Exploring Traditional Machine Learning Clustering Techniques for Big Data

Peerakarn Jitpukdee
School of Physics and Astronomy
University of Nottingham
Email: ppxpj2@nottingham.ac.uk

Ahmed Aman Ibrahim
School of Physics and Astronomy
University of Nottingham
Email: ppxai1@nottingham.ac.uk

Bala Krishnan Sekar
School of Physics and Astronomy
University of Nottingham
Email: ppxbs4@nottingham.ac.uk

Deepak Kumar
School of Physics and Astronomy
University of Nottingham
Email: ppxdc3@nottingham.ac.uk

Jenni Anna Mathew
School of Physics and Astronomy
University of Nottingham
Email: ppxjm5@nottingham.ac.uk

Arvin Corotana
School of Physics and Astronomy
University of Nottingham
Email: ppxac8@nottingham.ac.uk

***Abstract*—Utilizing appropriate algorithms for specific data types is crucial in extracting meaningful insights from data mining and analysis endeavors. For instance, when dealing with mixed data types comprising both numerical and categorical features in datasets, selecting suitable algorithms like k-prototype becomes paramount. However, the full potential of big data remains underutilized due to the absence of implementations tailored for this context. Leveraging the optimization capabilities of Apache Spark, we aim to bridge this gap by implementing well-known clustering algorithms within the PySpark framework. Our investigation will focus on parallelizing k-means, k-mode, k-prototype,DBSCAN algorithms, exploring both local and global implementations and experiments are performed to see how the algorithm scales-up and to analyse its run-time and the convergence of global approach to the sequential approach. Evaluation of our approach will be conducted using diverse datasets sourced from both the UCI repository and Kaggle.**

1. Introduction

In the era of big data, clustering plays a crucial role in data analysis, understanding, and insight discovery, leading to various further applications. Among many approaches, k-means clustering remains the most popular clustering algorithm [1]. However, k-means only deals with numerical data, while for categorical and mixed data types, other algorithms must be explored.

Clustering tasks dominated by categorical variables often face computational challenges when using k-means with one-hot encoded representations, mainly because it generates a large number of variables, especially in datasets with numerous unique values within each categorical variable. Additionally, employing the distance metric between data points and centroids within this transformed space may lack intuitive justification, particularly in datasets rich in categorical features. In contrast, the Kmode algorithm offers a pragmatic alternative, quantifying dissimilarity between a data point and its centroid using the hamming distance, which aligns naturally with categorical data structures [2]. Unlike k-means, k-mode updates cluster centroids by selecting the mode rather than the mean. Despite its benefits, an official implementation of k-mode for Spark remains absent, motivating our investigation into strategies for parallelizing k-mode, employing map-reduce framework [3], to enhance its scalability and efficiency within distributed computing environments, where we explore both global and local parallelization approaches.

We also explore the k-prototype clustering algorithm, first introduced in Huang's seminal work on clustering mixed-type data [4]. The k-prototype algorithm is specifically designed for datasets containing both numerical and categorical variables and has been successfully implemented across various domains. For example, it has been used in customer segmentation in retail [5], healthcare analytics [6], and fraud detection in financial transactions [7]. The algorithm combines elements of both k-means clustering and the k-mode algorithm, employing a distance metric that considers both numerical and categorical features. We explore both local and global approaches for this algorithm and benchmark its performance on a dataset of 100k data points, the Kaggle Financial Transactions dataset [8], evaluating the impact of different partition sizes on Resilient Distributed Datasets (RDDs). In our quest to optimize clustering algorithms for big data, we have drawn inspiration and valuable insights from a series of influential research papers, including those by Lian et al. [9] (2023), Yang et al. (2023) [10], Li (2021) [11], and Ajin and Kumar (2016) [12].

Another approach to clustering, with a specified number of clusters (k), is the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [13]. This method is considered more robust to outliers because it can classify data points as noise rather than forcing them into a cluster. This is achieved by defining a distance metric, epsilon (ϵ), to determine whether a data point is close enough to be part of the same cluster. Additionally, a minimum number of data points is specified to classify a data point as a core data point, which has the ability to grow the cluster if another point is within the epsilon distance. Data points near a border point but not near a core point are classified as noise points. Similar to k-means, k-mode, and k-prototype, there is no native implementation of DBSCAN within the Spark package. However, with the increasing prevalence of big data and the necessity to glean insights from it, the implementation of DBSCAN and other unsupervised learning algorithms can significantly enhance our understanding of complex data structures.

Furthermore, as we covered the global implementation of k-means in our Big Data lecture, leaving the local implementation as a challenge to explore, we decided to investigate further about the local implementation of K-means.

2. Methodology

2.1. k-means

2.1.1. Local approach. In the local approach, we implement a parallelized strategy where the k-means algorithm learns from each data partition simultaneously. This approach divides the full dataset into N partitions and selects k as the number of clusters. Consequently, the mapping phase generates Nxk centroids as output, where each centroid represents the mode of clusters within its respective partition. These centroids are then transmitted to the driver.

Upon receiving these centroids, which collectively approximate the behavior of the sequential algorithm, the driver performs the final clustering step. It treats these centroids as a dataset and applies the k-means algorithm to derive the final centroids for K clusters.

2.2. k-mode

2.2.1. Local approach. The local approach for k-mode clustering is almost identical to the local approach for k-means, except we use the k-mode algorithm instead of k-means.

2.2.2. Global approach. In our approach to cluster data into K groups using the k-mode algorithm, we initially initialize K number centroids by randomly sampling K data points as suggested in the original work [2]. These centroids are represented as vectors, denoted as $\mathbf{c^{(k)}}$ for $k = 1, 2, ..., K$. In the sequential implementation, we iteratively calculate the Hamming distance between each data point $\mathbf{x^{(i)}}$ and centroids $\mathbf{c^{(k)}}$, which is defined as:

$$D(\mathbf{x^{(i)}}, \mathbf{c^{(k)}}) = \sum_{j}^{P} \delta(x_j^{(i)}, c_j^{(k)}) \qquad (1)$$

where P is the number of features, $x_j^{(i)}$ denotes the value of feature j for data point i, $c_j^{(k)}$ is the value of feature j for centroid k, and $\delta(a, b)$ is the Kronecker delta function, which equals 1 if $a = b$ and 0 otherwise. We assign each data point to its closest centroid with the smallest mismatch. Subsequently, new centroids are derived by computing the mode values of each feature within their respective clusters. This iterative process continues until convergence, where the new centroids remain unchanged.

To parallelize the Kmode algorithm effectively, we utilize a map-reduce framework. In each iteration, the centroids $c^{(k)}$ are broadcast to the worker nodes. During the mapping phase, the hamming distance between each centroid and data point is computed. The centroid with the least mismatch with the data point is designated as the key, while the data point is hashed into a suitable format for efficient processing in the reduce phase. Specifically, each datapoint $x^{(i)} = [s_1^{(i)}, s_2^{(i)}, ..., s_M^{(i)}]$ is hashed into $h^{(i)} = \{1 : \{s_1^{(i)} : 1\}, 2 : \{s_2^{(i)} : 1\}, ..., P : \{s_P^{(i)} : 1\}\}$ format. This hashing facilitates the counting of occurrences of each element value $s_1, s_2, ..., s_P$ in each cluster during the combine phase and reduce phase, simplifying the calculation of the mode. Consequently, there is no need to move the entire dataset to the driver code for mode calculation, thereby minimizing memory consumption and data movement.

During both combine and reduce phase, the counts of hashed data within the same cluster (with the same centroid as the key) are aggregated using a reducer function. Below is the pseudo code for the $mergeCountHashed$ reducer function.

Algorithm 1 mergeCountHashed

```
1: A = { p: { $a_i : n_i$ for i = 1, 2, ..., N} for p = 1, 2, ..., P }
2: B = { p: { $b_j : m_j$ for j = 1, 2, ..., M} for p = 1, 2, ..., P }
3: function MERGECOUNTHASHED(A, B)
4:     for $p = 1, 2, ..., P$ do
5:         for $keyB$ in A[p].keys() do
6:             if $keyB\ in\ keys\ of\ A[p]$ then
7:                 $A[p][keyB] += B[p][keyB]$
8:             else
9:                 $A[p][keyB] = B[p][keyB]$
10:            end if
11:        end for
12:    end for
13:    return A
14: end function
```

After computing the merged count hashed data for each cluster, represented as (cluster, merged count-hash) pairs, they are transmitted to the driver code. Subsequently, the new centroid (mode) for each cluster can be easily determined using the following approach:

Given H_A, which represents the merged count-hash data for cluster A obtained after the reduce phase, where $H_A = \{p : \{a_i : n_i\ for\ i = 1, 2, 3, ..., N_A\}\ for\ p = 1, 2, 3, ..., P\}$

In H_A, p represents the individual features (or dimensions) within the dataset, a_i represents a specific element value within feature p, and n_i represents the count of occurrences of a_i within the cluster. These counts are aggregated during the reduce phase to facilitate centroid calculation.

Then, the centroid $c^{(A)}$, for cluster A, is computed as an array containing the argument of maximum count values n_i for each feature: $c^{(A)} = \langle argmax_a \ (H_A[p]) \rangle$

These updated centroids are then broadcast to the worker nodes for the subsequent iteration. This iterative process continues until convergence, as determined by the Hamming distance between the new centroids and the old ones reaching the stopping criterion.

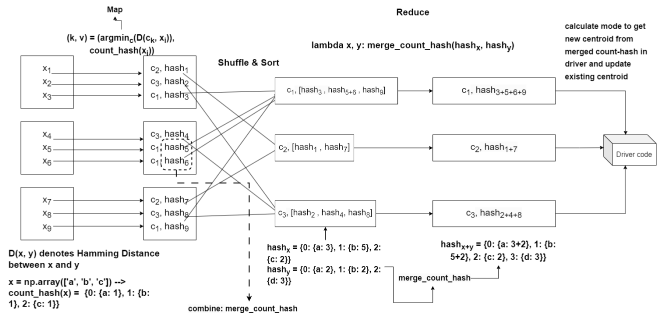
Map
(k, v) = (argmin_c(D(c_k, x_i)), count_hash(x_i))
Reduce
lambda x, y: merge_count_hash(hash_x, hash_y)
Shuffle & Sort
calculate mode to get new centroid from merged count-hash in driver and update existing centroid
x1
x2
x3
x4
x5
x6
x7
x8
x9
c2, hash1
c3, hash2
c1, hash3
c3, hash4
c1, hash5
c1, hash6
c2, hash7
c3, hash8
c1, hash9
c1, [hash3 , hash5+6 , hash9]
c2, [hash1 , hash7]
c3, [hash2 , hash4, hash8]
c1, hash3+5+6+9
c2, hash1+7
c3, hash2+4+8
Driver code
D(x, y) denotes Hamming Distance between x and y
x = np.array(['a', 'b', 'c']) --> count_hash(x) = {0: {a: 1}, 1: {b: 1}, 2: {c: 1}}
combine: merge_count_hash
hash_x = {0: {a: 3}, 1: {b: 5}, 2: {c: 2}}
hash_y = {0: {a: 2}, 1: {b: 2}, 2: {d: 3}}
merge_count_hash
hash_x+y = {0: {a: 3+2}, 1: {b: 5+2}, 2: {c: 2}, 3: {d: 3}}

Figure 1. Map-Reduce process illustrating each iteration in the global algorithm of K-mode clustering

2.3. k-prototype

The k-prototype algorithm is a fusion of the k-means and k-mode algorithms, providing an efficient means to cluster mixed data types that encompass both categorical and numerical features. Initially, k centroids are randomly selected from the dataset. Subsequently, the algorithm segregates the categorical and numerical data to evaluate their distances independently. Then, these distances are amalgamated to assign each data point to the nearest cluster. Following this, a new centroid is recalculated based on the updated clusters, and this process iterates until convergence is achieved. We implemented this algorithm as described in [4]. We evaluate the algorithm on the Kaggle Retail Transactions Data which have 100k data points.

2.3.1. Local approach. In the local approach, we train a k-prototype model on each partition of the dataset. A k-prototype model is trained on all the partitions in parallel. If there is p number of partitions, we would have p k-prototype models. Finally, to aggregate all the results, in the reduce phase, we train a final k-prototype model on all the centroids from other models (use the p*k centroids as a dataset). This is an approximation for the actual k-prototype algorithm.

2.3.2. Global approach. In the global k-prototype algorithm, we use map-reduce to parallelize the k-prototype algorithm. First, we select k random points from the dataset to be centroids and broadcast them. A distance function which is a combination of both the Euclidean distance and Hamming distance is mapped to all the data points. We then use the numpy.argmin() function to find which centroids are closest to each data point. We then create an RDD which has the form (centroid, (1, numerical_feature)) for numerical variables. We then use reduceByKey function to find the average of all numerical features to be used for the new centroids. For categorical variables, we create another RDD of the form (centroid, categorical_feature). We then find the mode of each categorical feature column for each centroid. Finally, we join the numerical and categorical centroids to become the new centroids, and repeat the distance to data points calculations until the centroids stop moving. The global approach to k-prototype is the exact algorithm.

2.4. DBSCAN

2.4.1. A Local Approach. A parallelised approach was taken to process and perform DBSCAN on the dataset, prompting the partitioning of the data to a number specified in the search algorithm chosen. The process as a whole leverages a two-step process combining spatial data partitioning with clustering. This methodology is crucial for managing large datasets efficiently, especially in datasets like those in geographic information systems. The recursive KD-tree method [14] was selected for the purpose of creating partitions of the dataset in a dataframe structure, with a max_depth of 3, 8 partitions were formed. This partitioning is based on alternating splits between longitude and latitude, ensuring that datapoints that are closer are likely to end up in the same or neighbouring partitions. This helps in reducing the computation complexity by localising the datapoints which are likely to be in the same clusters. Other methods to perform this partitioning were tested leading to imbalanced partitions and poorer performance leading to KD-tree being chosen over Quadtree and Adaptive grid partitioning [15]. This resulted in the formation of an RDD. Once the data is partitioned, it is scaled using standardscaler and DBSCAN is applied to each partition independently with specific parameters for epsilon (the maximum distance between two samples to be considered as in the same neighborhood) and min samples (the minimum number of samples in a neighborhood for a point to be considered as a core point).

To achieve global clustering that spans across partitions, we identified border points in each partition by calculating the minimum and maximum values for the latitude and longitude. Points close to these boundaries, within a specified margin, were tagged as border points. To adapt the clustering parameters dynamically, we determined epsilon by calculating the average distance to the second nearest neighbor among the border points.The dataset was then converted from a Spark DataFrame to a Pandas DataFrame, facilitating the use of a KD-tree for efficient querying of nearby points. By querying pairs of border points within the dynamically determined epsilon, we formed edges between clusters, indicating potential connections.

A graph was subsequently built based on these edges, identifying clusters that were close enough to consider merging. The merging process integrated these clusters across partitions, thus forming global clusters that extend beyond the confines of individual partitions. This approach allowed us to maintain continuity and consistency in cluster characteristics across the dataset.

3. Experiments

3.1. k-means

Our experiment deals with numerical variables related to the magnetic fields detected from the soil [16], which enables to detect mines. This dataset was taken from the UCI machine learning repository then converted as a spark data frame for implementation of K-Means in pyspark. The K value was chosen as 3, elbow method was not used to optimise aiming for parallelisation instead. The iterations was limited to 10 and stopping is based on Euclidean distance between updated and old centroids. To assess scalability, we vary data proportions (25%, 50%, 75%, 100%) and observe runtime performance under different partition counts, ensuring fair comparison with consistent initial centroids.

3.2. k-mode

Our experiment focuses on clustering datasets predominantly composed of categorical features, suitable for the KMode algorithm. We analyze two datasets: one from UCL [17] and another from a Kaggle competition [18], both have majority of features to be categorical features. They both roughly have around 60k samples. We preprocess the data by dropping numerical columns for mushroom dataset and handle missing values by treating them as a separate category.

Additionally, for scalability, we incorporate the Kaggle 1M rows Turkish Market Sales Dataset [19], which includes 1 million rows. For this dataset, we select only four categorical variables as the others are redundant. We also categorize the price (a numerical variable) into categorical levels (low, mid, high, and expensive) based on quartiles.

As our focus is on parallelizing algorithms, we set K to 3 (K to 5 for turkish dataset) without utilizing the elbow method for determining the cluster count for global, local, and sequential implementations, enabling direct comparison within the same K. Additionally, we did not separate test and training sets, prioritizing parallelization, especially given the unsupervised nature of the algorithm. Moreover, we limit iterations to 10, with a stopping criterion based on the Hamming distance between updated and old centroids, set at 0.1 times the total number of features (P).

To assess scalability, we vary data proportions (25%, 50%, 75% and 100%) and observe runtime performance under different partition counts, ensuring fair comparison with consistent initial centroids.

3.3. k-prototype

We experimented with a clustering datasets which had both numerical and categorical variables. For preprocessing the numerical variables, we used min-max scaling. Also, we scaled the distance of categorical variables by $0.1\times$ the number of categorical features. For example, if there are 3 categorical features, we would scale it by $0.1 \times 3 = 0.3$. Finally, similar to k-mode, to assess scalability, we vary data proportions (25%, 50%, 75%, and 100%) to observe the runtime performance under different partition counts.

3.4. DBSCAN

The experimentation with the chosen methodology was implemented on a geospatial dataset. The dataset used was provided by Microsoft Research Asia, which includes GPS trajectories collected by different users and labelled with various modes of transportation [20]. The location, specified by longitude and latitude, was the feature/s of interest for clustering.

To assess the scalability of the methodology a series of runtime tests were carried out with differing amounts of datapoints sampled from the dataset. This was done with roughly 15, 30, 45 and 60 thousand datapoints. The number of partitions is determine by KD-tree meaning a variation of partitions was not experimented with.

4. Results and Discussion

4.1. k-means

While parallelizing the K-means clustering algorithm using Apache Spark may not yield significant scalability benefits for a small dataset of 338 data points, it is still essential to validate the correctness of the parallel implementation by comparing the final centroid obtained from it with the sequential execution. The final centroid in the provided code is computed by performing K-means clustering on the concatenated centroids from all partitions, which are calculated using the get optimal kmeans centroid function on each partition's data. Although the final centroids may not be identical due to the randomness involved in initialization and data processing order, they should be reasonably close if the implementation is correct. Comparing the final centroid with the sequential execution and ensuring the difference is within an acceptable threshold provides confidence in the parallel implementation's accuracy, serving as a foundation for handling larger datasets more efficiently in the future.

4.2. k-mode

4.2.1. Algorithm Approximation. With consistent configurations, initial centroids, and stopping criteria, both the global and sequential approaches produced identical final centroids for both the secondary mushroom and Kaggle categorical encoding datasets. This alignment was as expected, as the global approach effectively mirrors the sequential algorithm by parallelizing the counting of elements during the map-reduce phase and performing the mode calculation phase solely in the driver. However, the local approach, employing partition-specific centroid calculations, generated clusters that were almost identical but not exactly the same, due to its utilization of local initial centroids on each partition, leading to an approximation rather than an exact match with the sequential algorithm.

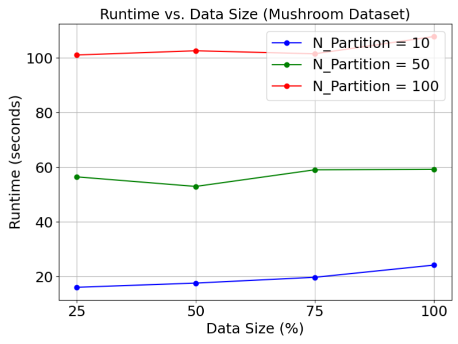

Figure 2. Mushroom Sizeup for K-Mode global approach

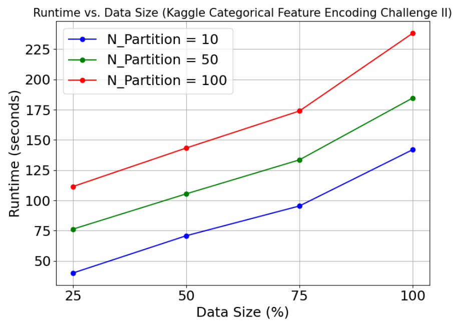

Figure 3. Kaggle Categorical Competition II Sizeup for K-Mode global approach

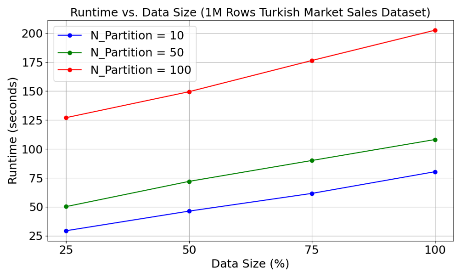

Figure 4. Kaggle 1M rows Turkish Market Sales sizeup for K-Mode global approach

4.2.2. Scalability. Our focus on the global approach, which mirrors the sequential algorithm, prompts an investigation into its scalability. Analyzing runtime behavior across datasets reveals differences. The Mushroom dataset exhibits consistent runtime even with scaling, while the Kaggle Categorical Feature Encoding Challenge dataset shows varied runtime with increasing data size. Notably, runtime increases with the number of partitions, especially pronounced with 100 partitions. Dataset characteristics likely contribute, with the Kaggle dataset's complexity impacting computational complexity. An inefficient implementation of the global approach may contribute to runtime increases, particularly during the reduce phase, where higher partition numbers lead to increased computational overhead and longer processing times, highlighting scalability challenges.

In contrast, our experiments on the 1M rows Turkish Market Sales Dataset demonstrate a more linear runtime increase with data size. However, the runtime still increases with the number of partitions. This linear scalability indicates that our implementation of the k-mode algorithm can efficiently handle large datasets under the global parallelization approach, with the runtime increasing proportionally to the data size, though still affected by the partition count.

4.3. k-prototype

4.3.1. Algorithmic Approximation.. The local version of k-prototype algorithm did not returned very different centroids when run for multiple iterations. This shows that the approach we used is not suitable when dealing with multiple k-prototype models. Therefore, we did the rest of teh experiments on the global approach.

4.3.2. Scalability. We trained the Kaggle Financial Transactions dataset which contains 2 numerical features and 2 categorical features, with a 100k data points, for 10 iterations. Figure 5 shows that the execution time for k-prototype algorithm increases linearly with dataset size. However, unexpectedly, more partitions result in a longer execution time. This could be either due to an inefficiency in our k-prototype implementation or the limited amount of CPU memory available in Google Colab where we ran the tests. More experiments on a more powerful distributed system are needed to confirm.

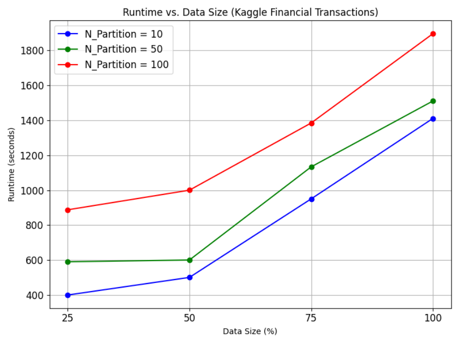

Figure 5. Kaggle Financial Transactions Data sizeup for K-Prototype global approach

4.4. DBSCAN

With differing levels of size in terms of samples used in the implementation we were able to measure the ability to scale. With approximately 15500 datapoints a run time

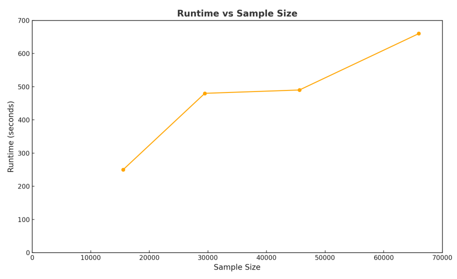

Figure 6. DBSCAN on Differing Samples Sizes of Geospatial Data

of 250 seconds was seen, which increased to 480 seconds when sample size jumped to around 29500. As expected, the runtime increased again for both sample sizes of roughly 45600 and 66000 with runtimes of 580 and 660 seconds respectively. The increase in runtime seems generally linear, seen in Figure 6, meaning that predicting the time required for a task is relatively simple. This is most likely due to the fact that with more data, there is likely more border points and more border point pair calculations, along with more cluster merges. Due to the number of partitions not scaling with the sample size of the dataset, there means there will be more calculations in parallel, which will lead to a longer compute time.To improve the confidence in higher levels of sample size this experiment should be implemented on a cloud based system as the memory allocation in Kaggle was a limitation.

5. Conclusion

This study parallelized traditional clustering algorithms (k-means, k-mode, k-prototype, DBSCAN) using Apache Spark, aiming to improve their scalability and efficiency on large datasets. The global k-mode approach maintained accuracy while exhibiting varying scalability, and k-means's local approach showed optimal performance at moderate fraction sizes and partition counts. K-prototype's scalability increased with dataset size but had longer execution times at higher partition counts, warranting further investigation. The local DBSCAN implementation showed promise but requires optimization and testing on larger datasets. Overall, this work paves the way for efficient unsupervised learning in distributed computing environments.

References

[1] Abiodun M Ikotun, Absalom E Ezugwu, Laith Abualigah, Belal Abuhaija, and Jia Heming. K-means clustering algorithms: A comprehensive review, variants analysis, and advances in the era of big data. *Information Sciences*, 622:178–210, 2023.

[2] Zhexue Huang. A fast clustering algorithm to cluster very large categorical data sets in data mining. *Dmkd*, 3(8):34–39, 1997.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, jan 2008.

[4] Zhexue Huang et al. Clustering large data sets with mixed numeric and categorical values. In *Proceedings of the 1st pacific-asia conference on knowledge discovery and data mining,(PAKDD)*, pages 21–34. Citeseer, 1997.

[5] Jason Soria, Ying Chen, and Amanda Stathopoulos. K-prototypes segmentation analysis on large-scale ridesourcing trip data. *Transportation Research Record*, 2674(9):383–394, 2020.

[6] Florin C Ghesu, Bogdan Georgescu, Awais Mansoor, Youngjin Yoo, Dominik Neumann, Pragneshkumar Patel, Reddappagari Suryanarayana Vishwanath, James M Balter, Yue Cao, Sasa Grbic, et al. Contrastive self-supervised learning from 100 million medical images with optional supervision. *Journal of Medical Imaging*, 9(6):064503–064503, 2022.

[7] Shamitha S Kotekani and V Ilango. Hemclust: An improved fraud detection model for health insurance using heterogeneous ensemble and k-prototype clustering. *International Journal of Advanced Computer Science and Applications*, 13(3), 2022.

[8] Fahad Rehman. Retail Transaction Dataset. https://www.kaggle.com/datasets/fahadrehman07/retail-transaction-dataset. Accessed: 2024-05-13.

[9] Yaohao Lian, Peng Wang, and Gaiyan Guo. Analysis model of efficiency and accuracy in big data based on clustering algorithm k-means. In *2023 International Conference on Internet of Things, Robotics and Distributed Computing (ICIRDC)*, pages 433–437, 2023.

[10] Yang Yang, Qi Chen, Tuo yi Huang, and Piyush Kumar Pareek. Application research of k-means algorithm based on big data background. In *2023 IEEE International Conference on Integrated Circuits and Communication Systems (ICICACS)*, pages 1–5, 2023.

[11] Liantian Li. Efficient distributed database clustering algorithm for big data processing. In *2021 6th International Conference on Smart Grid and Electrical Automation (ICSGEA)*, pages 495–498, 2021.

[12] V W Ajin and Lekshmy D Kumar. Big data and clustering algorithms. In *2016 International Conference on Research Advances in Integrated Navigation Systems (RAINS)*, pages 1–5, 2016.

[13] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96)*, pages 226–231. AAAI Press, 1996.

[14] Jon L Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[15] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned spatial indexes. *arXiv preprint arXiv:2008.10349*, 2020.

[16] Hamdi Tolga KAHRAMAN. Land Mines. UCI Machine Learning Repository, 2022. DOI: https://doi.org/10.24432/C54C8Z.

[17] Heider D. Wagner, Dennis and Georges Hattab. Secondary Mushroom. UCI Machine Learning Repository, 2023. DOI: https://doi.org/10.24432/C5FP5Q.

[18] Maggie Demkin Walter Reade. Categorical feature encoding challenge ii, 2019.

[19] OMER COLAKOGLU. 1M rows Turkish Market Sales Dataset, 2023.

[20] Yu Zheng, Xing Xie, and Wei-Ying Ma. Geolife gps trajectory dataset. Microsoft Research Asia, 2009. Available: https://www.microsoft.com/en-us/research/project/geolife-build-social-networks-using-human-location-histories/.