

Real-Time Clustering for Large Sparse Online Visitor Data

Gromit Yeuk-Yin Chan
gromit.chan@nyu.edu
New York University

Fan Du
fdu@adobe.com
Adobe Research

Ryan A. Rossi
ryrossi@adobe.com
Adobe Research

Anup B. Rao
anuprao@adobe.com
Adobe Research

Eunyee Koh
eunyee@adobe.com
Adobe Research

Cláudio T. Silva
csilva@nyu.edu
New York University

Juliana Freire
juliana.freire@nyu.edu
New York University

ABSTRACT

Online visitor behaviors are often modeled as a large sparse matrix, where rows represent visitors and columns represent behavior. To discover customer segments with different hierarchies, marketers often need to cluster the data in different splits. Such analyses require the clustering algorithm to provide real-time responses on user parameter changes, which the current techniques cannot support. In this paper, we propose a real-time clustering algorithm, *sparse density peaks*, for large-scale sparse data. It pre-processes the input points to compute annotations and a hierarchy for cluster assignment. While the assignment is only a single scan of the points, a naive pre-processing requires measuring all pairwise distances, which incur a quadratic computation overhead and is infeasible for any moderately sized data. Thus, we propose a new approach based on MinHash and LSH that provides fast and accurate estimations. We also describe an efficient implementation on Spark that addresses data skew and memory usage. Our experiments show that our approach (1) provides a better approximation compared to a straightforward MinHash and LSH implementation in terms of accuracy on real datasets, (2) achieves a $20\times$ speedup in the end-to-end clustering pipeline, and (3) can maintain computations with a small memory. Finally, we present an interface to explore customer segments from millions of online visitor records in real-time.

CCS CONCEPTS

• **Computing methodologies** → **Cluster analysis** .

KEYWORDS

Clustering, Sparse binary data, Density peaks, Sketching, Spark

ACM Reference Format:

Gromit Yeuk-Yin Chan, Fan Du, Ryan A. Rossi, Anup B. Rao, Eunyee Koh, Cláudio T. Silva, and Juliana Freire. 2020. Real-Time Clustering for Large Sparse Online Visitor Data. In *Proceedings of The Web Conference 2020 (WWW '20)*, April 20–24, 2020, Taipei, Taiwan. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3366423.3380183>

1 INTRODUCTION

People’s behavior during their visits to commercial websites is being continuously recorded by the companies’ web analytics systems.

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution.

WWW '20, April 20–24, 2020, Taipei, Taiwan

© 2020 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC-BY 4.0 License.

ACM ISBN 978-1-4503-7023-3/20/04.

<https://doi.org/10.1145/3366423.3380183>

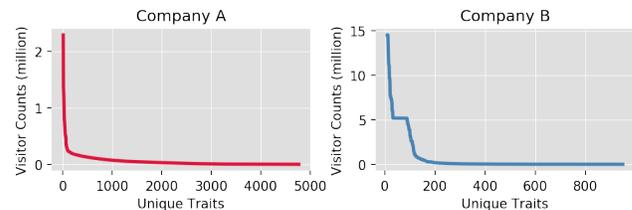


Figure 1: Histograms of website visitor traits of two large online retail stores.

Marketers of online retail stores often use custom binary attributes to capture the behaviors of their website visitors, such as whether a visitor has browsed a catalog of products, clicked an advertisement, or made a purchase. As a result, the visitor behavior is stored as a binary matrix $R \subseteq U \times T$, where U is a set of visitors u and T is a set of user behavior t . For example, an overview of two visitor datasets collected from two online retail stores is shown in Figure 1. The figure shows the distributions of traits by counting the numbers of visitors associated with each trait. The whole list of behavior consists of thousands of trait attributes to cover as many behaviors as possible from millions of online visitors every day so that it results in a large sparse binary matrix. One of the major objectives for capturing visitor behavior is to discover customer clusters based on their similar sets of attributes. For marketers, the ideal customer clusters vary based on a versatile range of needs. For example, they may want to find general clusters that divide the major populations or small clusters that represent niches. Also, they may want to further break down a cluster to obtain further splits based on the selected attributes. Therefore, to exploit the potential of the visitor behavior data to the fullest, the analytics system has to allow users to cluster visitors into different hierarchies in real-time.

Providing clustering results based on user inputs (e.g., number of splits) in real-time is challenging since it requires a response in a few seconds. Several data exploration goals such as making hypotheses or generalization will be adversely affected if a result does not come within 500ms [28]. Although there exist efficient clustering algorithms such as K-Means and Expectation Maximization (EM), they cannot provide real-time responses when the user changes the parameters and instead rerun the whole algorithm. One promising way to provide fast responses is to pre-compute a hierarchical relationship between data points so that the allocation of data to different clusters can be done within a single scan of data points. For example, it can be done within milliseconds for a few

million and scales linearly with a single modern CPU. Yet, the current hierarchical clustering techniques use linkage criterion, which requires a pairwise distance matrix that makes the pre-processing infeasible for any moderately sized data.

Since our focus is on the interactive data analysis, it is intuitive to think of ways to generate a hierarchy without pairwise distance comparisons. One data clustering algorithm that draws our attention is *density peaks* [37], which represents cluster centers as points that are surrounded by neighbors with lower densities in a local region. In Figure 2, cluster centers are the points with the highest densities in a local region. Once we identify the cluster center, the cluster assignment for remaining points can be done by referencing the same group from their respective nearest neighbor with a higher density (colored arrows in Figure 2). Clearly, although the naive calculations of density and nearest higher density neighbor for each point are to scan the whole sample resulting in total time complexity of $O(n^2)$, it is obviously not necessary since our objective is to query neighbors. Such observation provides a great room for efficiency improvement compared to hierarchical clustering while maintaining the benefit of providing real-time response.

In this paper, we propose a density-based sparse data clustering algorithm *sparse density peaks*, an efficient algorithm to compute density peaks for binary data using Jaccard as distance measure. Using set similarity measures like Jaccard similarity, we can approximate the density and distance attributes for each point with MinHash and Locality Sensitive Hashing (LSH) [17]. We leverage MinHash and LSH to provide close approximations of the attributes and — associate challenges at scale — an efficient implementation in Spark distributed computing framework. The implementation addresses data skew in similarity search and exploded intermediate products in neighbor query operations, which are two common challenges in handling large scale data. This can greatly utilize our approach in delivering quality results in real-world scenarios, as shown in our experimental results.

The contributions of this paper are summarized as follows:

- (1) A new clustering method called *Sparse Density Peaks* for clustering rows in large-scale sparse binary matrices based on their set similarities. We propose efficient and accurate approximations based on MinHash and LSH.
- (2) A highly scalable distributed implementation of Sparse Density Peaks using the Spark distributed computing framework. The implementation addresses the challenges of data distribution and memory usage in distributed computing.
- (3) Experiments demonstrating that our approaches enable a significant improvement in accuracy and speed up. These improvements enable real-time interactive clustering via an interactive user interface with visualizations.

2 RELATED WORK

Over the last decade, there have been a number of data clustering algorithms including k -means [29] and variants [33], hierarchical clustering [42], among many others. We refer the reader to a seminal survey on data clustering [4]. However, none of these approaches are well-suited for interactive real-time clustering since all are far too costly to compute after the user interactively changes the number of clusters or other hyperparameters. For instance, k -means

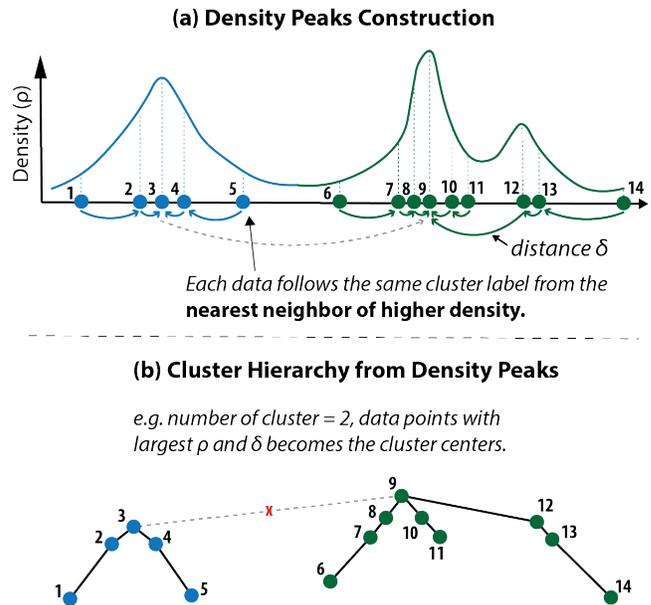


Figure 2: (a) Illustration of Density Peaks in one dimension data. Each data contains a density (ρ) attribute and a distance (δ) attribute that are used to construct a cluster hierarchy. (b) The hierarchy is constructed by attaching data points with their respective nearest neighbor of higher density.

takes roughly $O(nmkt)$ time where t is the number of iterations until convergence, n is the number of clusters, k is the number of attributes, n is the number of data points, and m is the number of attributes (columns). Notice that while t may be relatively small, both n and m can be quite large. It is also unstable, highly dependent on the initial starting centroids, and typically requires multiple runs to obtain a good clustering. More advanced online clustering methods include the usage of multi-armed bandit [16, 24, 27] and set cover [9]. While the t factor does not seem too bad, for large n it can become too costly, especially when requiring interactive real-time responses that are typically less than 1/10 of a millisecond. Most of them are also inappropriate for sparse data. Instead, the proposed sparse density peaks clustering is designed specifically for such large sparse data. In particular, sparse density peaks clustering leverages fast and efficient set similarity measures such as Jaccard similarity that are more appropriate for such large sparse data. In addition, sparse density peaks is designed to leverage the sparse structure of the data by considering only the nonzero entries of the large sparse matrix. Suppose X is the large sparse $n \times m$ matrix, then by definition $nm \gg \text{nnz}(X)$, where $\text{nnz}(X)$ is the number of nonzeros in X .

More recently, *density peaks clustering* [37] was proposed, which represents cluster centers as points that are surrounded by neighbors with lower densities in a local region. The original density peaks clustering has been applied in neuroscience [31], bioinformatics/gene expression analysis [30], image segmentation [39], key frame detection [45], among others [31]. Other work has extended density peaks with heat diffusion [31], fuzzy clustering [5], and even provided a MapReduce implementation of density peaks [2, 44].

However, density peaks is infeasible for real-time clustering of large data as it has a $O(n^2)$ time preprocessing step, which also requires nearly n^2 space. Due to the time and space requirements, density peaks clustering is impractical for use with any data of even moderate size. Moreover, its density based clustering nature makes data skew easily happened in neighbor search for sparse data. In this work, we describe a highly scalable clustering algorithm called *sparse density peaks* that addresses the time and space issues of density peaks to enable real-time clustering of large-scale customer data. There has been far less work focused on the real-time clustering problem. For real-time clustering, we require a nearly linear or sub-linear time algorithm. Since the goal is to use the real-time clustering algorithm for interactive data exploration, it should also be fast when a user changes the number of clusters.

There is also some work on clustering algorithms that find a hierarchy of clusters as opposed to a single clustering [36]. In this work, we focus on obtaining a hierarchy of clusters that can be interactively explored by the user in real-time. While most existing methods for this problem require nearly quadratic time (and space) [36], our approach is able to find a hierarchy of clusters in near-linear time using MinHash and LSH to approximate the density and distance computations.

There has also been recent work on parallel clustering algorithms for shared-memory and distributed architectures [1, 13, 15, 46]. Other work has focused on parallel implementations for GPUs [6]. In our work, we also provide a distributed implementation of our *sparse density peaks clustering* algorithm using Spark. Recently, there has been work on interactive data analysis and visualization [19, 22, 23]. The majority of work in this area has focused mainly on interaction techniques and visual interfaces for exploring data [14, 21, 26, 38] or tuning parameters of clustering algorithms and finding the best result sets [7, 25, 43]. In this work, we design fast and scalable clustering algorithms that can extend the support of interactive clustering and exploration to large-scale datasets.

3 SPARSE DENSITY PEAKS CLUSTERING

3.1 Density Peaks

Our approach to clustering binary sparse data is inspired by the concept of Density Peaks (DP) [37]. It has two assumptions: (1) cluster centers are surrounded by neighbors with lower local densities, and (2) they are relatively more distant to any points with higher local densities than the other data points. These observations are illustrated in Figure 2(a), where cluster centers (i.e. point 3 and point 9) are the points that are dense within their local regions. Here, the **local density** ρ_i of data point u_i is defined as:

$$\rho_i = \left| \left\{ u_j \mid \text{dist}(u_i, u_j) < d_c \right\} \right| \quad (1)$$

where d_c is the cutoff distance. The **minimum distance to higher density points** δ_i for each point is defined as:

$$\delta_i = \min_{j: \rho_i < \rho_j} \text{dist}(u_i, u_j) \quad (2)$$

By convention, the point with the largest density is assigned with the maximum value of δ . Based on these assumptions, DP computes local density ρ_i and distance δ_i for every point and users can label points with anomalously large δ_i and ρ_i as cluster centers.

Algorithm 1 Find Local Density

Input: dist (distance function), d_c (cutoff distance)

Output: ρ (density list)

```

1: for  $i = 1 : n$  do
2:    $\rho_i = 0$ 
3:   for  $j = 1 : n$  do
4:     if  $i \neq j$  and  $\text{dist}(i, j) \leq d_c$  then
5:        $\rho_i = \rho_i + 1$ 

```

Algorithm 2 Find Minimum Distance of Higher Density Points

Input: dist (distance function), ρ

Output: nn (nearest higher density neighbor list), δ (distance list)

```

1: for  $i = 1 : n$  do
2:    $\delta_i = \infty$ 
3:    $nn_i = null$  ▷ initialize nearest neighbor
4:   for  $j = 1 : n$  do
5:     if  $i \neq j$  and  $\rho_i < \rho_j$  and  $\delta_i > \text{dist}(i, j)$  then
6:        $\delta_i = \text{dist}(i, j)$ 
7:        $nn_i = j$ 

```

Algorithm 3 Cluster Assignment

Input: \mathcal{X} (cluster centers), nn (list of nearest higher density neighbor), sortedIndex (list of indexes sorted by ρ)

Output: C (list of cluster labels)

```

1: for  $i = 1 : \text{size}(\mathcal{X})$  do
2:    $C(\mathcal{X}(i)) = i$  ▷ assign cluster labels for centers
3: for  $i = 1 : \text{size}(\text{sortedIndex})$  do
4:   if  $C(\text{sortedIndex}(i)) == \text{empty}$  then ▷ no cluster labels
5:      $C(\text{sortedIndex}(i)) = C(nn(\text{sortedIndex}(i)))$ 

```

Finding δ_i also returns each point's **nearest neighbor with a higher density**¹ nn_i for each point. It act as the point's parent node in the hierarchy which are useful for cluster assignment. After choosing a set of cluster centers \mathcal{X} , each remaining point is assigned to the same cluster as its nearest neighbor, in an order of descending ρ , illustrated in Algorithm 3 and Figure 2(b).

Unlike popular partition-based clustering algorithms like K-Means, DP completes most of the computations before users start their analyses. As shown in line 3-5 in Algorithm 3, given the number of clusters, it only scans the points U once to produce the clustering results. Still, computing ρ and δ is expensive. Brute force approaches require measuring the distance between each pair of points (Algorithm 1 and 2), resulting in an $O(n^2)$ time complexity.

3.2 Extensions of Density Peaks

Before proposing our framework for accelerating the density and distance computations, we first introduce two useful extensions of DP. We extended these features based on the requirements of commercial data platforms and the needs of real customers.

Batch processing. Commercial data often comes in a huge volume, which requires a consistent and long usage of computing

¹Without loss of generality, we use the term *nearest neighbor* to refer to the *nearest neighbor with a higher density* for the remainder of the paper.

resources. Instead of constructing the density peaks in a single run, the algorithm can be run in an *anytime* manner [20] in which the dataset is pre-processed with interruptions. To be specific, given a set of points U , to compute the density ρ_i and distance δ_i for each point p_i , p_i does not need to compare with the whole U in one time. Instead, we can split U into multiple disjoint subsets and compare it with each subset one by one. Suppose after t -th batches of subsets, p_i obtains a density ρ_i . When the $(t + 1)$ -th batch U_{t+1} comes, the density is updated as follows:

$$\rho_{i_{t+1}} = \rho_{i_t} + \left| \left\{ u_j \mid \text{dist}(u_i, u_j) < d_c \forall u_j \in U_{t+1} \right\} \right| \quad (3)$$

After finishing computing the density ρ for all data points, we can compute the distance δ batch by batch similarly. Suppose after t -th batches of subsets, a data point p_i obtains a distance δ_i . When the $(t + 1)$ -th batch U_{t+1} comes, the distance is updated as follows:

$$\delta_{i_{t+1}} = \min(\delta_i, \min_{j: \rho_i < \rho_j \forall j \in U_{t+1}} \text{dist}(u_i, u_j)) \quad (4)$$

While the update functions look straightforward as they either involve only a summation or comparison, they provide huge advantages for both operations and implementation. First, it allows the pre-processing to be scheduled. For example, the algorithm can stop running if the server is busy and resumes when resources become more available. This enables the infrastructure to allocate the tasks more efficiently. Second, the pairwise comparisons are only conducted between a subset and the whole dataset at a time, which can drastically reduce the number of intermediate results throughout the computation. The reduce in memory usage can provide rooms for speed up strategies which we will introduce and demonstrate in Section 4 and 5.

Hierarchical clustering. As in Algorithm 3 and Figure 2, each data point is assigned to the same cluster as its nearest neighbor in descending order of density, which is the same as a hierarchy. The parent node of a data point is its nearest neighbor and all children nodes under a parent must have the same cluster label. Therefore, it is possible to perform clustering again on any cluster (i.e. subtree) without recomputing the whole pipeline. In practice, this allows analysts to drill down into a sub-group and discover fine-grained insights. For example, when one cluster exhibits a unique purchase behavior towards electronic appliances, analysts can further split the cluster into sub-groups of mobile phones or television purchases. We will propose an interactive system that leverages this capability in Section 5.3.

3.3 MinHash + LSH

As described in Section 3.1, in order to cluster a dataset, DP needs to estimate local density ρ_i and distance δ_i for each point. The time and memory required for computing the all-pair distances become untenable as numbers of data points grow. Since computing the density needs to find all the neighbors within a cutoff distance and computing the distance requires us to find the nearest neighbor with a higher density, our thoughts naturally turn to efficient methods for nearest neighbor search.

Our framework employs a MinHash with Locality-Sensitive Hashing (LSH) approach [8, 34] for reducing the search space of neighbors. We first generate a MinHash signature for each data

point (i.e., visitor) that approximates the Jaccard similarity of its column values (i.e., user behavior). Then, using the MinHash signatures, LSH hashes similar points into the same bucket, so that we only need to consider points within the same bucket when estimating the density and distance.

Formally, a MinHash signature is a vector of n MinHash values (i.e., sketches) $[h_1(u_i), h_2(u_i), \dots, h_n(u_i)]$, such that the similarity between two points is the probability of MinHash collision between two signatures (i.e., $\text{sim}(u_i, u_j) = \Pr[h_k(u_i) = h_k(u_j)]$). To produce MinHash signatures that can approximate the weighted set similarity, each MinHash function h_k is defined by the smallest element of a set of columns T_i , under ordering induced by the hashing function f for weighted set similarity [10]:

$$h_k(u_i) = \underset{t \in T_i}{\text{argmin}} \frac{-\log(x)}{w_t}, \text{ where } x \sim \text{Un}(0,1) \quad (5)$$

After the signature is computed, LSH will partition the signature vectors into b bands so that each band will have n/b hashes. The values of hashes become a bucket id, so that if two points are hashed into the same bucket id, it means that they have at least n/b number of same hashes (i.e., $1/b$ MinHash similarity).

3.4 Approximating Density ρ

Given a set of LSH partitions on the MinHash signatures, the query for a point's number of neighbors within a cutoff distance d_c (i.e., the point's local density ρ) can be illustrated by the following SQL expression:

```
SELECT COUNT(DISTINCT points) AS density
WHERE LSH_collision(p, points) >= N
AND distance(p, points) < d_c;
```

where `LSH_collision` measures the number of same hashes both the query and points share, N_B refers to the number of bands in LSH, N is a user-defined integer, and `distance` measures the true distance between the points that rectifies the false positives in the LSH partitions. For example, if there are 10 bands in the partitions and d_c equals to 0.5, each point will search for neighbors that share equal or more than 5 LSH buckets. As a result, each point does not need to compare with all data points (i.e. replace the for loop of line 4 in Algorithm 1 with the results from the neighbor query). The *observed density* will be the number of retrieved points whose distances are within d_c .

The reason the density derived from here is called *observed* is because the neighbors are queried based on a probabilistic nature of LSH. LSH might fail to include some neighbors that do not contain enough collisions but are located within the threshold. To produce more precise results, the approximated ρ should cover the estimation of including this missing population as well. The estimation works as follows. Denote the similarity between two points as s . The probability that two points share the same LSH bucket is s^r , where r is the number of rows in a LSH band. As the number of collisions of LSH buckets between two points N_c follows a binomial distribution with probability equals to s^r , the probability that N_c is greater than N is the cumulative probability function:

$$P(N_c \geq N | s^r) = \sum_{i=N}^{N_B} \binom{N_B}{i} (s^r)^i (1 - s^r)^{N_B - i} \quad (6)$$

Using the example above ($d_c = 0.5$, $N = 5$ and $N_B = 10$), if each band has one MinHash signature ($r = 1$), the probability that points with $s = 0.6$ being retrieved equals to 0.834. With the probability and observed neighbors, we can estimate the total population N_s using expectation:

$$E(Y_s) = P(N_c \geq N|s^r) \times N_s \quad (7)$$

where $E(Y_s)$ is the observed number of points having similarity s with the queried point.

By summing all N_s obtained from observed points, we can give a more precise estimation of the density ρ . For example, if the query returns two points with $s = 0.6$ and one point with $s = 0.5$, the approximated ρ will be $\frac{2}{0.834} + \frac{1}{0.623} = 4.00$.

3.5 Approximating Distance δ

Given a set of LSH partitions on MinHash signatures, the query for the nearest neighbor with a higher density can be illustrated by the following SQL expression:

```
SELECT TOP(points) AS nn
WHERE LSH_collision(p, points) >= 1
AND p.density < points.density
ORDER BY distance(p, points) ASC;
```

Similarly, by querying neighbors by LSH collisions, we can avoid the search of the whole sample space (i.e. replace line 5 in Algorithm 2 with the results from the neighbor query). Here, we examine all data points that fall into 1 of the same LSH buckets. We hope that the nearest neighbor appears among the collected samples. Yet, how probable is our “hope” correct? We need to estimate the accuracy of correct results since a wrong δ will impact the cluster quality.

Assume δ_i and nn_i exist for each point u_i and the retrieved nearest neighbor \bar{nn}_i has the distance $\bar{\delta}_i$. The true nearest neighbor must have a distance that is smaller or equal to \bar{nn}_i . Also, its LSH hashes do not collide with the point being queried. Therefore, the event that nn_i cannot be retrieved by the query is formulated as follows: (1) $\delta_i < \bar{\delta}_i$ and (2) $N_c < 1$. Since the only information we know about nn_i is that its similarity must be greater or equal to the similarity of the current query result \bar{s} , the probability of the event thus can be bounded as follows:

$$\begin{aligned} Accuracy(\delta_i) &= 1 - P(nn_i \neq \bar{nn}_i) \\ &= P(\text{no LSH collisions between } p_i \text{ and } nn_i) \\ &= 1 - (1 - s_{nn}^r)^{N_B} > 1 - \int_{\bar{s}}^1 (1 - s^r)^{N_B} ds \quad (8) \end{aligned}$$

The integral calculates the sum of chances that no LSH buckets collide across N_B bands for any possible outcome that has a similarity greater than the queried nearest neighbor. It only contains one variable s so that the integral can be solved by trapezoid rule [12]. Once we can obtain an accuracy for the nearest neighbor being retrieved, we can identify the points which have low accuracies and further process them with a correction approach.

As a result, the next question is how can we handle the points that have no nearest neighbor returned from the query or have a low value of accuracy. Since precisely correcting these points requires comparing them to the whole dataset which is inefficient, we rely on a heuristics that cluster centers are normally dense. We

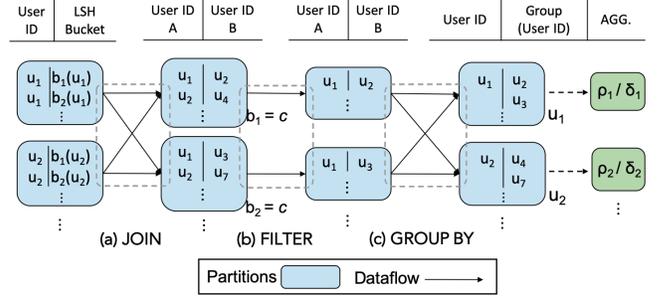


Figure 3: General pipeline of attribute calculations. A many to many data flow between partitions in different stages means that a re-partition occurs.

compare these points with a fixed number of most dense points (e.g. the top 1000 points with highest densities). The intuition is that these dense points are more likely to be the cluster centers without any prior information of nearest neighbors and they are more likely to return a nearest neighbor for each queried point. We will show this approach provide good effectiveness in Section 5.

4 IMPLEMENTATION

Given a large number of data, millions of densities and distances have to be computed. Yet, the querying operations can be run independently within each stage of density and distance calculations for each point. To leverage the parallel nature of these computations, we implement the framework using Spark. Like MapReduce, Spark is a distributed cluster computing framework and it also provides SQL query languages on structured data. For distributed computing frameworks, optimization for several types of queries plays an important role in speed and performance. To provide clear motivations and explanations of our query strategies, we first give an overview of the Spark architecture and main requirements for speed up strategies.

4.1 End-to-end Performance Overview

In Spark architecture, data are distributed across different partitions. All partitions execute the tasks concurrently until there is a need to re-partition the data, which is called a *shuffle*. The *shuffle* step is needed for a JOIN or GROUP BY operation since the data with the same keys have to reside in the same partition. For example, to compute the MinHash signatures and LSH bucket hashes, each data point only needs to calculate its hash values so that no re-partition is needed. However, to find the neighbors and calculate the density or distance, it requires a join on data points using hashes as the key and then a group by on user ID to calculate the counts or find out the nearest neighbor. These steps often result in a potential explosion of intermediate results and uneven distribution of data among the partitions, which are the main bottlenecks among the tasks in computing the density peaks. While Spark automatically generates an execution plan to reduce the possible number of shuffle operations, we list two design strategies when considering our implementation strategies:

e.g. number of bands = 4, requires at least 3 LSH collisions

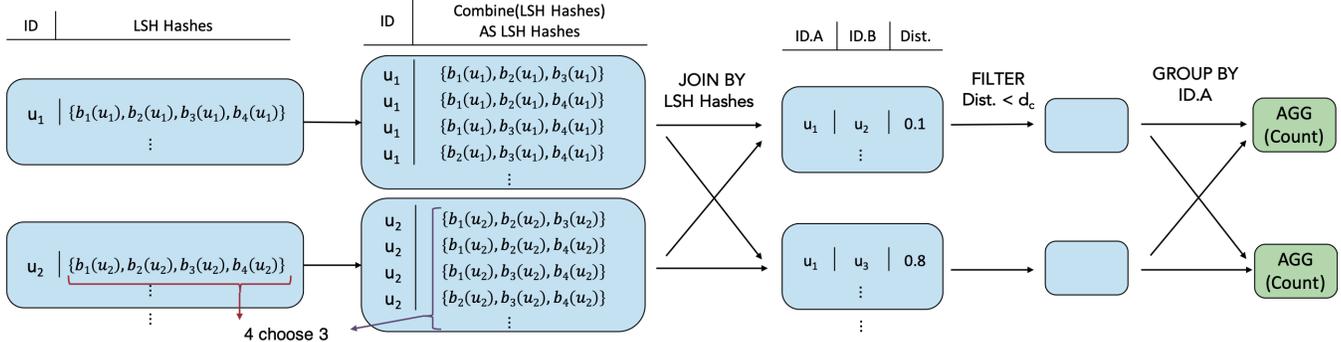


Figure 4: Join, filter, and group by conditions in calculating density.

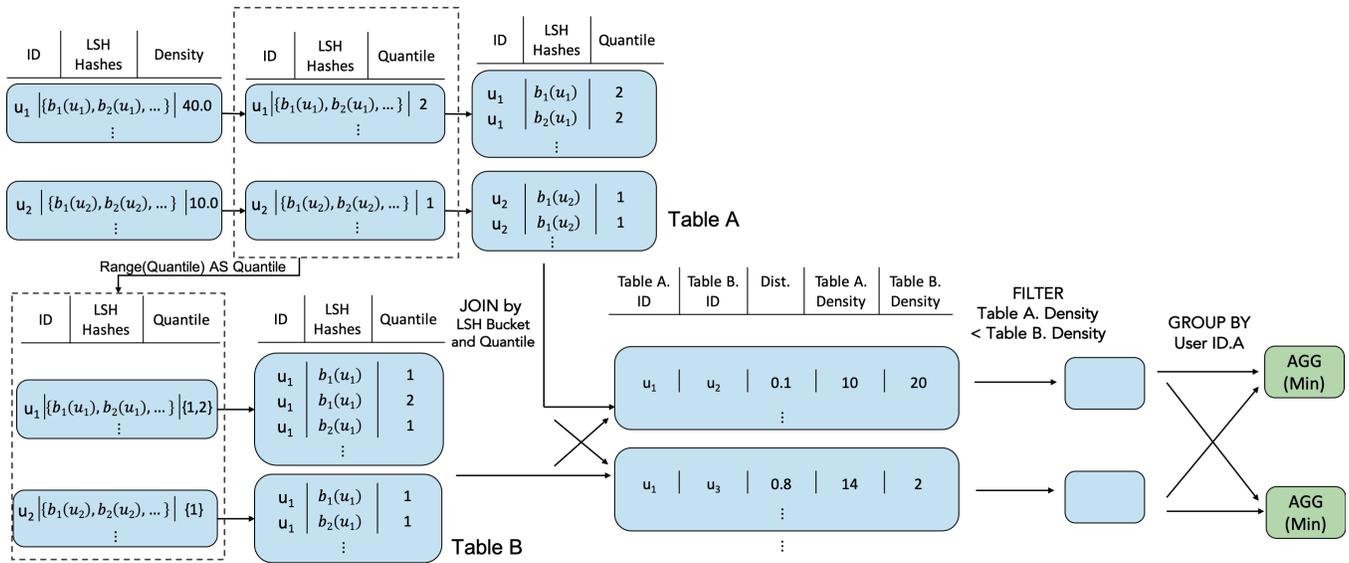


Figure 5: Join, filter, and group by conditions in calculating distance.

G1 Reduce the number of duplicate keys when joining or grouping two tables. When two tables are joined with duplicate keys, the number of intermediate products will grow quadratically which produces a high memory and network pressure to the partitions. For Spark, one of its efficiency comes from its ability to process data within the memory of the computing nodes. Although using memory provides a much faster runtime than saving and retrieving data through hard disk, the amount of memory available is often much smaller. If the amount of joined data exceeds the available memory, Spark will spill the joined data to hard disk first before applying further tasks, which will drastically slow down the computations. Since only joining data by a subset of hashes will cause highly similar pairs to have multiple collisions, we should introduce more criteria to further reduce the number of duplicate keys to reduce these unnecessary collisions.

G2 Ensure an even distribution of data among partitions. Since the partitions run concurrently and wait until all partitions finish the tasks, it is important to make sure all partitions share a similar amount of data so that the tasks will not be stuck by a single partition.

4.2 Join-Filter-Group By based Algorithm

Our calculations on density and distance both have a similar pipeline containing join, filter, and group by operations (Figure 3). First, all data points generate a set of LSH buckets as hashes. Having one column of ID and one column of hashes, the table performs a join using the hashes as the key to collect collided user pairs. Then, the pipeline undergoes a filter operation to remove ineligible pairs such as duplicates or pairs exceeding the cutoff distance. Finally, to calculate the attributes, it performs a group by operation using user ID as the key and aggregates the groups by count or minimum distance to collect ρ or δ respectively. Since these operations are performed

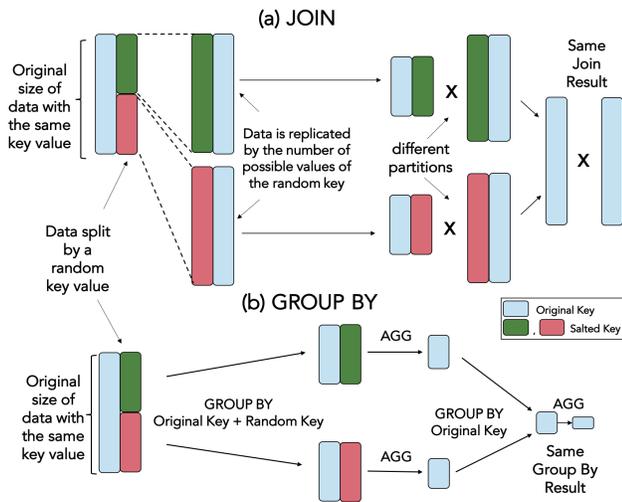


Figure 6: Illustration of salting techniques in join and group by operations. The goal is to split data with same key values into a different partitions.

in a distributed system, the data need to be re-partitioned during the self-join and group by steps. We now provide more details about how we compute density and distance.

Density (ρ) calculations. The SQL expression in Section 3.4 can be fit into the pipeline by mapping the WHERE clause involving LSH_collisions into the join operation, the WHERE clause involving distance into the filter operation, and the COUNT aggregation into the group by objects. To apply the condition of having more than a number of hashes collisions as the keys in the join operation, our goal is to reduce the number of duplicated joined products from highly similar pairs since they share many same hash values (G1), we can directly use the combination of hashes as the keys in the join operation (Figure 4). For example, if we have 10 hashes and would like to get the user pairs that have more than 5 collisions, we can generate a table of users with their C_5^{10} combinations of LSH buckets in separate rows. As a result, the joined user pairs are guaranteed to have at least 5 LSH collisions and the pairs having 5 collisions will only be joined once.

Distance (δ) calculations. Similarly, the SQL expression in Section 3.5 can be fit into the pipeline by mapping the whole WHERE clauses into the join operation, the density comparisons into the filter operation, and the MIN(distance) aggregation into the group by objects. Since we want to join each data with candidates which have at least one hashes collision and also higher densities, in addition to LSH hashes, we create a new column that bins the density for each user into their corresponding quantile (with the *approxQuantile* method in Spark that uses the Greenwald-Khanna algorithm [18]) and use it as the key in the join operation. Once the quantile column is computed, we create a new table that transforms the quantile into a list of smaller or equal quantiles for each data (Table B in Figure 5). This is to enable each data row (Table A in Figure 5) to be joined not only the candidates with the same quantile but also the ones with higher density quantiles. Once the join is completed,

the distances will be fetched by performing a group by operation using the data from the original table.

Batching Processing. In Section 3.2, we illustrate that our approach allows data to update its density and distance whenever they are compared with a new subset of data. These can be easily done by replacing one side of the join with the subset in Figure 4 and 5. For density computation, the table join by hashes is done between the table containing all data points and the table containing the subset of points. By joining these tables and performing the group by operation by the IDs from the table with all data (i.e. ID.A are the IDs from the whole dataset and ID.B are the IDs from the subset in Figure 4), the count obtained will be the density of points collected from the subset, which can be added to the existing density results. For distance computation, we can replace the Table B in Figure 5 and perform similar operations. The nearest neighbor obtained from the subset can be compared with the current nearest neighbor.

4.3 Speed Up Strategies Addressing Data Skew

The major obstacle that hinders our implementation to increase the concurrency of parallelism in the system level (G2) is the data skew that happens commonly in real data (Figure 1). This problem is reflected in two stages of our pipeline. First, if many data have similar sets of attributes, they will share many common LSH hash values. It means that when the data are joined with these buckets (Figure 3(a)), there will be a small number of partitions accommodating the majority of joined intermediate products, leaving few tasks running for a long time in this stage. Secondly, after the join and filter operation, some data may have lots of candidates to proceed to the group by stage. For example, data points with small density will have lots of potential neighbors to compare with. Therefore, some partitions need to accommodate a disproportionately large number of data to calculate the aggregated results. This problem exists in all distributed computing frameworks like MapReduce and Flint as data are shuffled among different partitions. To address these problems, we apply a technique called “salting”, which is about changing the join or group by keys to redistribute data among partitions evenly.

For join operations between two tables, the salting technique works as follows. On one side of the table, we replicate it n times by creating a new key column where the value lies between 0 and n . On the other side, we add a new key column with random values between 0 and n . Note that we do not replicate the table on this side. Then, we join these two tables using the new column and column of hashes together as the keys. In this case, the rows associated with the same original hash values, which were supposed to be put into the same partition before, are now distributed into n partitions created from the replicates (Figure 6(a)). The figure illustrates that the original join result is split into multiple partitions.

For group by operations, the salting technique works by using one more group by stage (Figure 6(b)). Again, we add a new key column with random values. Then, we perform a group by operation using both as the keys and aggregate the groups with COUNT for density and MIN for distance. After that, we perform a group by operation using the original key and finally we aggregate the groups with SUM(COUNT) for density and MIN for distance.

Table 1: The table shows the clustering quality in terms of Rand Index of *sparse density peaks* vs. some state-of-the-art clustering algorithms using several common distance measures.

Dataset	Jaccard Distance				L1 (Manhattan) Distance			L2 (Euclidean) Distance		
	DP <i>approx.</i> (DP <i>approx.</i> <small>no correction</small>)	DP _{exact}	K Medoid	Hierarchical	DP _{exact}	K Means	Hierarchical	DP _{exact}	K Means	Hierarchical
Classic	0.81 (0.71)	0.83	0.82	0.82	0.63	0.71	0.78	0.77	0.71	0.77
last.fm	0.71 (0.69)	0.71	0.71	0.68	0.57	0.65	0.63	0.57	0.52	0.55
MovieLens	0.72 (0.65)	0.72	0.56	0.62	0.57	0.55	0.45	0.55	0.55	0.42

5 EVALUATIONS

In this section, we conduct both the quantitative experiments on the performance of the clustering pipeline and the qualitative visual analysis of the clustering results. Our goal is to demonstrate:

- (1) Our sketching-based approach to computing density peaks achieves a close approximation to the original result.
- (2) Our pipeline contributes meaningful performance improvement. Altogether, we achieve an almost 20× speed up and maintain a low memory environment.
- (3) The improved scalability enables a new visual analytics system on real industrial datasets. Our marketing experts are able to discover new customer segments using the functionalities provided by the system.

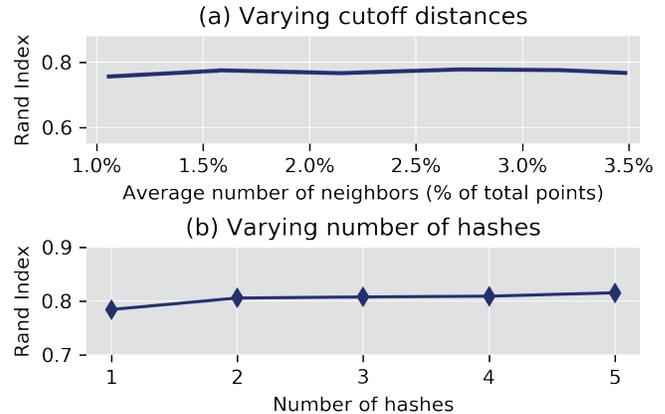
Dataset. For evaluating scalability, we experiment on two public online customer behavior datasets. The first dataset Instacart includes 3019354 visitors with 49677 unique purchase attributes. The second dataset Retailrocket includes 1407580 visitors with 270766 shopping behavior attributes. For evaluating clustering quality, we experiment on sparse binary datasets with ground truth group labels. The first dataset Classic consists of 7095 documents and 6134 unique words with four document categories. The second dataset last.fm consists of 947 users with 13413 songs listened to. We label the users with the most listened genre of songs (9 genres in total). The final dataset MovieLens consists of 1000 users with 4465 unique ratings on 1700 movies. We label the users with the most highly rated genre of movies (18 genres in total).

Experimental setup. We report results on a Hadoop cluster with 10 nodes and each node has four AMD Opteron 6276 2.3 GHz CPUs with 256GB of RAM. We report runtime averages from multiple trials. Without any specifications, our LSH hash tables contain 4 bands with 1 MinHash signature in each band (i.e., 4 hash tables) and we join users with at least 1 LSH bucket collision in both density and distance calculations. We fix the cutoff distance of DP to 0.5.

5.1 Quality

In this section, we evaluate the clustering quality through comparative experiments against alternatives and assess the trade-offs of resources for core pipeline parameters. We use the Rand Index [35] to measure the accuracy of clusters with ground truth labels.

Comparison against alternatives. We compare the clustering quality of our algorithm to several state-of-the-art clustering algorithms with careful tuning of parameters under weighted Jaccard distance and two common distance measures in three datasets. DP *approx.* and DP *approx.* no correction are Density Peaks that use MinHash

**Figure 7: Effect on varying (a) cutoff distance d_c and; (b) number of hashes on accuracy.**

and LSH to approximate the attributes with and without the corrections from Equation 7 and 8 respectively. DP_{exact} are Density Peaks that use brute force approaches to obtain the exact values. We can see that the clustering quality after rectifying the density and distance attributes becomes much closer to the brute force results. Also, we demonstrate the advantage of using set similarity measures in clustering sparse binary data. Noted that we do not claim that DP is superior to other state-of-the-art clustering algorithms in terms of accuracy. Rather, we want to show that it is competitive with those algorithms while having the ability to provide real-time responses with a relatively lower cost of pre-processing, which justifies our choice of its use.

Effect of pipeline parameters. To demonstrate that the accuracy of *sparse density peaks* is not sensitive to the parameter choices, we use the Classic dataset to generate four clusters with varying cutoff distance d_c and measure the Rand Index obtained. As shown in Figure 7(a), the choices of d_c is very wide which corroborates the original density peaks results [37].

On the other hand, we vary the number of hashes and measure the Rand Index obtained. As shown in Figure 7(b), we can achieve a stable accuracy with varied numbers of hashes. The reason is that although a small number of hashes will lower the accuracy for the attributes computed from the query, the correction stage will accommodate these points to generate a closer approximation.

End-to-end evaluation. We report the run time breakdown of the baseline LSH implementation of the pipeline and the effects of salting optimizations. We use the Retailrocket 1.4 million customers to test the runtime on density and distance computation.

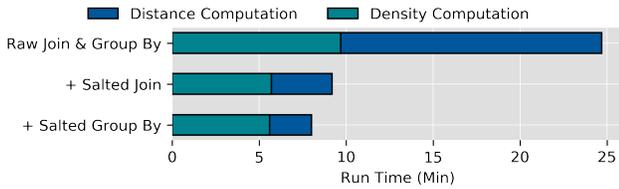


Figure 8: Effect of salting in computing density and distance.

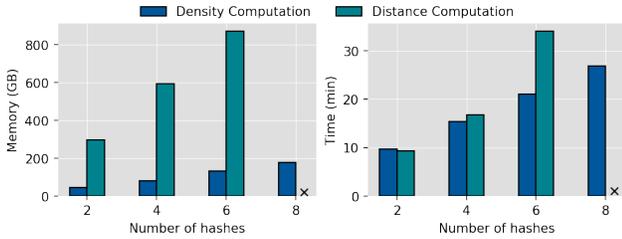


Figure 9: Effect of LSH parameter on similarity join runtime and memory consumption.

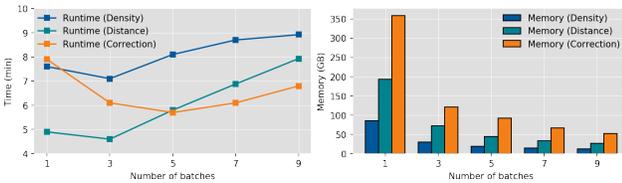


Figure 10: Effect of batch processing on similarity join runtime and memory consumption.

Figure 8 shows the cumulative runtime after applying each optimization. In general, it scales well and removes 60% of the time compared with using pure LSH. It shows that shuffled data are more evenly spread and are less likely to have the whole cluster waiting for few numbers of partition to complete.

5.2 Scalability

In this section, we evaluate the effect of trade-offs between memory and run time and demonstrate that there is an optimal setup between one-off processing and batch processing. Also, we evaluate against alternative set similarity search algorithms.

LSH parameters. We report the effect of runtime and memory usage of various numbers of hashes in Figure 9. We compute all data from Instacart (i.e., 3 million customers) in a single batch. We set a timeout of 2 hours. It can be seen that the memory consumption scales linearly for both computations, while the runtime of distance computation grows more drastically. The reason is related to the optimization of Spark. Spark tries to hold the shuffled data within the memory of each partition but when the memory is full, it transfers the remaining data to hard disk which causes a highly I/O cost. Also, while the chances of the join between points with larger distance are enabled, the number of duplicated join between similar points will increase as well, causing larger shuffled data.

Table 2: Index + Query Time for LSH and set similarity joins. Min-hash LSH incurs a 6.29% false negative rate while enabling up to 19.6× speedup.

	Algorithm	Index + Query (s)	Speedup
DP _{approx.}	MinHash LSH	31.8	-
	Mlib LSH	186.6	5.88×
DP _{exact}	VernicaJoin [41]	308	9.69×
	MRGroupJoin [11]	89.831	2.83×
HIERARCHICAL	AllPairs [3]	633	19.6×

Batch processing. We report the effect of runtime and memory usage of varying the number of batches in Figure 10, using the same data from Instacart. We can observe that splitting the data into different batches provides a remedy for the challenges shown in the above experiment of increasing LSH parameters. The memory usage can be decreased drastically by increasing the number of batches. Although increasing the number of batches to a further extent will linearly increase the runtime since it increases the number of shuffle stages, we demonstrate that there exists an equilibrium between increasing the number of shuffle stages and the memory usage.

Comparison against alternatives. We evaluate against alternative similarity computation algorithms that help generate a cluster hierarchy in Density Peaks and hierarchical clustering using Retailrocket data. We compare the query performance of our MinHash LSH to 1) an open source Spark Mlib LSH library² 2) two state-of-the-art MapReduce exact similarity join algorithms: VerincaJoin [41] and MRGroupJoin [11] 3) all pairs similarity join in MapReduce [3]. We set the similarity to 0.5 and the number of hashes to 4 which results in a 6.29% false-negative rate compared with exact similarity joins. In return, our MinHash LSH enables 5.88× speedup to Mlib LSH and 2.83× and 9.69× speedup against exact similarity join. Also, our choices of using density peaks allow a 19.6× speedup compared with using Hierarchical clustering as the real-time clustering solution. These results show that LSH are more effective than pruning strategies that leverage the similarity threshold when the threshold is moderate and MinHash signatures perform better than general LSH on sparse and binary data [40].

5.3 Qualitative Evaluation

Our accelerated clustering algorithm opens new opportunities for developing interactive tools for audience management at scale. For the use case of digital marketing, we propose a system to help marketing analysts cluster online visitors. Our development of the user interface followed an iterative, user-centered design process [32]. We iteratively refined the design through demos, discussions, and semi-structured interviews with marketers who analyze visitor behavior logs for large online retail stores. Based on the feedback, we identified three key design needs: (1) controls for trying our different clustering parameters, (2) visualizations for interpreting a group, and (3) support for re-splitting a group. The user interface of the web application is shown in Figure 11.

²<https://spark.apache.org/mllib/>

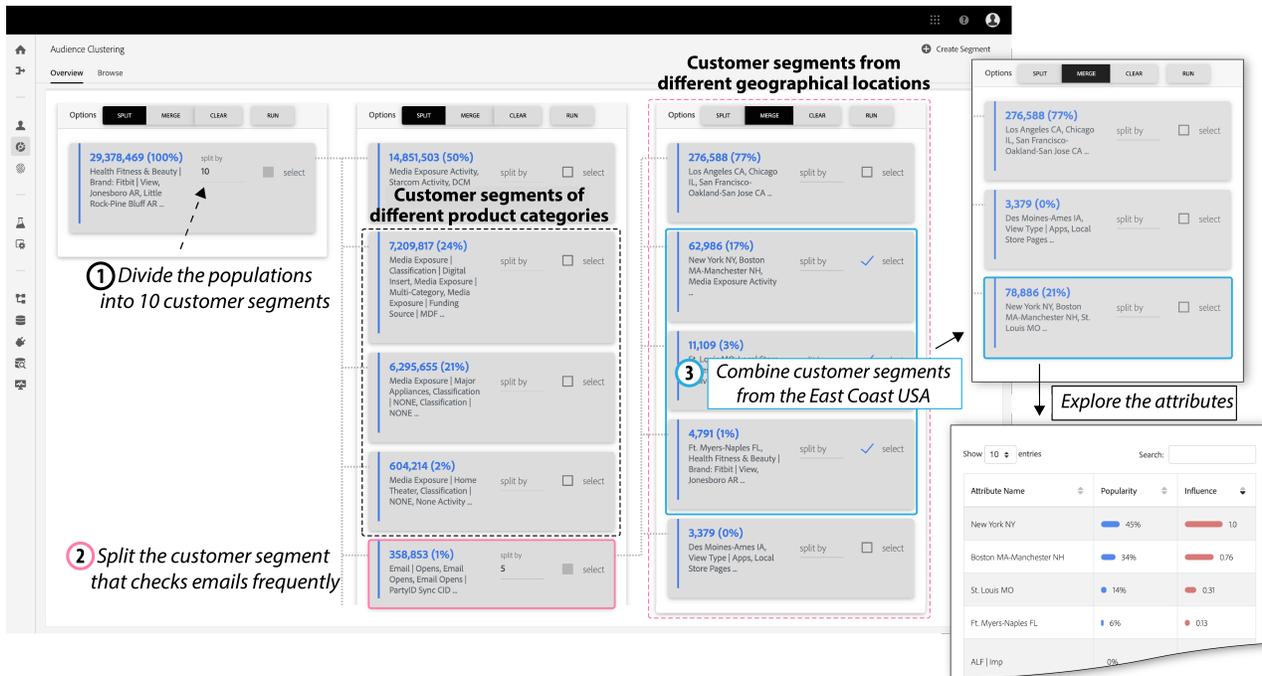


Figure 11: Proposed user interface providing (1) controls for clustering parameters, (2) further split of clusters, and (3) support for combining clusters and visualizations for interpreting a cluster. This figure illustrates a synthetic online visitor dataset.

To qualitatively evaluate the effectiveness of our system, we conducted a case study with two analysts who worked at a large company that provides digital marketing services, in which they used our system to explore a real dataset of over 30 million online visitors. First, the algorithm produced the groups by the analysts’ specifications and visually presented the population and user attribute information of each group. By trying different customer segmentation results by specifying a different number of clusters (Figure 11(1)), the analysts quickly discovered a meaningful cluster result that segmented online customers by three interest groups on specific purchasing product categories (i.e. multi-category, major appliances, and home theater). Furthermore, the analysts inspected each group’s popular attributes, influential attributes, and distinguishing attributes compared to its parent group. From the results, they discovered that some attributes within each group were very diverse. Thus they increased the number of clusters to get more fine-grained similar groups. For example, the analysts selected a group of customers who were labeled as “frequent email openers” for further analysis (Figure 11(2)). They further split this group and found that the influential attributes in the sub-groups were more geographically related, which were useful to run regional email campaigns. They merged three sub-groups that contain customers on the east coast and saved the resulted 78,000 visitors for sending region aware marketing campaign (Figure 11(3)).

In summary, our proposed system allows analysts to quickly experiment with different settings and identify groups of interest from clustering results. We believe such user experience will inspire new audience management technologies and commercial solutions that were not possible before.

6 CONCLUSION

In this paper, we have proposed a novel clustering algorithm *sparse density peaks* for real-time cluster exploration on sparse binary data. Our approach constructs a cluster assignment hierarchy for data efficiently by leveraging MinHash and LSH in Spark distributed computing framework. While directly applying LSH will miss the true nearest neighbors and provide a biased estimation to the attributes of density peaks, we have provided more accurate estimation measures and an additional correction step to provide a better approximation. Also, a straightforward implementation of LSH will fail to scale beyond an increased size of data and an increased number of hashes. Thereby, we have introduced efficient salting strategies to ensure even parallelism of data in the distributed platform. To demonstrate the application of the approach in real-life scenarios, we have designed a user interface that utilizes the results to address real-life business decision making. We believe these results will benefit data scientists to design an end-to-end clustering pipeline for exploring industry scale online user behavior data.

ACKNOWLEDGMENTS

Cláudio Silva and Juliana Freire are partially supported by the DARPA D3M program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA. The experimental evaluation reported in this paper was carried out using equipment acquired with the NSF award CNS-1229185.

REFERENCES

- [1] Domenica Arlia and Massimo Coppola. 2001. Experiments in parallel clustering with DBSCAN. In *European Conference on Parallel Processing*. Springer, 326–331.
- [2] SatyaJaswanth Badri. 2019. A novel Map-Scan-Reduce based density peaks clustering and privacy protection approach for large datasets. *International Journal of Computers and Applications* (2019), 1–11.
- [3] Roberto J Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up all pairs similarity search. In *Proceedings of the 16th international conference on World Wide Web*. ACM, 131–140.
- [4] Pavel Berkhin. 2006. A survey of clustering data mining techniques. In *Grouping multidimensional data*. Springer, 25–71.
- [5] Rongfang Bie, Rashid Mehmood, Shanshan Ruan, Yunchuan Sun, and Hussain Dawood. 2016. Adaptive fuzzy clustering by fast search and find of density peaks. *Personal and Ubiquitous Computing* 20, 5 (2016), 785–793.
- [6] Christian Böhm, Robert Noll, Claudia Plant, and Bianca Wackersreuther. 2009. Density-based clustering using graphics processors. In *Proceedings of the 18th ACM conference on Information and knowledge management*. ACM, 661–670.
- [7] Marco Cavallo and Çağatay Demiralp. 2018. Clustrophile 2: guided visual clustering analysis. *IEEE transactions on visualization and computer graphics* 25, 1 (2018), 267–276.
- [8] Gromit Yeuk-Yin Chan, Panpan Xu, Zeng Dai, and Liu Ren. 2018. ViBr: Visualizing Bipartite Relations at Scale with the Minimum Description Length Principle. *IEEE transactions on visualization and computer graphics* 25, 1 (2018), 321–330.
- [9] Xiaojun Chen, Yixiang Fang, Min Yang, Feiping Nie, Zhou Zhao, and Joshua Zhexue Huang. 2017. Purtreeclust: A clustering algorithm for customer segmentation from massive customer transaction data. *IEEE Transactions on Knowledge and Data Engineering* 30, 3 (2017), 559–572.
- [10] Ondrej Chum, James Philbin, Andrew Zisserman, et al. 2008. Near duplicate image detection: min-hash and tf-idf weighting. In *Bmvc*, Vol. 810. 812–815.
- [11] Dong Deng, Guoliang Li, He Wen, and Jianhua Feng. 2015. An efficient partition based method for exact set similarity joins. *Proceedings of the VLDB Endowment* 9, 4 (2015), 360–371.
- [12] Peter Deuffhard and Andreas Hohmann. 2003. *Numerical analysis in modern scientific computing: an introduction*. Springer.
- [13] Inderjit S Dhillon and Dharmendra S Modha. 2002. A data-clustering algorithm on distributed memory multiprocessors. In *Large-scale parallel data mining*. Springer, 245–260.
- [14] Fan Du, Catherine Plaisant, Neil Spring, and Ben Shneiderman. 2018. Visual interfaces for recommendation systems: Finding similar and dissimilar peers. *ACM Transactions on Intelligent Systems and Technology (TIST)* 10, 1 (2018), 9.
- [15] D Foti, D Lipari, Clara Pizzuti, and Domenico Talia. 2000. Scalable parallel clustering for data mining on multicompilers. In *International Parallel and Distributed Processing Symposium*. Springer, 390–398.
- [16] Claudio Gentile, Shuai Li, Purushottam Kar, Alexandros Karatzoglou, Giovanni Zappella, and Evans Etrue. 2017. On context-dependent clustering of bandits. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 1253–1262.
- [17] Aristides Gionis, Piotr Indyk, Rajevee Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *Vldb*, Vol. 99. 518–529.
- [18] Michael Greenwald, Sanjeev Khanna, et al. 2001. Space-efficient online computation of quantile summaries. *ACM SIGMOD Record* 30, 2 (2001), 58–66.
- [19] Jeffrey Heer and Maneesh Agrawala. 2008. Design considerations for collaborative visual analytics. *Information visualization* 7, 1 (2008), 49–62.
- [20] James Hendler. 1992. *Artificial intelligence planning systems: proceedings of the first international conference, June 15-17, 1992, College Park, Maryland*. Morgan Kaufmann.
- [21] Dong Hyun Jeong, Caroline Ziemkiewicz, Brian Fisher, William Ribarsky, and Remco Chang. 2009. iPCA: An Interactive System for PCA-based Visual Analytics. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 767–774.
- [22] Daniel Keim, Gennady Andrienko, Jean-Daniel Fekete, Carsten Görg, Jörn Kohlhammer, and Guy Melançon. 2008. Visual analytics: Definition, process, and challenges. In *Information visualization*. Springer, 154–175.
- [23] Daniel A Keim, Florian Mansmann, Jörn Schneidewind, Jim Thomas, and Hartmut Ziegler. 2008. Visual analytics: Scope and challenges. In *Visual data mining*. Springer, 76–90.
- [24] Nathan Korda, Balázs Szörényi, and Li Shuai. 2016. Distributed clustering of linear bandits in peer to peer networks. In *Journal of machine learning research workshop and conference proceedings*, Vol. 48. International Machine Learning Society, 1301–1309.
- [25] Bum Chul Kwon, Ben Eysenbach, Janu Verma, Kenney Ng, Christopher De Filippi, Walter F Stewart, and Adam Perer. 2017. Clustervision: Visual supervision of unsupervised clustering. *IEEE transactions on visualization and computer graphics* 24, 1 (2017), 142–151.
- [26] Hanseung Lee, Jaeyeon Kihm, Jaegul Choo, John Stasko, and Haesun Park. 2012. iVisClustering: An interactive visual document clustering via topic modeling. In *Computer graphics forum*, Vol. 31. Wiley Online Library, 1155–1164.
- [27] Shuai Li, Alexandros Karatzoglou, and Claudio Gentile. 2016. Collaborative filtering bandits. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*. ACM, 539–548.
- [28] Zhicheng Liu and Jeffrey Heer. 2014. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2122–2131.
- [29] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28, 2 (1982), 129–137.
- [30] Rashid Mehmood, Saeed El-Ashram, Rongfang Bie, Hussain Dawood, and Anton Kos. 2017. Clustering by fast search and merge of local density peaks for gene expression microarray data. *Scientific reports* 7 (2017), 45602.
- [31] Rashid Mehmood, Guangzhi Zhang, Rongfang Bie, Hassan Dawood, and Haseeb Ahmad. 2016. Clustering by fast search and find of density peaks via heat diffusion. *Neurocomputing* 208 (2016), 210–217.
- [32] Tamara Munzner. 2009. A nested process model for visualization design and validation. *IEEE Transactions on Visualization and Computer Graphics* 6 (2009), 921–928.
- [33] Dan Pelleg, Andrew W Moore, et al. 2000. X-means: Extending k-means with efficient estimation of the number of clusters. In *ICML*, Vol. 1. 727–734.
- [34] Anand Rajaraman and Jeffrey David Ullman. 2011. *Chapter 3, Mining of massive datasets*. Cambridge University Press.
- [35] William M Rand. 1971. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association* 66, 336 (1971), 846–850.
- [36] Chandan K Reddy and Bhanukiran Vinzamuri. 2018. A survey of partitionial and hierarchical clustering algorithms. In *Data Clustering*. Chapman and Hall/CRC, 87–110.
- [37] Alex Rodriguez and Alessandro Laio. 2014. Clustering by fast search and find of density peaks. *Science* 344, 6191 (2014), 1492–1496.
- [38] Tobias Schreck, Jürgen Bernard, Tatiana Von Landesberger, and Jörn Kohlhammer. 2009. Visual cluster analysis of trajectory data with interactive kohonen maps. *Information Visualization* 8, 1 (2009), 14–29.
- [39] Yong Shi, Zhensong Chen, Zhiquan Qi, Fan Meng, and Limeng Cui. 2017. A novel clustering-based image segmentation via density peaks algorithm with mid-level feature. *Neural Computing and Applications* 28, 1 (2017), 29–39.
- [40] Anshumali Shrivastava and Ping Li. 2014. In defense of minhash over simhash. In *Artificial Intelligence and Statistics*. 886–894.
- [41] Rares Vernica, Michael J Carey, and Chen Li. 2010. Efficient parallel set-similarity joins using MapReduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 495–506.
- [42] Joe H Ward Jr. 1963. Hierarchical grouping to optimize an objective function. *J. Amer. Statist. Assoc.* 58, 301 (1963), 236–244.
- [43] John Wenskovitch, Ian Crandell, Naren Ramakrishnan, Leanna House, and Chris North. 2017. Towards a systematic combination of dimension reduction and clustering in visual analytics. *IEEE transactions on visualization and computer graphics* 24, 1 (2017), 131–141.
- [44] Yanfeng Zhang, Shimin Chen, and Ge Yu. 2016. Efficient distributed density peaks for clustering large data sets in mapreduce. *IEEE Transactions on Knowledge and Data Engineering* 28, 12 (2016), 3218–3230.
- [45] Hong Zhao, Tao Wang, and Xiangyan Zeng. 2018. A clustering algorithm for key frame extraction based on density peak. *Journal of Computer and Communications* 6, 12 (2018), 118–128.
- [46] Weizhong Zhao, Huifang Ma, and Qing He. 2009. Parallel k-means clustering based on mapreduce. In *IEEE International Conference on Cloud Computing*. Springer, 674–679.