Evaluating runtime in Binary clustering of Single-cell RNA Sequencing data

Milan de Koning[1]

Supervisor(s): Marcel Reinders, Gerard Bouland

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Milan de Koning
Final project course: CSE3000 Research Project
Thesis committee: Marcel Reinders, Gerard Bouland, Bart Gerritsen

Abstract

As single-cell RNA sequencing techniques improve and more cells are measured in individual experiments, cell clustering procedures become increasingly more computationally intensive. This paper studies the runtime performance impact of a specialized clustering algorithm for data converted to a binary format, in order to reduce computational burden. We experimentally show that our specialized algorithm runs faster than the Seurat library on small datasets, and that with proper dimensionality reduction and approximation techniques, the algorithm could be more scalable than current methods. Optimizations for cluster quality and memory efficiency are not considered in this paper.

1 Introduction

Single-cell RNA sequencing (scRNAseq) techniques allow researchers to measure the gene expression levels of individual cells. Many data analysis methods for this scRNAseq data exist to improve biological insight [1]. As techniques for scRNAseq improve and datasets with over two million cells [2] are currently being generated, data analysis methods become increasingly more computationally intensive. A commonly used downstream analysis method for this data is cell clustering, where cells are grouped together based on gene signature similarity between cells. Since this approach requires that every cell is compared to every other cell, it can result in computationally intensive procedures. Currently, datasets can include more than 40.000 genes and over two million cells [2], leading to very high time and memory costs. Because of this, it becomes increasingly more important to find a resource- and time-efficient method to perform these clustering procedures.

To address the computational challenges posed by large-scale scRNAseq datasets, current algorithms (Seurat [3]) use a multistep approach involving pre-processing, dimensionality reduction, k-Nearest-Neighbour (k-NN) algorithms and community detection algorithms to generate clusters [4]. In the pre-processing steps, the data is normalized, variable features are selected, and the data is centered and scaled. Following this, dimensionality reduction is performed using approximate Principal Component Analysis (PCA) in order to make further computations feasible. After this step, an approximate k-Nearest-Neighbour algorithm is used to create a connectivity graph. Finally, a community detection algorithm (e.g., Leiden [5]) is applied to generate the final clusters from this graph. (Figure 1). However, in many cases, this process still requires a lot of time and memory.

Along with recent developments that allow for more cells to be measured in individual experiments, overall sparsity in datasets is increasing as well [6]. In this context, higher sparsity means that a higher proportion of the data values is zero. Bouland et al. [7] argue that even though zeros may be measured due to either technical or biological factors, they still provide relevant biological information. In the same paper, it is also shown that for sparser datasets, a binary data representation is as informative as count-based data representations. The binary representation only takes into account whether a gene was measured in a cell, as opposed to storing how frequently each gene was measured in each cell (Figure 2).

Moreover, Bouland et al. show that storing the scRNAseq data in a binarized form can achieve a 17-fold reduction in storage requirements and suggest that a specialized clustering algorithm for binary scRNAseq data could allow for significant improvements in terms of runtime and peak memory usage. For example, lower memory usage and use of binary logic operations instead of numerical operations could save a significant amount of time for clustering procedures.

The goal of this research is to create and evaluate a specialized clustering algorithm for binarized data. Specifically, we introduce two approaches: An exact binary algorithm where every cell is compared to every other cell, and an approximated binary algorithm where cells are only compared to cells that are likely to be similar (Section 4). We aim to find differences in runtime when using binary clustering algorithms compared to a traditional method. Furthermore, binary metrics are compared against each other in terms of runtime. However, we do not go into detail on either memory constraints or optimizing for biological accuracy. Furthermore, even though reading the data into memory is a step where runtime gains can be made, evaluation of this is out of scope for this study due to the number of different possible file formats and data representations. ScRNAseq data analysis tools can sometimes support importing data from up to 11 different file formats or pre-processing tools [8]. All algorithms, experiments, and raw results are available on `https://github.com/mdek2053/Binaryclustering`.

The runtimes of different approaches were evaluated experimentally, since theoretical runtime analysis can be very complex and usually does not accurately reflect runtimes in practice. The results show that binary algorithms run faster than Seurat when fewer features are used and that algorithms that approximate nearest neighbours run faster on datasets with more cells. We conclude that binary clustering has the potential to be faster and more scalable than traditional clustering methods, but that binary dimensionality reduction and approximation techniques are required for scalable binary algorithms.

2 Results

The runtimes of three approaches were compared: 1) Seurat, 2) an exact binary algorithm, and 3) an approximated binary algorithm. (See section 4). Raw results can be found in the online repository.

Binary clustering performs significantly faster than Seurat on smaller datasets

Both binary clustering algorithms take significantly less time than Seurat when applied to smaller datasets. On datasets of 1.000 cells, the mean runtime of Seurat is 1.2 seconds, compared to 0.02 s and 0.21 s for the exact and approximate binary approaches respectively (Figure 3a). This holds on datasets for 10.000 cells as well, where the means are 7.34

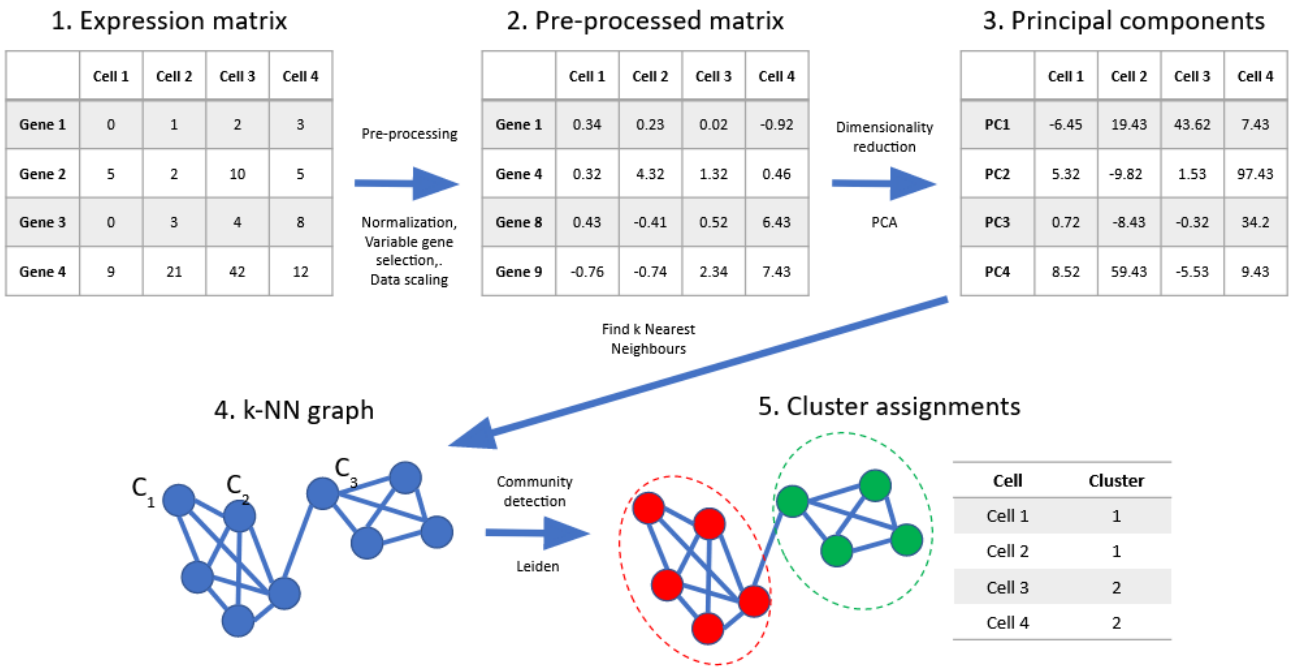


Figure 1: Conventional scRNAseq clustering workflow. The raw gene expression matrix is normalized, variable genes are selected, and the data is scaled and centered to produce a new matrix to which Principal Component Analysis (PCA) can be applied. After PCA is applied, every cell is expressed with a fixed number of principal components. Using these principal components, a k-Nearest-Neighbour (k-NN) graph is created. Finally, a community detection algorithm, such as Leiden, is applied to assign each cell to a cluster.

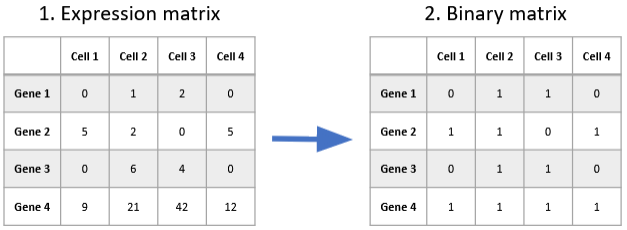


Figure 2: Raw scRNAseq data (left) converted to binary scRNAseq data (right). Every value of 0 remains a 0, but every value greater than 1 becomes a 1.

s for Seurat, 0.55 s for Binary exact, and 3.66 s for binary approximated (Figure 3b).

This difference occurs because binary algorithms can compare cells more efficiently, skip PCA, and have less overhead. PCA is the most time-consuming step in the Seurat workflow for smaller datasets, as it takes an average of 53% of the total runtime for datasets of 1.000 and 10.000 cells (Figures 5a and 5b).

Approximated methods scale better when cell counts increase

Both approximated methods, Seurat and Binary approximated, scale much better than Binary exact. When applying the algorithms to 10 times more cells, the runtimes of approximated approaches grow by average factors of 8.9 and 18.7 respectively. In contrast, the average factor by which the exact binary algorithm grows for a 10-fold increase in cells is 86.1 (Figure 4).

This can be attributed to a difference in runtime complexity. The exact algorithm compares all cells in $\mathcal{O}(n^2)$ time, because every cell is compared to every other cell. Conversely, the implementation of the approximated nearest neighbour library allows building the k-NN graph in $\mathcal{O}(n \cdot log(n))$ time. (Section 4) This means that especially for larger datasets, the exact method will scale significantly more.

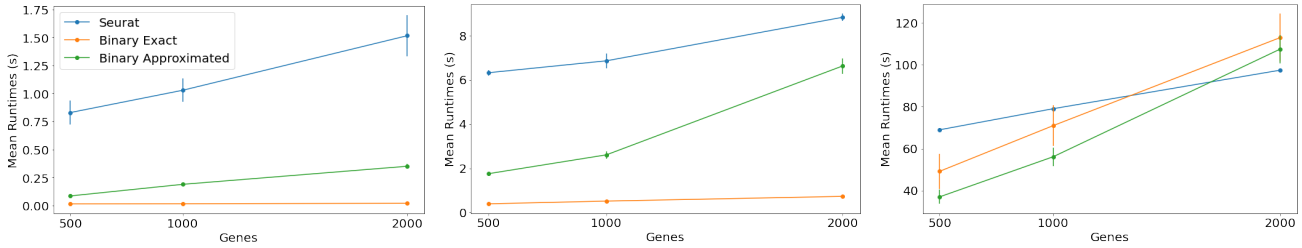
When gene counts rise, Seurat scales much less than binary methods on larger datasets

For datasets with 100.000 cells, doubling the number of genes increases the runtime of Seurat by an average factor of 1.2. In comparison; Binary exact scales with an average factor of 1.5 and Binary approximated with an average factor of 1.7 for doubling the number of genes (Figure 3c).

Further investigation into the runtime of different stages of the Seurat workflow shows that the k-NN graph does not scale when the number of genes increases, because it uses a fixed number of dimensions. Moreover, Seurat uses an approximated version of PCA as well, which scales with only an average factor of 1.5 when doubling the number of genes (Figure 5).

Creating a k-NN graph using binary data is orders of magnitude faster than using continuous data

Even though the full clustering procedure is not always faster, k-NN graph creation with binary data is still much faster than with continuous data, since the binary approach creates a k-

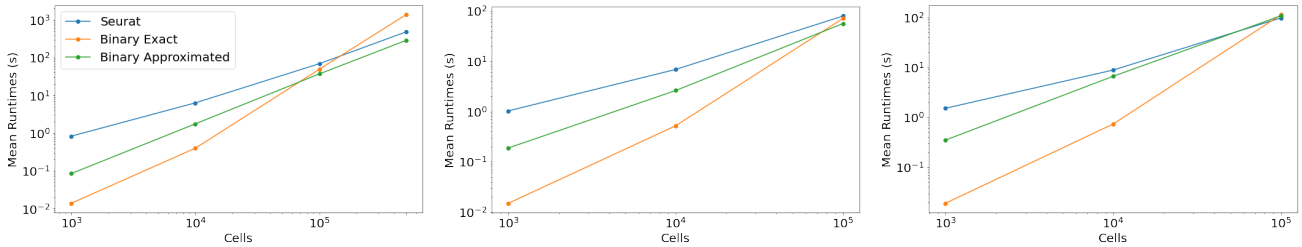


(a) Mean runtimes for datasets with 1.000 cells.

(b) Mean runtimes for datasets with 10.000 cells.

(c) Mean runtimes for datasets with 100.000 cells.

Figure 3: Mean runtimes compared across three clustering algorithms, shown for different cell counts. Variances are shown for every point but are too small to be visible for some. (a) shows the values for 1.000, (b) for 10.000, and (c) for 100.000 cells.

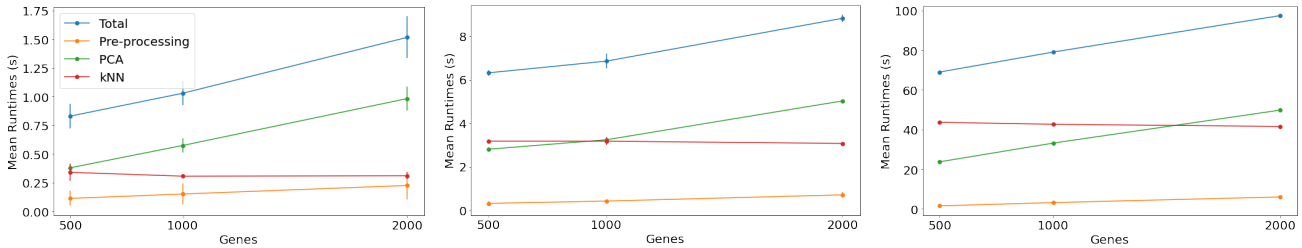


(a) Mean runtimes for datasets with 500 genes.

(b) Mean runtimes for datasets with 1000 genes.

(c) Mean runtimes for datasets with 2000 genes.

Figure 4: Mean runtimes across different clustering algorithms, shown for different gene counts. Variances are too small to be visible on any plot due to the logarithmic scale. (a) shows the values for 500, (b) for 1.000, and (c) for 2.000 genes.

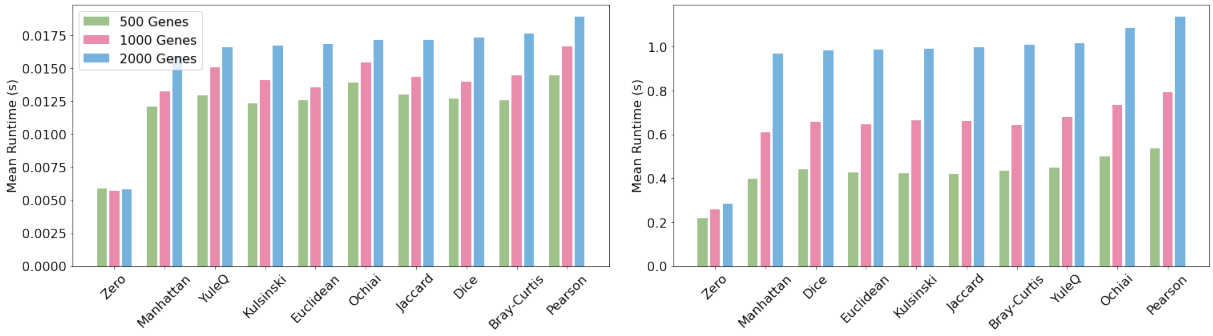


(a) Mean runtimes for different stages of the Seurat workflow on datasets with 1000 cells.

(b) Mean runtimes for different stages of the Seurat workflow on datasets with 10000 cells.

(c) Mean runtimes for different stages of the Seurat workflow on datasets with 100000 cells.

Figure 5: Mean runtimes compared among three clustering algorithms, shown for different cell counts. Variances are shown for every point but are too small to be visible for some. (a) shows the values for 1.000, (b) for 10.000, and (c) for 100.000 cells.



(a) Mean runtimes for different metrics on datasets with 1.000 cells

(b) Mean runtimes for different metrics on datasets with 10.000 cells

Figure 6: Mean runtime for different similarity metrics. (a) shows the runtimes for datasets with 1.000 cells, and (b) shows them for 10.000 cells. The "Zero" metric, which always returns zero, was also included.

NN graph using 500, 1.000 or 2.000 dimensions, while Seurat only uses 10 continuous dimensions after applying PCA. When normalizing the runtimes of the k-NN graph creation stages in Seurat and Binary approximated for the number of dimensions, the Binary approximated approach is around 118 times faster than Seurat. This is a valid comparison since both algorithms use the same k-NN algorithm, with the only difference being comparison logic. For both binary methods, creating the k-NN graph accounts for over 99% of the total runtime.

Manhattan is fastest and Pearson is slowest

When comparing different binary metrics using the exact binary algorithm, Manhattan distance results in the lowest runtimes, with means of 0.14 and 0.66 for datasets with 1.000 and 10.000 cells, while Pearson similarity results in the highest runtimes, with means of 0.17 s and 0.82 s respectively. (Figure 6). Overall, the trend is that metrics with more calculations take more time, and metrics that use square roots tend to take more time, since this is the most time-consuming mathematical operation used in any of the metrics. However, the mean runtime for every metric is over 80% of the mean runtime for the Pearson similarity.

3 Discussion

As suggested by Bouland et al. in [7], binarizing data and applying a specialized algorithm can, in some cases, significantly reduce the runtime compared to existing methods. However, for larger datasets, the binary algorithms do not perform as well as existing algorithms, even though the aim was to create a more scalable algorithm. This can be attributed to the fact that even though we showed that k-NN graph creation on binary data is approximately 118 times faster than creating it using continuous data, applying PCA allows the k-NN graph to be computed using a fixed, small number of dimensions. PCA is so effective because it can convert sparse input data into very information-dense continuous output data. In contrast, retaining this level of information with binary output is not feasible unless more dimensions are used. Moreover, the fact that PCA runtimes do not scale proportionally to an increase in genes and k-NN graph creation does not scale at all, allows Seurat to perform better on datasets with more genes. Finally, it was found that applying different distance metrics does not result in runtime gains or losses greater than 20%.

Improving the scalability of the binary clustering algorithm

The results emphasize that dimensionality reduction is essential for a binary clustering algorithm to be competitive in terms of runtime. A binary clustering algorithm could be competitive with existing methods in terms of runtime if a binary dimensionality reduction method that satisfies the following constraints can be realized:

1. The binary dimensionality reduction can retain sufficient information in less than approximately 118 times the amount of required principal components.

2. The binary dimensionality reduction is not slower than current dimensionality reduction methods for numerical data.

Pratap et al. [9] show that binary dimensionality reduction can be performed efficiently for sparse datasets while maintaining estimates for several distance measure and outputting data in a binary format. If such an algorithm satisfies the aforementioned constraints in practice, it must be applied in order to achieve scalability in future iterations.

Furthermore, the results show that approximated methods are necessary in order to achieve scalable algorithms. The only binary distance metric that the `Annoy` library supports is the Hamming distance. This could either be extended or a specialized approximate nearest neighbour library for binary data could be developed.

Finally, binary clustering could be significantly faster if an efficient popcount operation for longer bit vectors is used. Mula et al. [10] show that significant gains can be made compared to the built-in popcount operations for longer bit vectors.

Future research

The results still leave questions that should be addressed in future research. Bouland et al. [7] show that binary data can be stored much more efficiently. This could result in large runtime gains when loading the data compared to other formats. Moreover, the choice of k value could have a large impact on runtime as well. The several possible improvements mentioned in the previous paragraph should be tested in practice against the benchmarks set in this paper. Finally, using more powerful hardware, larger datasets should be evaluated too.

Apart from runtime evaluations, it is imperative to consider cluster quality as well. The choice of k, similarity metric, and community detection algorithm could all have an impact on biological accuracy. In addition, the impact of several approximation and dimensionality reduction methods on cluster quality and accuracy should be researched as well.

Finally, in case it is not feasible to make the entire pipeline up to creating the k-NN graph work entirely on binary data in a time-efficient manner, a specialized version of PCA that converts sparse binary input to dense continuous output could be used. This can still yield significant improvements in run time while decreasing storage and memory requirements, as described in [7]. A method to compute principal components on sparse binary datasets is proposed by Lee et al. in [11].

Responsible research

For the sake of transparency and reproducibility, all relevant information is available in either this paper or in the public repository. This paper should give a broad overview of the experiment and research, while all the algorithms, scripts for experiments, and results are available online. Furthermore, it should be mentioned that some pilot experiments were run beforehand. The results of these experiments were discarded since they were conducted on either older versions of the algorithm or with incorrect configurations. Finally, the main experiments are reproducible using the code from the repository and the hardware and software configurations mentioned.

Conclusion

Binary clustering shows promise because, with some additions and modifications, it could be more scalable than existing clustering algorithms. Proper binary dimensionality reduction and approximation techniques are essential in achieving this.

4 Methods

Metric selection

The metrics for evaluation were selected based on recent research into scRNAseq clustering. Metrics were taken from [12] and [13]. For most of the metrics, Choi et al. [14], define binary versions for most of the selected metrics. Metrics that were impossible to convert to a binary version are left out. Furthermore, some metrics are equivalent to each other when used on binary data, so duplicates were also removed. This results in the following list of metrics:

1. Bray-Curtis distance [14]
2. Ochiai distance [1] [14]
3. Dice's index [14]
4. Euclidean distance [14]
5. Manhattan distance [14]
6. Jaccard similarity [14]
7. Kulsinski's index [2] [17]
8. Pearson correlation [3] [14]
9. Yule's Q [14]

4.1 Binary algorithm implementations

The binary clustering algorithm was implemented in C++[4], using the `igraph` [18], `Rcpp` [19], and `Annoy`[5] libraries. The `igraph` library was used for Leiden and Louvain clustering, `Rcpp` was used for integration with `R` [20] and `Annoy` was used for efficiently approximating nearest neighbours. The source code is available in the online repository.

Workflow

The algorithm follows a simplified version of the traditional workflow of clustering algorithms. The data is loaded into memory, pre-processed, connected to form a k-NN graph, and clustered (Figure 7). The k-NN graph can be either generated exactly or approximated. Every step can be triggered individually from `R`, although only the final step returns the results.

Data representation

The scRNAseq data is loaded as a `csv` file and stored in a binary format. Each cell is stored as a vector of unsigned longs, with each gene represented as one bit. This allows 64 features to be stored in one unsigned long. During the preprocessing step, the popcount (amount of 1 values) for each cell is computed and stored along with the raw binary data. Following this, either an exact nearest neighbour graph can be, or the nearest neighbours can be approximated using the `Annoy` library. Finally, the clusters are generated by using the previously generated k-NN graph as input for the `igraph` library and applying a clustering algorithm.

Approximate k-NN graph generation

The approximation techniques are implemented using the `Annoy` library. The approximation consists of two stages: index tree creation and nearest neighbour retrieval [21]. In the first stage, trees are built by recursively dividing the space into subspaces of approximately equal size. In the experiments for this paper, 50 trees were built, which is the Seurat default. The second stage iterates over all cells and queries all trees for every cell. This way, cells are only compared to other

[1] [14] also defines a binary cosine similarity, but this instance uses a squared denominator, unfaithful to the non-binary formula as found in other work, such as [15]. [16] mentions that "For binary vectors, the cosine measure is also called the Ochiai coefficient".

[2] [14] defines two versions of this metric, but neither is the one used in [12].

[3] [14] defines both the Pearson coefficient ρ as well as multiple metrics using this coefficient. We use the coefficient ρ as the metric.

[4] https://cplusplus.com/

[5] https://github.com/spotify/annoy

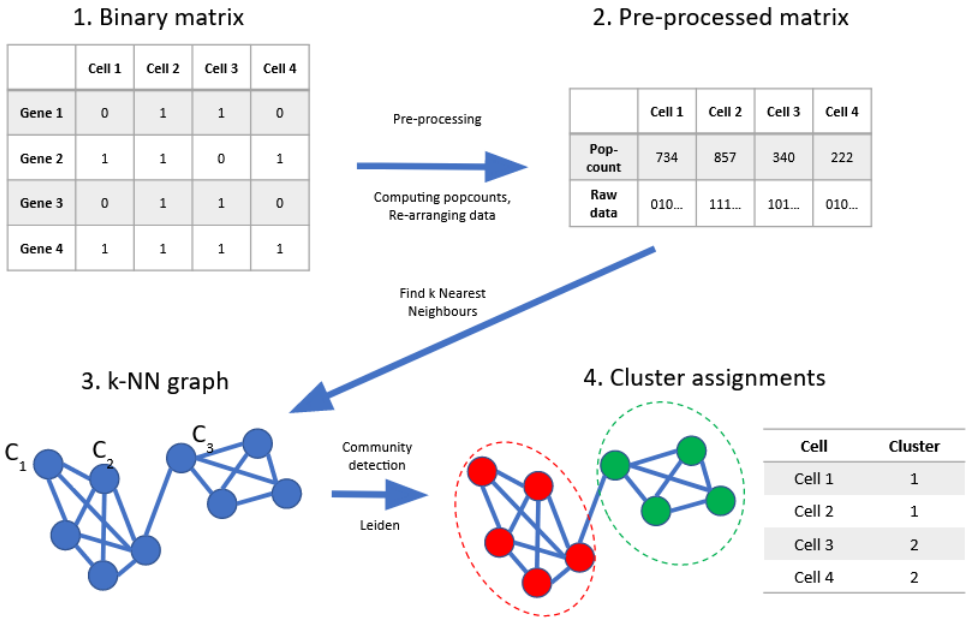


Figure 7: Binary clustering workflow. The binary data is pre-processed by computing the popcount (amount of 1 values) and storing the raw data alongside it. Using this pre-processed matrix, a k-Nearest-Neighbour (k-NN) graph is created. Finally, a community detection algorithm is applied to assign each cell to a cluster.

cells that are in the same subspace in any of the trees, instead of being compared to all other cells. The runtime complexity of the entire process is $\mathcal{O}(n \cdot log(n))$.

This process is divided over the maximum number of hardware-supported threads. As of now, Hamming distance (equivalent to binary Manhattan distance) is the only supported metric for this library. This metric is implemented by summing the popcounts of the XOR of each unsigned long in the cell vector. The XOR of two bits gives a 0 if they are equal and a 1 if they are not equal. Counting the 1 values in the result of the XOR gives exactly the amount of dimensions in which the values differ. This value is, by definition, equal to the Hamming distance. `__builtin_popcount` is used to implement the popcount.

Exact k-NN graph creation

The exact technique is implemented in a more traditional way by simply comparing each cell to every other cell. The cells are divided into chunks of equal size and divided over the maximum numbert of hardware-supported threads. Each thread compares all cells in its respective chunk to every other cell in the entire dataset. Although the creation of a similarity matrix is supported, all experiments used the direct creation of a k-NN graph since this is more time-efficient.

Metric implementation

For the exact approach, more binary metrics are supported. Here, the main principles applicable to all metrics are illustrated, while the implementations can be found in either Appendix A or in the public repository.

We defined metrics the same way as in [14]. In short, when comparing two binary vectors v and u, each feature can have either a (1, 1) pair (features are present in both v and u), a (0, 1) pair (feature is only present in u), (1, 0) pair (feature is only present in v) or a (0, 0) pair (feature is present in neither v nor u). We define a, b, c and d to be the number of (1, 1) pairs, (0, 1) pairs, (1, 0) pairs, and (0, 0) pairs respectively (Table 1). The number of dimensions in the vector should always equal $a + b + c + d$.

	1 in u	0 in u
1 in v	a = count(1, 1)	c = count(1, 0)
0 in v	b = count(0, 1)	d = count(0, 0)

Table 1: Definitions of a, b, c and d. a is equal to the number of pairs where both values are 1, b to the number of (0, 1) pairs, c to the number of (1, 0) pairs, and d to the number of (0, 0) pairs.

The value of a is computed in a similar way to the `Annoy` library: By summing the popcounts of the AND of each of the unsigned longs in the cell vector. Performing an AND operation only results in a 1 if the initial features were (1, 1). Therefore, counting the 1 values in the resulting vector yields the amount of (1, 1) pairs in the two initial vectors, which is equal to a by definition. Popcounts are also computed using the `__builtin_popcount` function.

After the value of a is computed, b, c and d can be computed (if required) in constant time. This is because the popcount for each cell was computed beforehand. The popcounts of the two cells represent the values $(a + b)$ and $(a + c)$, so the values of b and c can be found by subtracting a from the popcounts. After this, d can be found by subtracting a, b and c from the length of the vector. Using these four values, every metric can be computed.

4.2 Experiment

Datasets

Since the most relevant property of datasets for the experiments was size, 10 random samples of different sizes were taken from the "Human Lung Cell Atlas" [2] dataset, available at `https://beta.fastgenomics.org/p/hlca`. The sparsity of datasets does not impact the performance of the binary algorithms because an equal amount of computations is performed for any proportion of ones and zeros. The random samples have 1.000, 10., and 100.000 cells, and 500, 1.000, and 2.000 genes respectively, resulting in nine datasets. The gene counts were chosen to simulate realistic sizes after variable gene selection [22]. The cell counts were chosen to represent somewhat realistic dataset sizes. Furthermore, one dataset with 500.000 cells and 500 genes was used to evaluate behaviour on very large datasets. Due to hardware constraints, no larger datasets could be evaluated.

Hardware & Compilation

All experiments were conducted on a machine with 16 GB of memory, an AMD Ryzen 5 CPU with 8 threads and a frequency of 2.1 GHz, running 64-bit Ubuntu 20.04.6. The binary algorithm was compiled using the `g++` version 9.4.0 compiler and the `-Ofast` and `-march=native` flags. For the required libraries, include and link flags were added to the compile command as well.

Setup

The experiments were conducted using three separate `R` scripts. The first script measures the runtime of Seurat, the second one measures the runtime of both versions of the binary algorithms, and the third one measures the runtime of different binary metrics. All experiments use a value of 30 for k, as this is the default value for Seurat. All scripts are available in the online repository.

To evaluate the runtime of Seurat, the data was loaded into memory, normalized, scaled, PCA was applied, and the k-NN graph was constructed using the first 10 principal components. Contrary to the standard workflow as described in [4], variable feature selection was skipped because the gene counts in the dataset are already representative of gene counts after variable feature detection [22]. Moreover, no steps after k-NN graph creation were evaluated, as binarized data has no effect on runtime afterwards. Start times, intermediate times, and end times were recorded using the `Sys.time()` method. The PCA and k-NN creation steps were measured individually, while normalizing and scaling the data is grouped under Pre-processing. Reading data is not included in any time measurement, as this heavily depends on the data format and is out of scope for this research. For each dataset, 10 runtimes were recorded, except for the 500.000 $\times$ 500 dataset, for which one runtime was recorded due to time and hardware constraints.

The experiments for the binary algorithms were conducted similarly. For this, the data was only read into memory, preprocessed, and converted into a k-NN graph. Measuring the runtimes was also done similarly to the Seurat experiments, by starting measurements only after reading data and stopping after the k-NN graph was generated. The same sample sizes that the Seurat experiment uses were used in evaluation of both the exact and approximate binary algorithms.

Finally, the runtime of different binary metrics was evaluated by randomizing the order of the list of metrics and evaluating them consecutively. For this experiment, only the runtime of the k-NN graph creation was measured, since that is the only stage that is affected by the different metrics in terms of runtime. This experiment was conducted using the six smallest datasets, ranging from 1.000 $\times$ 500 to 10.000 $\times$ 2.000. For each of these datasets, 10 runtimes were recorded for each metric.

References

[1] G. Chen, B. Ning, and T. Shi, "Single-cell rna-seq technologies and related computational data analysis," *Frontiers in Genetics*, vol. 10, 2019. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fgene.2019.00317

[2] L. Sikkema, D. C. Strobl, L. Zappia, E. Madissoon, N. S. Markov, L.-E. Zaragosi, M. Ansari, M.-J. Arguel, L. Apperloo, C. Becavin *et al.*, "An integrated cell atlas of the human lung in health and disease," *bioRxiv*, pp. 2022–03, 2022.

[3] Y. Hao, S. Hao, E. Andersen-Nissen, W. M. M. III, S. Zheng, A. Butler, M. J. Lee, A. J. Wilk, C. Darby, M. Zagar, P. Hoffman, M. Stoeckius, E. Papalexi, E. P. Mimitou, J. Jain, A. Srivastava, T. Stuart, L. B. Fleming, B. Yeung, A. J. Rogers, J. M. McElrath, C. A. Blish, R. Gottardo, P. Smibert, and R. Satija, "Integrated analysis of multimodal single-cell data," *Cell*, 2021. [Online]. Available: https://doi.org/10.1016/j.cell.2021.04.048

[4] P. Hoffman and S. Lab, "Seurat command list," Mar 2023. [Online]. Available: https://satijalab.org/seurat/articles/essential_commands.html

[5] V. A. Traag, L. Waltman, and N. J. Van Eck, "From louvain to leiden: guaranteeing well-connected communities," *Scientific reports*, vol. 9, no. 1, p. 5233, 2019.

[6] S. C. Hicks, F. W. Townes, M. Teng, and R. A. Irizarry, "Missing data and technical variability in single-cell rna-sequencing experiments," *Biostatistics*, vol. 19, no. 4, pp. 562–578, 2018.

[7] G. A. Bouland, A. Mahfouz, and M. J. Reinders, "The rise of sparser single-cell rnaseq datasets; consequences and opportunities," *bioRxiv*, pp. 2022–05, 2022.

[8] R. Hong, Y. Koga, S. Bandyadka, A. Leshchyk, Y. Wang, V. Akavoor, X. Cao, I. Sarfraz, Z. Wang, S. Alabdullatif *et al.*, "Comprehensive generation, visualization, and reporting of quality control metrics for single-cell rna sequencing data," *Nature Communications*, vol. 13, no. 1, p. 1688, 2022.

[9] R. Pratap, D. Bera, and K. Revanuru, "Efficient sketching algorithm for sparse binary data," in *2019 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2019, pp. 508–517.

[10] W. Muła, N. Kurz, and D. Lemire, "Faster population counts using avx2 instructions," *The Computer Journal*, vol. 61, no. 1, pp. 111–120, 2018.

[11] S. Lee, J. Z. Huang, and J. Hu, "Sparse logistic principal components analysis for binary data," 2010.

[12] E. R. Watson, A. Mora, A. Taherian Fard, and J. C. Mar, "How does the structure of data impact cell–cell similarity? Evaluating how structural properties influence the performance of proximity metrics in single cell RNA-seq data," *Briefings in Bioinformatics*, vol. 23, no. 6, 09 2022, bbac387. [Online]. Available: https://doi.org/10.1093/bib/bbac387

[13] M. A. Skinnider, J. W. Squair, and L. J. Foster, "Evaluating measures of association for single-cell transcriptomics," *Nature methods*, vol. 16, no. 5, pp. 381–386, 2019.

[14] S. Choi, S.-H. Cha, and C. Tappert, "A survey of binary similarity and distance measures," *J. Syst. Cybern. Inf.*, vol. 8, 11 2009.

[15] F. Rahutomo, T. Kitasuka, and M. Aritsugi, "Semantic cosine similarity," in *The 7th international student conference on advanced science and technology ICAST*, vol. 4, no. 1, 2012, p. 1.

[16] P. Kalgotra, R. Sharda, and A. Luse, "Which similarity measure to use in network analysis: Impact of sample size on phi correlation coefficient and ochiai index," *International Journal of Information Management*, vol. 55, p. 102229, 2020, impact of COVID-19 Pandemic on Information Management Research and Practice: Editorial Perspectives. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0268401220314286

[17] T. S. community, "Scipy.spatial.distance.kulsinski." [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.kulsinski.html

[18] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal*, vol. Complex Systems, p. 1695, 2006. [Online]. Available: https://igraph.org

[19] D. Eddelbuettel and R. François, "Rcpp: Seamless r and c++ integration," *Journal of statistical software*, vol. 40, pp. 1–18, 2011.

[20] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2023. [Online]. Available: https://www.R-project.org/

[21] E. Bernhardsson, "Spotify/annoy: Approximate nearest neighbors in c++/python optimized for memory usage and loading/saving to disk," Apr 2023. [Online]. Available: https://github.com/spotify/annoy/blob/main/README.rst

[22] K. Su, T. Yu, and H. Wu, "Accurate feature selection improves single-cell rna-seq cell clustering," *Briefings in Bioinformatics*, vol. 22, no. 5, p. bbab034, 2021.

A Metric implementations

```
auto jaccardDistance = [](const CellData& left, const CellData& right) {
    int diff = ANDCount(left.chunks, right.chunks);

    return 1.f - (diff / (float)(left.count + right.count - diff));
};

auto euclideanDistance = [](const CellData& left, const CellData& right) {
    int diff = XORCount(left.chunks, right.chunks);
    return std::sqrt((float)diff);
};

auto ochiaiDistance = [](const CellData& left, const CellData& right) {
    int diff = ANDCount(left.chunks, right.chunks);
    return 1.f - (diff / std::sqrt((float)left.count * (float)right.count));
};

auto manhattanDistance = [](const CellData& a, const CellData& b) {
    return (float)XORCount(a.chunks, b.chunks);
};

auto kulsinskiDistance = [](const CellData& left, const CellData& right) {
    int a = ANDCount(left.chunks, right.chunks);

    int lcnt = left.count;
    int rcnt = right.count;
    int N = left.size;

    int numerator = lcnt + rcnt - 3 * a - N;
    int denominator = lcnt + rcnt - 2 * a - N;

    return numerator / (float)denominator;
};

auto pearsonSimilarity = [](const CellData& left, const CellData& right) {
    float a = ANDCount(left.chunks, right.chunks);
    float b = left.count - a;
    float c = right.count - a;
    float d = left.size - a - b - c;
    return (a * d - b * c) / std::sqrt(left.count * right.count * (b + d) * (c + d));
};

auto yuleQSimilarity = [](const CellData& left, const CellData& right) {
    float a = ANDCount(left.chunks, right.chunks);
    float b = left.count - a;
    float c = right.count - a;
    float d = left.size - a - b - c;
    return (a * d - b * c) / (a * d + b * c);
};

// Bray-Curtis distance is 1 - sorensen similarity
auto sorensenSimilarity = [](const CellData& a, const CellData& b) {
    int symdif = XORCount(a.chunks, b.chunks);
    return 1.0f - (symdif / (float)(a.count + b.count));
};
```

```
auto diceSimilarity = [](const CellData& left, const CellData& right) {
    int a = ANDCount(left.chunks, right.chunks);
    return 2 * a / (float)(left.count + right.count);
};
```