstract

In this paper, we will implement the fast agglomerative hierarchical clustering algorithm described in the paper titled "Fast Agglomerative Hierarchical Clustering Algorithm using Locality-Sensitive Hashing" by Koga, Ishibashi, and Watanabe. The single linkage clustering method is a fundamental agglomerative hierarchical clustering algorithm where n data points each start as a single cluster, and all end up in one large cluster through the recursive merging of the most similar pairs of clusters, n representing the number of data points. However, the single linkage method has a time complexity of $O(n^2)$ and rapidly decreases in efficiency as datasets increase in size. This makes the algorithm inefficient for very large datasets. The algorithm explored in this paper aims to provide an alternative to the single linkage method with a time complexity of $O(nB)$, where B is the maximum number of points going into a single hash entry which diminishes to a small constant relative to n for sufficiently large hash tables. It achieves this by using locality-sensitive hashing (LSH), a probabilistic approximation algorithm for the nearest neighbor search. Therefore, instead of relying on a distance matrix, the LSH-link finds the points with the highest probabilities of being close to a given point p, and then compares their distances to p.

2 Background

The algorithm, named the "LSH-link algorithm" by its developers, addresses the time complexity problem of the single linkage clustering method due to computation of pairwise distances between all data points in order to determine which clusters to merge at any given step. By using locality-sensitive hashing, the LSH-link method introduces a step that first identifies points in different clusters with the same hash values, and only computes pairwise distances between those points. As a result, the LSH-link clustering method demonstrates greater efficiency, especially when it comes to large datasets.

2.1 Possible Applications

In the same way that single-linkage clustering is used in cluster analysis, in various fields such as biology, medicine, or information science, the LSH-link method uses a bottom-up approach to group data points into clusters in such a way that points within the same cluster are more similar to each other than to points in other clusters. Therefore, the LSH-link method can be applied to any problem where the single-linkage method is relevant. The difference between the two is that researchers are able to adjust the LSH parameters to obtain a simpler ("coarse-grained") hierarchical structure to represent the data because LSH-link uses an approximation to find the nearest neighbors. This is especially useful when the dataset is large, and the data analysts wish to get an overview of the data in a cost-effective way before proceeding with further analysis.

2.2 Advantages and Disadvantages

Advantages:

- Relative to the single-linkage method, the LSH-link algorithm extracts fewer layers and can produce simple dendrograms that can be easily understood. This is useful if the data analysts wish to obtain a "coarse-grained" view of the data.
- The LSH-link algorithm runs faster than the single linkage method.

Disadvantages:

- The LSH-link method Does not reach the same degree of accuracy as the single-linkage method, as the LSH-link algorithm uses nearest neighbor approximation.
- It can be difficult to decide how to set parameters, and understand how these parameters affect the results.

3 Description of Algorithm

The parameters of the algorithm are first set. Parameters are C, r, k, l, A.

Parameter	Description
C	a constant such that the maximum value of a coordinate in point $p = (x_1, ..., x_d)$ is less than C, where d is the dimension of p
r	a distance threshold value to determine whether points are merged or not
k	the number of sampled bits from a set $\{1, 2, ..., C * d\}$ values
l	the number of hash functions to generate values from
A	a constant corresponding to the increase ratio of r

The algorithm works as follows:

Step 1 : For each point $p = (x_1, x_2, ..., x_d)$ in the dataset, l hash values, $h_1(p), h_2(p), ..., h_l(p)$ are computed. For $i = 1, ..., l$, p is stored in the bucket in the i-th hash table with index $h_i(p)$. However, if another point in the same cluster as p is already in the bucket, p is not stored.

How to compute the hash value of a data point $p = (x_1, x_2, ..., x_d)$:

First we transform p to a Cd-dimensional vector

$$\nu(p) = \text{Unary}_C(x_1) \dots \text{Unary}_C(x_d)$$

where $\text{Unary}_C(x) = 11111 \dots 00000$ with x ones and $C - x$ zeroes.

Then we sample k values out of the set $\{1, 2, 3, ..., Cd\}$. The k values form a set I corresponding to indices. Suppose $k = 5$, $I = \{2, 4, 5, 8, 9\}$ and $\nu(p) = 11100000001111110000$. Then this means that we will pick the 2nd, 4th, 5th, 8th, and 9th bit from $\nu(p)$ and the hash value of p, $h_I(p)$ is 10000.

l hash functions are created by sampling different I for a given k. In this way, a hash table is created for each p and l.

Step 2 : For each point p, from the set of points that enter the same bucket as p in at least one hash table, the algorithm computes their distance from p and selects the points whose distances from p are less than r. In our algorithm, we use the Euclidean distance or $l2$-norm to compute distance r between any two points.

Step 3 : The pairs of clusters, where each cluster is represented by a point in *Step 2*, are connected.

Step 4 : If the number of clusters after *Step 3* is 1, the algorithm terminates. If the number of clusters is larger than one, r is set to a larger vaue, as determined by the constant A, i.e. new value of r, $r_{\text{new}} = Ar$. Then the algorithm repeats using r_{new} and attempts to connect the remaining clusters.

Step 5 : LSH-link decreases k, the number of sampled bits, as r increases.

$$k \approx \frac{dC}{2r}\sqrt{d}$$

Using this new k-value, we return to *Step 1* to generate new hash values. This is done until we end up with one cluster.

4 Optimization for Performance

4.1 Vectorization

Our first step to do optimization is vectorization. We use NumPy array to replace most list type and use map, reduce functions to replace most for-loops.

4.2 Cython

Using a small data set (about 1,500 points), we profiled LSH-clustering function. As shown below, most of the computing time was spent calling function "fit", and in the function "fit", searching the set of the nearest points of the point p ("nn search") takes the most time. This function takes about 70% of the total time. So we used Cython to turn this function into C code.

We had difficulty compiling the .pyx file into a python module. The C codes and result can be observed by running LSH_optimized-test.ipynb.

6000	0.099	0.000	0.099	0.000	<ipython-input-4-1bf9024215f3>:25(change_unary)
180000	5.984	0.000	11.310	0.000	<ipython-input-4-1bf9024215f3>:35(get_h_value)
4	1.937	0.484	16.871	4.218	<ipython-input-4-1bf9024215f3>:38(hash_table)
6000	0.117	0.000	0.215	0.000	<ipython-input-4-1bf9024215f3>:44(<lambda>)
180000	3.273	0.000	14.583	0.000	<ipython-input-4-1bf9024215f3>:47(<lambda>)
4	0.002	0.000	204.233	51.058	<ipython-input-4-1bf9024215f3>:50(get_buckets)
120	0.024	0.000	204.231	1.702	<ipython-input-4-1bf9024215f3>:51(<lambda>)
6000	38.699	0.006	284.420	0.047	<ipython-input-4-1bf9024215f3>:54(nn_search)
3801000	85.940	0.000	147.337	0.000	<ipython-input-4-1bf9024215f3>:58(<lambda>)
170462	6.433	0.000	35.064	0.000	<ipython-input-5-80fc7901cc40>:11(merge_nodes)
1	13.774	13.774	717.674	717.674	<ipython-input-5-80fc7901cc40>:42(fit)
1	0.018	0.018	0.033	0.033	<ipython-input-5-80fc7901cc40>:46(<listcomp>)

4.3 Parallelism

We use ipyparallel and set 4 engineers to improve the "get buckets" function. Because most parts of the function need to deal with the class Nodes, we just can parallel the part of preprocessing the raw data.

4.4 Time Cost Comparative Analysis

After turning the "nn search" function into C code, we can find there is a significant improvement in the time cost. The time cost decreases from 2 minutes 40 seconds to only 29.1 seconds, which saves about 81% of total time. After applying parallelism, the time cost decreases about 8 seconds, which is also a good improvement.

Optimization methods	Vectorization	Vectorization + Cython	Vectorization + Cython + Parallelism
Time cost (CPU times)	user 2min 40s, sys: 400 ms, total: 2min 40s	user 28.7 s, sys: 392 ms, total: 29.1 s	user 20.7 s, sys: 564 ms, total: 21.2 s

5 Applications to Datasets

5.1 Simulated Datasets

5.1.1 First Simulated Dataset

The first simulated dataset aims to mirror the one used in Figure 9 in the original paper.

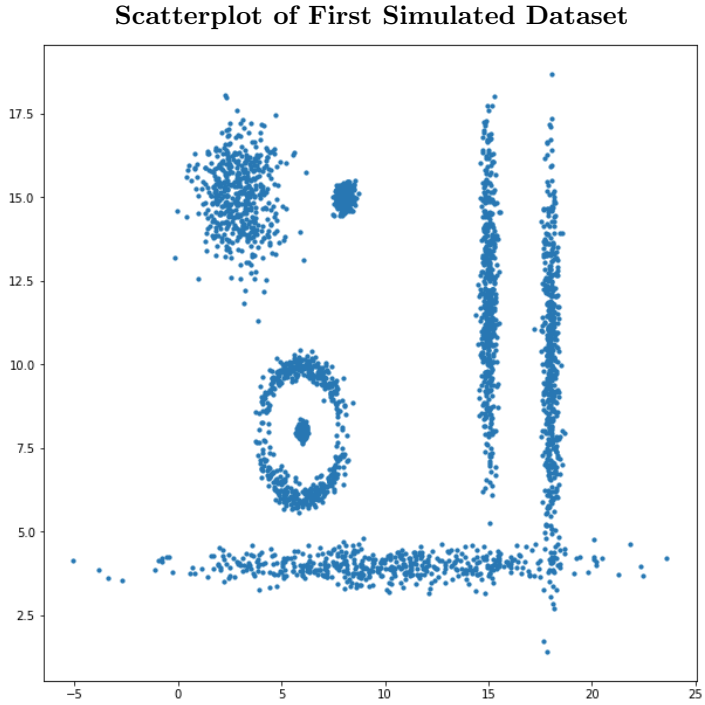

Figure 1: 7 clusters

As we can see from the scatterplot, we expect to find 7 main clusters in the data.

Applying the LSH-link algorithm with parameters $k = 10$, $R = 0.8$, $A = 1.5$, $C = 11$, $l = 30$ to the first simulated dataset, we obtain the following:

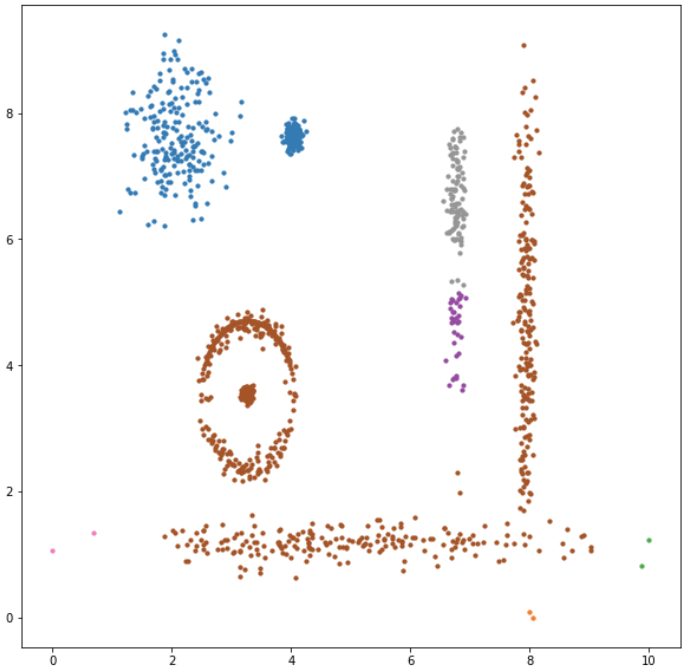

Figure 2: CPU times: user 22.1 s, sys: 436 ms, total: 22.5 s, Wall time: 22.8 s

Since the algorithm runs until only a single cluster remains, we have to examine the tree output in order to find the desired number of clusters we wish to display. Here, we display 7 clusters.

5.1.2 Second Simulated Dataset

For the second simulated, we drew inspiration from the demo provided at the following link: Plot Cluster Comparison

The two colors represent the truth, i.e. that there are two main clusters within the data.

Scatterplot of Second Simulated Dataset

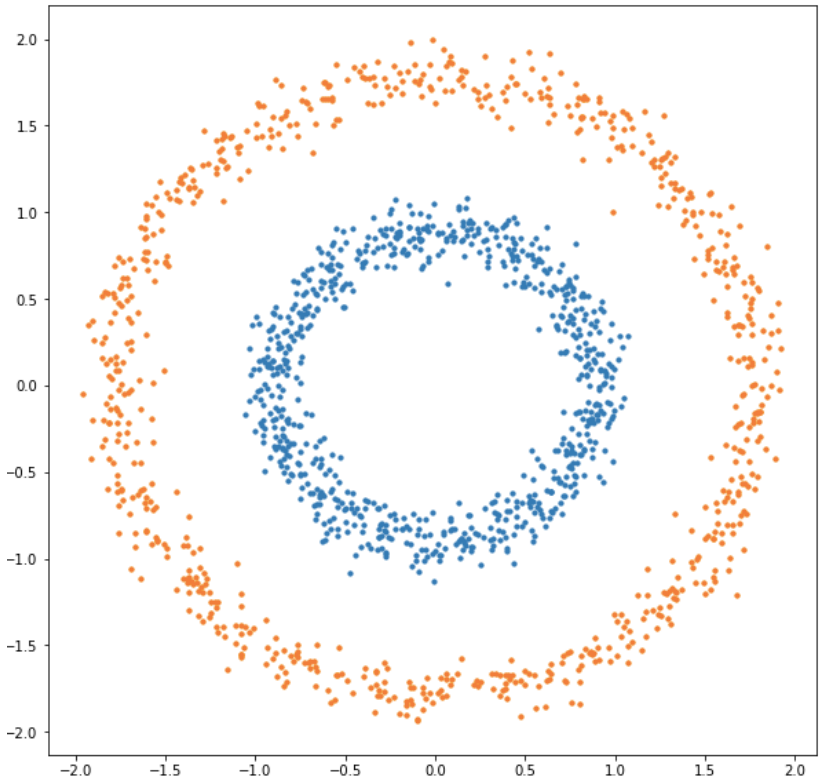

Figure 3: 2 clusters

Applying the LSH-link algorithm to the second simulated dataset with parameters $k = 5$, $R = 0.8$, $A = 1.5$, $C = 11$, $l = 30$, we obtain the following results:

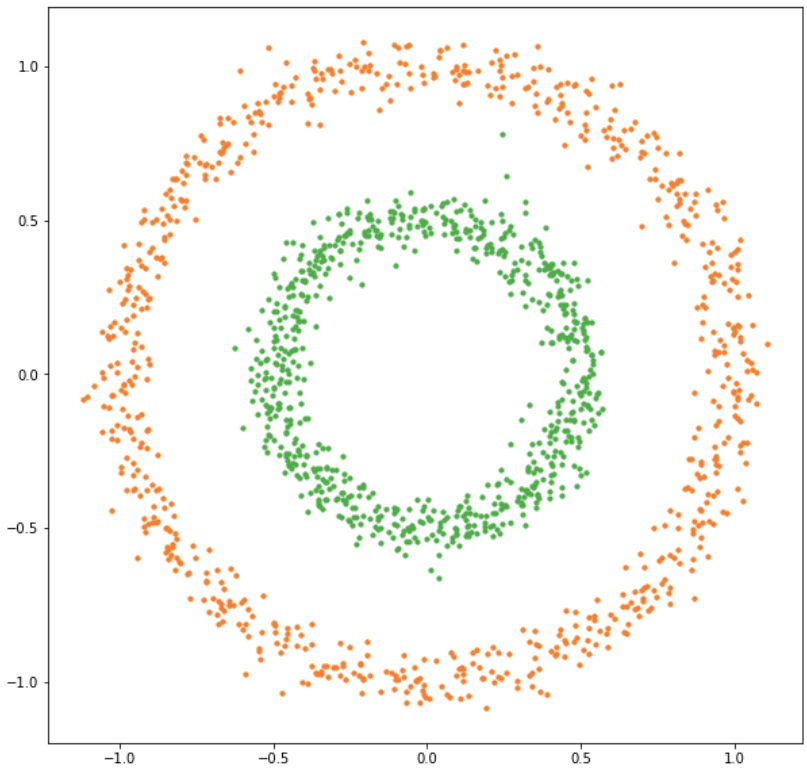

Figure 4: 2 clusters

5.2 Real Datasets

We were unable to find the datasets presented in the original paper, so we decided to implement the algorithm using the famous iris dataset and a business dataset.

5.2.1 Iris dataset

We fit the LSH algorithm with parameters $R = 0.2$, $A = 1.5$, $C = 11$, $l = 30$. To monitor accuracy, we compare it to the results obtained using the single linkage algorithm from the available "sklearn" package and, as we examine the array of labels assigned to each data point, we find that the results from the 2 algorithms are very similar, except for a few data points, when we cut down the tree to 3 clusters. This means that our algorithm does in this case approach the accuracy of the single linkage algorithm.

Labels for different clusters from LSH algorithm

```
x1.astype(int)
```

```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 0, 2, 2, 0, 2, 2, 2, 2, 2,
       2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 0,
       2, 2, 2, 2, 2, 0, 2, 2, 2, 2, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 2, 2, 0, 2, 0, 0,
       0, 2, 0, 0, 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2])
```

Figure 5: 3 clusters

Labels for different clusters from Single Linkage algorithm

```
np.array(model1.labels_)
```

```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 2, 2, 2, 2, 0, 2, 0, 2,
       2, 0, 2, 0, 0, 2, 2, 2, 2, 0, 2, 0, 2, 0, 2, 2, 0, 0, 2, 2, 2, 2,
       2, 0, 0, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2, 0])
```

Figure 6: 3 clusters

5.2.2 Business dataset

We fit the LSH algorithm with parameters $R = 1.5$, $A = 1.5$, $C = 11$, $l = 30$. Again, we compare it to the results obtained using the single linkage algorithm from the available "sklearn" package. We cut down the tree until we obtain 3 main clusters. After examining the array of labels assigned to each data point, we notice that the results are once more very similar to the single linkage results. This means that our algorithm approaches the accuracy of the single linkage algorithm.

Labels for different clusters from LSH algorithm

```
x.astype(int)
```

```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Figure 7: 3 clusters

Labels for different clusters from Single Linkage algorithm

```
x.astype(int)
```

```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Figure 8: 3 clusters

6 Comparative Analysis with Competing Algorithms

We will compare LSH-link with single linkage clustering and DBSCAN, an algorithm that uses a density-based approach to find neighbors. All three algorithms are applied to the first simulated

dataset, similar to the one used in the original paper.

6.1 Single Linkage Clustering

We compare LSH-link to single linkage clustering.

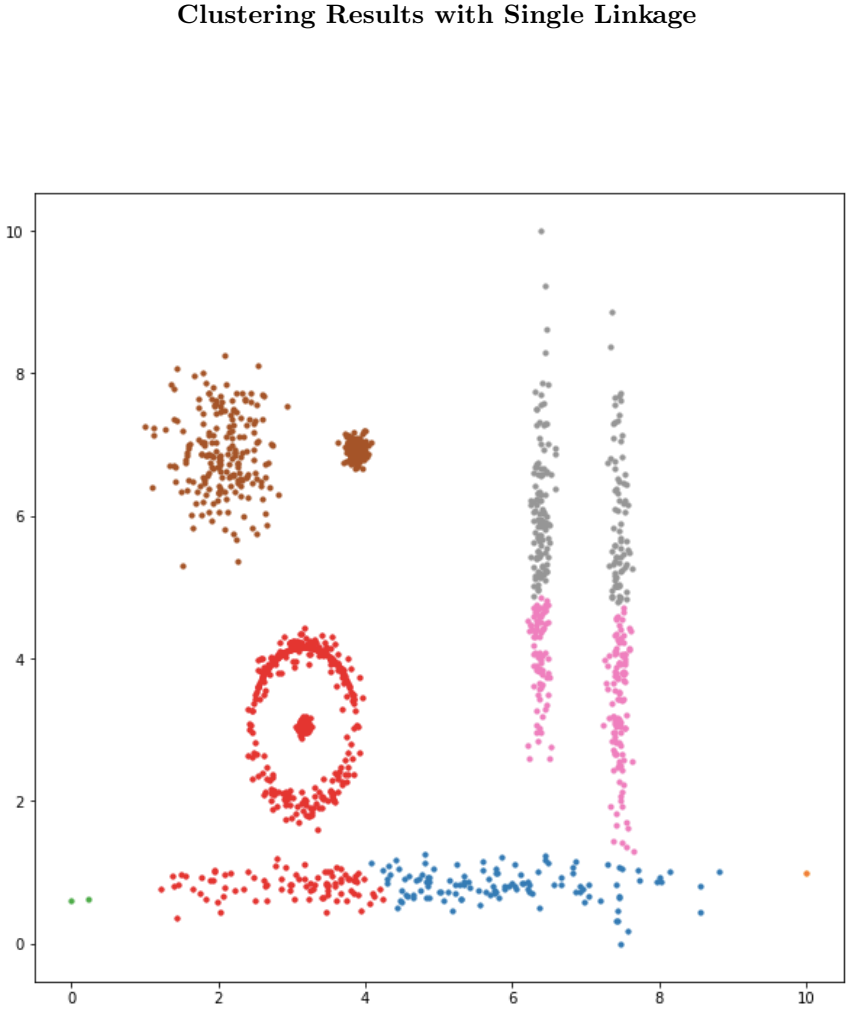

Figure 9: CPU times: user 6min 33s, sys: 800 ms, total: 6min 34s, Wall time: 6min 37s

As we can see from the graph, the accuracy of the single linkage method is average, as it is not able to detect the two distinct clusters at the top left and the bottom left. Timewise, the code runs extremely slowly.

6.2 DBSCAN: Density-Based Spatial Clustering of Applications with Noise

We compare LSH-link with DBSCAN, a density-based clustering algorithm with ϵ (parameter of DBSCAN) = $R = 0.8$, as recommended by the authors of the paper.

Clustering Results with DBSCAN

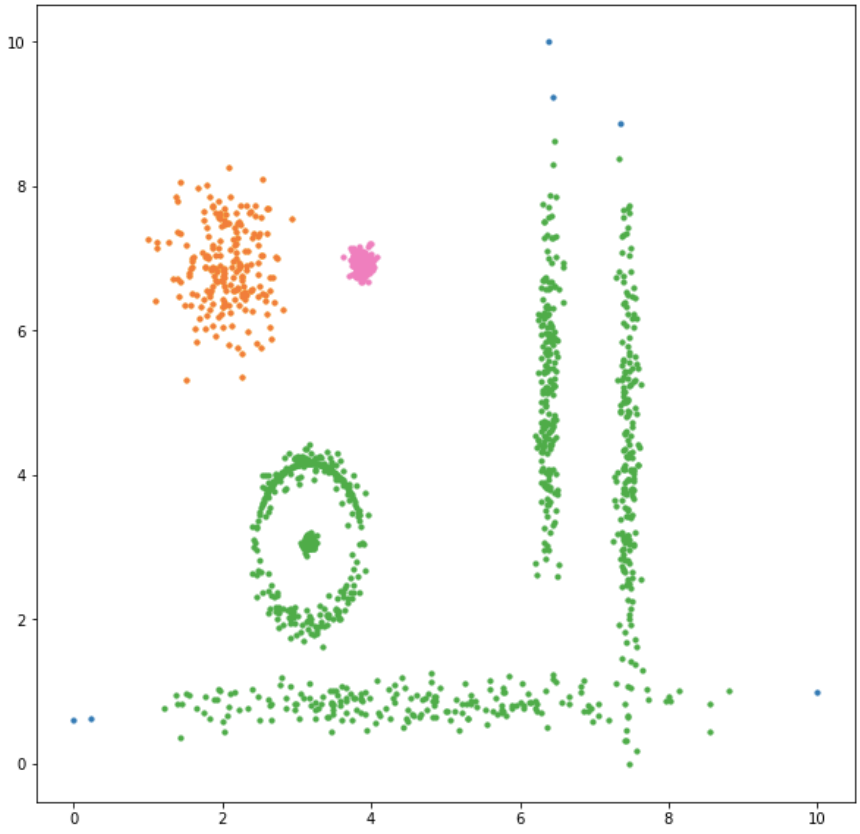

Figure 10: CPU times: user 16 ms, sys: 0 ns, total: 16 ms, Wall time: 17.1 ms

As we can see from the output graph, DBSCAN has average performance in terms of accuracy as it is able to detect 4 clusters. However it is important to note that it can achieve much higher accuracy if we reduce the value of ϵ to 0.5. It performs extremely well timewise (17.1 ms vs. 17.2 s for LSH-link).

7 Discussion and Conclusion

The Locality-Sensitive Hashing clustering algorithm is one kind of unsupervised clustering method. It uses a very ingenious idea to try to get around the need to compute distances in the single linkage algorithm. The use of hash functions is a very cost-effective way to find the nearest neighbors of a point p. Compared to the single-linkage hierarchical clustering, it can achieve a higher speed and get more accurate cluster result in some cases. However, the LSH clustering needs four parameters to do the whole process (the initial distance R, the increase ratio A, the number of hash functions l, and the constant to do unary change C), and the outcome differs a lot based on different parameters. Also, because of the randomness of choosing hash functions, the algorithm may get different outcome given the same parameter. Thus, the algorithms robustness to the parameters is not good, and this may be the reason why this algorithm is not widely used.

7.1 Limitations

There are some limitations in this algorithm

a. The parameter R and A will influence the clustering outcome a lot. If the R and A are too small, there will be too many sub-clusters; while if the R and A are too big, all the data points will be divided into one cluster. So, we need to try different R and A to find an optimal number of clusters.

b. Additionally, unlike the normal single-linkage algorithm, there may be more than more clusters being merged into on cluster in each step. So, it is hard to stop the algorithm to the number of clusters that we want. We can only get all the clusters and then divide them manually.

c. Although the paper gives a function to compute k: $k \approx \frac{dC}{2r}\sqrt{d}$

   k still needs to meet the limitation of: $k \leq Cd$

   So we can get: $r \geq \sqrt{d}/2$

   When the number of attributes get larger, the initial distance becomes larger too, which may lead to the bad clustering performance mentioned before. So, if we want to use a small distance, we need to reset the k value.

d. Finally, the process to create hash functions is random, so each time you run the algorithm you may get a bit difference in the outcome, usually the id of each cluster node will change even the structure of the clustering tree keeps unchanged

8 References

1. Koga, Hisashi, et al. "Fast Agglomerative Hierarchical Clustering Algorithm Using Locality-Sensitive Hashing." Knowledge and Information Systems, vol. 12, no. 1, 2006, pp. 2553., doi:10.1007/s10115-006-0027-5.

2. "Comparing Different Clustering Algorithms on Toy Datasets." Comparing Different Clustering Algorithms on Toy Datasets - Scikit-Learn 0.19.1 Documentation, scikit-learn.org/stable/auto_examples/cluster/plot_cluster_comparison.html#sphx-glr-auto-examples-cluster-plot-cluster-comparison-py.

3. "Demo of DBSCAN Clustering Algorithm." Demo of DBSCAN Clustering Algorithm - Scikit-Learn 0.19.1 Documentation, scikit-learn.org/stable/auto_examples/cluster/plot_dbscan.html#sphx-glr-auto-examples-cluster-plot-dbscan-py.

4. "Hierarchical Clustering." https://zhuanlan.zhihu.com/p/32438294