ESE589 Course Project I

Star-Cubing Algorithm October 24, 2019  
Professor Alex Doboli  
Kenneth Ng (110211727)  
Kurt Butler (109587387)

# Introduction

In this project, we implement the Star-Cubing algorithm as an approach to data cube computation. We consider a Python-based version of the algorithm. We evaluate its performance using small data sets to demonstrate functionality and large data benchmarks from the UCI Machine Learning repository [1] to test computational efficiency. Finally, we consider possible changes that can be made to Star-Cubing to apply it in other contexts.

Contents

[Introduction 1](#_Toc22936140)

[Algorithm Implementation 2](#_Toc22936141)

[Data Preprocessing 2](#_Toc22936142)

[Star Table 2](#_Toc22936143)

[Star Tree 3](#_Toc22936144)

[Textbook Example 4](#_Toc22936145)

[Table 1: Textbook Example⸸ 4](#_Toc22936146)

[Table 2: Textbook Example Star Table 4](#_Toc22936147)

[Sample Output File 5](#_Toc22936148)

[Varied Threshold 5](#_Toc22936149)

[Table 3: Varying MinSup for Soybean(Large) Data Set 5](#_Toc22936150)

[Plot 1: Execution Time and Memory Use vs MinSup 5](#_Toc22936151)

[Benchmark Performance 6](#_Toc22936152)

[Table 4: UCI Machine Learning Repository Benchmarks 6](#_Toc22936153)

[Memory Usage 7](#_Toc22936154)

[Plot 2: Benchmark Memory Usage 7](#_Toc22936155)

[Execution Time 7](#_Toc22936156)

[Plot 3: Benchmark Execution Times 7](#_Toc22936157)

[Plot 4: Normalized Execution Time vs Normalized Min Support 8](#_Toc22936158)

[Star-Cubing on Cyber-Physical Systems 8](#_Toc22936159)

[Conclusion 9](#_Toc22936160)

[References 9](#_Toc22936161)

# Algorithm Implementation

The star cubing algorithm in python requires four different components: data preprocessing, star table generation, star tree generation, and star cubing. In data preprocessing, our algorithm must handle many different types of datasets. The different datasets in the UCI machine learning database have categorical, binary, and numerical categories [1].

## Data Preprocessing

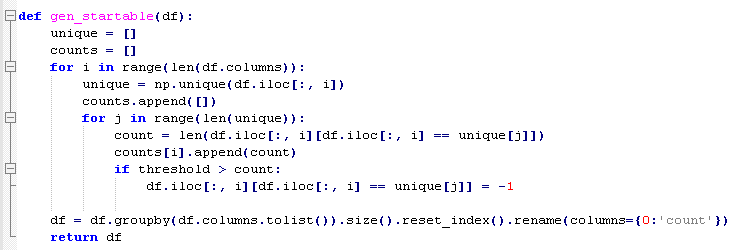
For Categorical and Boolean attributes, there is no *a priori* transformation that can be done to them, aside from enumerating them for use in a matrix. For Real and Integer Numerical attributes, we consider equilength binning to discrete the data. The number of partitions used to discretize the attributes of a data set are determined by inspection on a small subset of the data set.

When the program starts, the entire data set is scanned twice. On the first scan, the maximum and minimum for each attribute is measured, denoted and respectively. The bin length for each attribute, is determined by

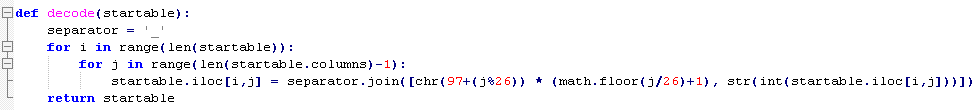
On the second scan, for each sample (indexed by ) and each numerical attribute (index by ), we map the values to an integer from 0 to representing their bin:.

## Star Table

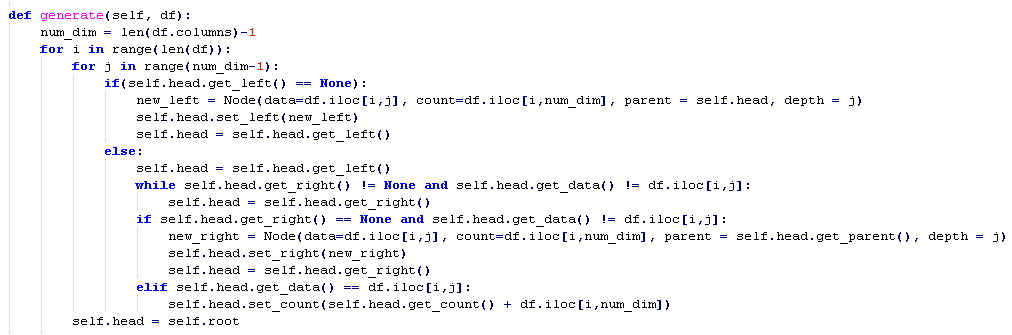
Once each category has been determined, the star table is generated. The unique counts for each category or histogram bin are determined. Any category or bin that is less than the threshold is reassigned to –1, our star case. This reassignment is guaranteed because the category and bins are labelled from 1 to N and each binary identification is 0 or 1. Therefore, the cuboid is never negative. The cuboids are then grouped according to each unique row (or sample) and the count is taken and appended to the end of the dataset.



With the star table, to make the columns more distinctive, each category is assigned a string to uniquely identify the cuboid. The convention is to label the first 26 columns a-z, aa-zz the following column, etc.



After the star table, the star tree is generated. Each tree has a root, head, and neighbor. Each node within the tree has a link “left” to the child node, “right” to sibling nodes, and “parent” to the parent node. As the tree is filled out, the dataset is read in row by row and traverses through the tree, inserting the node if the child node does not exist including the siblings, aggregating the count if the child node is the same as the one being inserted.



## Star Tree

The final task is generating the star tree. In contrast to the pseudo code given in the textbook, the recursive function does not leave the base tree for the child tree generated. Instead, the depth of the tree is monitored in each node and when the base tree makes a right reference to the sibling, the head of each neighboring tree is reset to match the base tree. During this process, the child node reference for the child of cnode is deleted. This implementation works because instead of using a recursive function to call the neighboring tree, a while loop is used while referencing the neighboring trees. Both implementations should fundamentally produce the same result; however, recursive calls typically take more time and while loops take more memory due to the extra references for identifying the location of the current node. Further investigations may analyze the differences between the two implementations. The following pseudo code is our program’s implementation:

Procedure starcubing(T, cnode) #cnode is the current node of the tree

For each non null neighbor, insert cnode to the neighboring tree if the current depth of the head of the neighboring tree is greater than cnode, traverse the head of the node upwards until the depths of both nodes are equivalent. Determine if the nodes are equal. If they are equivalent, aggregate. If not, insert as the child node.

If cnode.count >= threshold

If cnode != root

Print path to file

If cnode not a leaf

Cc = new copy of cnode

Tc = new tree

Append Tc to tree T’s neighbors

If cnode is not a leaf

Starcubing(T, cnode.child)

If Cc exists

Remove reference to Tc from Tc’s parent tree

Remove reference to cnode.child from cnode

If cnode sibling exists

Starcubing(T, cnode.sibling)

# Textbook Example

In *Data Mining* by Han et.al. [3], they consider the following toy data set (originally sourced from [2]) to illustrate the operation of the algorithm. We reuse this data set as a first example. Again, we consider the aggregate function SUM, which iceberg condition given by .

### Table 1: Textbook Example⸸

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | Count |
| 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 4 | 3 | 1 | 1 |
| 1 | 2 | 2 | 2 | 1 | 1 |
| 2 | 3 | 3 | 4 | 1 | 1 |
| 2 | 4 | 3 | 4 | 1 | 1 |

⸸Retrieved from page 206 of [3].

When the script is run, the program begins by building the star-table. In software, we represent the \* value by =-1, and all other attribute values are enumerated. The third entry of the intial data set is represented, but never processed, because it is not aggregated into any entry of the star table that satisfies the iceberg criterion.

### Table 2: Textbook Example Star Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | Count |
| 1 | \* | \* | \* | 1 | 1 |
| 1 | 1 | \* | \* | 1 | 2 |
| 2 | \* | 3 | 4 | 1 | 2 |

In Python, the Star Table is generated and saved explicitly. In the output.txt file, exploration of the Star Tree is illustrated. Below, we show the output generated by the textbook example. Based on the Star Table, the program descends through the star tree until it reaches a leaf (). At this point, the program scales back up the tree, computing cuboids in each subtree with each step, until it reaches a node with a linked sibling or root. If it encounters a node with sibling, it moves to the sibling instead of ascending and continues the procedure. The resulting star tree is identical to the one computed in [2].

### Sample Output File

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | root,a\_1  root,a\_1,b\_-1  root,a\_1,b\_-1,c\_-1  root,a\_1,b\_-1,c\_-1,d\_-1  root,a\_2  root,a\_2,b\_-1  root,a\_2,b\_-1,c\_3  root,a\_2,b\_-1,c\_3,d\_4 |

# Varied Threshold

If we consider one data set, in this case Soybean-Large from [1] (No. 10 in our listing), we can ask what the effect is of varying the minimum support threshold. As the threshold increase, the algorithm will prune more values and spend less time populating the tree, shown in Plot 1. In the same plot, we observe no correlation between memory usage and threshold.

### Table 3: Varying MinSup for Soybean(Large) Data Set

|  |  |  |
| --- | --- | --- |
| Threshold Value | Execution Time (sec) | Memory Usage (kB) |
| 0 | 9.83 | 137,003 |
| 10 | 7.68 | 139,604 |
| 50 | 7.57 | 140,632 |
| 70 | 7.19 | 137,232 |
| 100 | 4.47 | 139,330 |
| 150 | 1.71 | 141,279 |
| 200 | 0.91 | 138,174 |

### Plot 1: Execution Time and Memory Use vs MinSup

# Benchmark Performance

We consider the efficiency of our implementation of Star-Cubing using the benchmark data sets available from [1]. Because each data set is attributed to different contributors, we will refer to each data set using its name within the UCI database. For each benchmark, we consider the computer memory usage and execution time of the program operating on each data set.

### Table 4: UCI Machine Learning Repository Benchmarks

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No. | Data Set Name | Attribute Types | Execution Time (sec) | Memory Usage (kB) | No. Samples | No. Attributes |
| 1 | Lymphography | Categorical | 2.532 | 111,251 | 148 | 18 |
| 2 | Cloud⸸ | Real | 25.937 | 125,194 | 1024 | 10 |
| 3 | Cloud⸸ | Real | 33.151 | 125,129 | 1024 | 10 |
| 4 | SPECT Heart | Categorical | 1.543 | 116,437 | 267 | 22 |
| 5 | SPECTF Heart | Categorical | 10.400 | 122,024 | 267 | 44 |
| 6 | Digital Colposcopy (Green) | Real | 29.467 | 129,278 | 287 | 69 |
| 7 | Digital Colposcopy (Hinselmann) | Real | 30.642 | 125,579 | 287 | 69 |
| 8 | Digital Colposcopy (Schiller) | Real | 28.052 | 125,870 | 287 | 69 |
| 9 | Soybean (Small) | Categorical | 1.809 | 125,526 | 47 | 35 |
| 10 | Soybean (Large) | Categorical | 10.917 | 125,444 | 307 | 35 |
| 11 | Travel Reviews | Real | 11.430 | 119,591 | 981 | 25 |
| 12 | Website Phising | Integer | 6.442 | 125,477 | 1353 | 10 |
| 13 | Wine | Integer, Real | 53.657 | 109,830 | 178 | 13 |
| 14 | Japanese Vowels | Real | 6.371 | 129,020 | 4274 | 12 |
| 15 | Seismic | Real | 23.034 | 115,401 | 2584 | 19 |

⸸The Cloud data set includes two separate data sets concatenated. For test no. 2, we used data set 1 (DB1) and for test no. 3 we used data set 2 (DB2).

## Memory Usage

We notice that across data sets, the memory usage of the algorithm is generally consistent, except for the Adult data set which was considerably larger than the other benchmarks used. When the file size is considered, most data sets did not exceed 500kB, and so when compared to Plot 2, such fluctuations due to file size have little effect.

### Plot 2: Benchmark Memory Usage

## Execution Time

In Plot 3, we consider the execution time of each benchmark example, which took less than a minute, apart from the Adult data set which required a very long time to process.

### Plot 3: Benchmark Execution Times

If we consider many combinations of thresholds, and benchmarks, we can understand the trend in performance. Since each benchmark varies in size (No. samples or No. attributes), we define *norm time* by . We can also consider a *normalized threshold*, which is defined to be . When plotted together, we observe a general trend of inverse correlation, meaning that execution time is higher when the threshold accounts for a lesser percentage of data, inducing less pruning. For some data sets, such as Wine, we noticed that the dependence on Norm Threshold was not as apparent. We attribute this to the fact that the overall file size of Wine is small (12kB), and so the execution time, measured to be about 2 seconds, depends less on data scaling and more on the boilerplate code. Thus, selection of threshold is important for determine how the execution time scales with data set size.

### Plot 4: Normalized Execution Time vs Normalized Min Support

# Star-Cubing on Cyber-Physical Systems

In [4], the authors consider a distributed computing network of embedded systems. We wish to propose an implementation of Star-Cubing for this architecture. We will consider the case that sensors are arranged in a lattice-style configuration (as visualized in the paper), as a non-lattice configuration would require a non-trivial approach to aggregation along spatial dimensions. In this case, we consider the dimension of the lattice network to be n=1, 2, or 3. We will consider sensors that measure one attribute (e.g. temperature). In this sense, we have an dimensional data set, if we consider recorded data as a set, or we have an dimensional data set if we let time measurements be an attribute.

If the system takes measurements synchronously, then for data aggregation using count, we know what cuboids surpass the iceberg condition by considering the number of samples recorded at that time and the geometry of the lattice.

For the synchronous system, the iceberg condition using count is nontrivial. For each sensor, we can consider if it meets the minimum support, and if so, it can aggregate its values directly. Then by parallel transmitting those aggregates along each dimension of the lattice, more abstracted cuboids can be computed simultaneously. To compute the apex cuboid, we necessarily need to repeat this process across each dimension. In this way, as we compute partial aggregations across each dimension, we have the potential to begin considering the star tree before it is processed.

Consider ordering the attributes by , where T is time, S is a sensor measurement that we wish to aggregate over, and X,Y,Z correspond to some spatial orientation. First, we consider aggregates over time, so we abstract over T at each sensor. S can arbitrarily be a vector of independent quantities dependent on T,X,Y,Z, but for analysis it suffices to consider only a scalar quantity. We have the \*XYZ cuboids computed stored at each node. For those that pass threshold, we can transmit relevant information in parallel down the X direction. At one side of the lattice, the terminal sensors can aggregate over X to compute partial star tables for \*\*YZ cuboids. Transmitting over Y direction is a bit different in that first \*XYZ cuboids are shared, to compute \*X\*Z, and then \*\*YZ cuboids are transmitted to compute \*\*\*Z. Finally, transmission over Z occurs in four steps, first \*XYZ to get \*XY\*, then \*\*YZ to get \*\*Y\*, then \*X\*Z to get \*X\*\* and finally \*\*\*Z to produce the apex \*. With a complete star table made, the star tree can be generated and relevant data queried in a similar manner.

# Conclusion

We implemented the Star-Cubing algorithm in Python and considered its performance across various data sets. We note the importance of choosing a proper threshold for ensuring a desired execution time. Based on our implementation, there is a necessary memory usage based on the environment and import of Python libraries. Further revisions of this code would expand upon this version by reducing the overhead memory usage by examining these factors more closely. Finally, we considering a possible method to adapt the algorithm to a non-Von Neumann architecture.

# References

1. D. Dua, C. Graff. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.
2. D. Xin, J. Han, X. Li, B. W. Wah. “Star-Cubing: Computing Iceberg Cubes by Top-Down and Bottom-Up Integration.” VLDB’03.
3. J. Han, M. Kamber, J. Pei. (2012). Data Cube Technology, *Data Mining: Concepts and Techniques* (3rd ed., pp. 187-242). Waltham, MA: Morgan Kaufmann Publishers.
4. A. Umbarkar, V. Subramanian, A. Doboli, “Linear Programming-based Optimization for Robust Data Modeling in a Distributed Sensing Platform”, IEEE Transactions on CADICS.