# Improving J48 Memory Efficiency by Implementing Sparse Matrices

Michael Frew

Biomedical Engineering
Department of Systems Design Engineering, University of Waterloo

*Abstract* – The increasing number of cybersecurity threats necessitates the use of complex algorithmic, machine learning, and deep learning models to filter noise and identify bad actors. J48 is a tree-based machine learning model used extensively to flag suspicious security events. The implementation prioritizes speed and simplicity at the expense of being highly inefficient with nominal attributes. This report outlines the steps taken to decrease the memory used by the model in training and classification. Data manipulation and feature hashing was considered but did not yield the desired results. J48 was then reimplemented to J49, which stores data in a sparse representation using hash maps. Also, it was implemented with a design conducive to future improvements and fast testing of new sparse array storage methods. Other methods were briefly explored that could yield further reductions in size. These techniques greatly decreased the size of the models yielding 10-100x improvements in space efficiency.

*Keywords* – Cybersecurity, Machine Learning, Sparse Matrices

## I. SITUATION OF CONCERN & PROJECT OBJECTIVES

In the 21st century, cyberattacks are alarmingly common and continually increasing with a record breaking 50% rise from 2020 to 2021 [1]. Security Information and Events Management (SIEM) systems collect, normalize, and analyze events from various sources to identify malicious activity in real time [2]. To identify a threat, suspicious activity is first flagged by machine learning (ML) threat detection algorithms, and these flags are then reviewed by a team of analysts. These models must be robust and efficient to facilitate the classification of thousands of events per second and continuous training.

Our company uses ML algorithms to filter events and detect suspicious activity to be reviewed by a team of dedicated analysts. Waikato Environment for Knowledge Analysis (WEKA) is an open source, graphical user interface (GUI) and collection of machine learning algorithms [3]. At our company, many of the ML models are built with WEKA, since it is simple to import into the Java native SIEM and because the GUI allows for efficient data mining and testing.

One commonly used decision tree algorithm from WEKA is J48, which is an optimized version of C4.5. It constructs its tree on training data using information gain and gain ratio as the decision heuristics [4]. Information gain refers to the decrease in entropy of a dataset after a transformation of some sort (in this case partitioning). Entropy can be thought of as the randomness of a dataset; in the context of ML, this refers to how diverse the class values are. The J48 algorithm forms a decision (or split) by calculating the entropy of the system, and the entropy values that would result from each possible split. Taking the difference between these two values, the information gain can be maximized, and the ideal split is chosen. This process repeats recursively until the data is sufficiently grouped and the decision tree is complete.

One issue faced by many decision tree classifiers in SIEMs is that the data being processed contains nominal attributes with high cardinality. A nominal attribute is one where the possible values are names, categories or codes, without any ordering. They are also called categorical attributes. Cardinality in this context is the number of unique names or categories that show up in a specific attribute.

When J48 is trained on high cardinality datasets the amount of memory used increases significantly until the heap is full, resulting in a program crash. Obviously, this must be avoided, so currently all tree models have a capped nominal attribute cardinality of 400 to conserve memory. Any labels past the first 400 are therefore considered 'unusual' by the system and have been overloading the system with alerts. In one instance, a month's worth of data contained more than 40 000 different labels! The issue can be solved by muting the predictors, but this is only a temporary measure. Therefore, the question was posed: Can the J48 algorithm be re-implemented in a much more efficient manner? A successful solution will use much less memory (in bytes) while having classification and build times similar to the original J48.

## II. DESIGN METHODS

With a clear goal: to make J48 decision trees more efficiently, the first step of the design process was to conduct research to gain a thorough understanding of the problem.

*Code Analysis*

Prior to the undertaking of this project, personal study was conducted to learn the fundamental workings of the C4.5 decision tree classifier (which J48 is based). The next step was to delve into the J48 algorithm, starting with the WEKA framework for classifiers. To get an initial feel for the software, the GUI was downloaded and tested with various training datasets. Next, a 'trivial classifier' (does not perform any important function) was implemented. This was to learn the structure of the general WEKA classifier as well as the way that instances (the data) is handled.

After the various methods, default properties, and storage methods of the data was understood, Unified Modelling Language (UML) diagrams were generated from the source code to visualize the packages of interest. J48 is made of 21 classes, so by creating UML diagrams the overall structure of the model could be quickly visualized and understood. Small tests were then run to generate J48 models within the IntelliJ debugger tool (an Integrated Development Environment, or IDE). By this, the fields and objects contained within the model could be expanded and examined with the goal of uncovering unnecessarily stored values and inefficiencies.

The Distribution class was found to store values very inefficiently. The class stores the distribution of the class values by 'bags' which are subsets of the larger dataset. An array of double arrays called 'perClassPerBag' is used to store these values. As the dataset is partitioned, each 'bag' contains only a few class labels, yet the dimension of 'perClassPerBag' is large enough so that weights can be stored for all class labels. The result is that the matrices within the Distributions store many zeros. This can be called a 'sparse' matrix or a matrix with low density [5].

| | *0* | *1* | *2* | *3* | *4* | *5* |
|---|---|---|---|---|---|---|
| *0* | 1 | 0 | 0 | 2 | 0 | 0 |
| *1* | 0 | 0 | 3 | 0 | 0 | 0 |
| *2* | 5 | 4 | 0 | 0 | 0 | 3 |
| *3* | 0 | 0 | 1 | 0 | 0 | 0 |
| *4* | 0 | 0 | 0 | 0 | 0 | 0 |
| *5* | 0 | 0 | 0 | 0 | 8 | 0 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| [row] | 0 | 0 | 1 | 2 | 2 | 2 | 3 | 5 |
| [col] | 0 | 3 | 2 | 0 | 1 | 5 | 2 | 4 |
| [val] | 1 | 2 | 3 | 5 | 4 | 3 | 1 | 8 |

Figure 1. A sparse matrix with 8/36 values being non-zero (L) and its custom compressed sparse row representation (R) [Source: MF, 2022]

Figure 1 is an example of a sparse matrix since the number of values that store information are much less than the size of the matrix. In this case, the density, which is measured by the number of non-negative values divided by the total number of possible values, is 22%. As shown, by storing this data in a different format, the total dimension can be decreased from 36 (6x6) to 24 (3x8).

*Brainstorming and Planning*

The storage of J48 models built on high cardinality was clearly the root cause of the space inefficiency so the next step was to determine a plausible solution to the issue. The first approach taken was changing the way that the data is preprocessed. This is because it does not require changing the structure of any of the models already in production but just the form of the data that these models run on. If it worked, it would have been the most cost-effective solution. Techniques such as One-Hot-Encoding and feature hashing were explored to change the cardinality and dimensionality of the input data, which yielded some positive results, but still did not solve the issue of high cardinality in the class labels [6] [7]. It was then decided to explore reimplementation of J48.

The new implementation was planned to be the same algorithm as J48 with key changes to the storage of Distributions. Essentially, the plan was to create Distributions that do not store zeros, but rather have zeros be the default value to save space. Libraries and packages were explored to learn which options were already available online.

*Proof of Concept*

Before a full solution, it was decided that the concept should be proven at a small scale to prevent wasting time and to have a better understanding of the problem before starting development. To do this, a miniature solution was created separate from the WEKA package. First, a Distribution interface was made that encapsulates the core functionality. This way, multiple versions could be tested against each other with the same signature.

Two unique implementations were tested: A custom compressed sparse row (CSR) version shown in Figure 1, and one using hash maps [5]. The hash map implementation is shown in Figure 2.

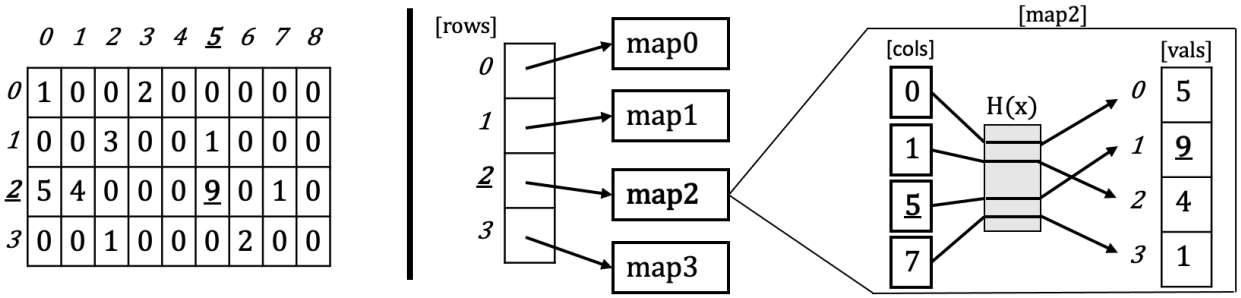


Figure 2. A sparse array (L) and its custom hash map representation (R) where H(x) denotes a hash function [Source: MF, 2022]

A hash map was created for each row in the original matrix, but the sizes of the hash maps are dependent on the number of entries per row, as shown. the original Distribution was implemented to act as a control to compare the test versions. Unit tests were created iteratively to ensure the test classes functioned as intended. A random data generation function was made to test the different implementations on build time, method times, object size, and serialized object size while varying the bags, classes, and density percentage. All data generation and tests were completed with Java and visualized in excel.

## III. DESIGNED SOLUTION

The chosen solution (referred to as J49) involved reimplementing J48 so that the Distributions store hash maps instead of arrays. Some files from J48 could be copied to the new package, but most classes required refactoring. Table 1 shows the key differences in the Distribution class fields.

Table 1. Internal fields of J48 and J49 Distribution Class where 'B' equals the number of bags and 'C' equals the number of class labels [Source: MF, 2022]

| Field | Purpose | J48 type | J49 type |
|---|---|---|---|
| perBag | Stores weightings per bag | double[B] | double[B] |
| perClass | Stores weightings per class | double[C] | hashmap |
| perClassPerBag | Stores weightings by class and bag | double[B][C] | hashmap[B] |
| total | Stores the total weights | double | double |
| numClasses | Stores the total number of classes | N/A | int |

As shown in Table 1, the 'perBag' and 'total' fields did not change between implementations. The 'perClass' field in J48 was an array with the index being the class, and the corresponding stored value the weighting. In code, this can be written as:

$$perClass[class] = value$$

In J49, key-value pairs are stored instead in the form:

$$\{class, value\}$$

The benefit of this is that since the hash does not use a contiguous block of memory like the primitive array, zero values are assumed as the default and do not need to be stored. The field

'perClassPerBag' functions in the same way, wherein J48 has a list of arrays, J49 has a list of hash maps. Finally, the 'numClasses' field had to be added to J49 because in J48 it is stored as the length of 'perClass'.

The decision to use hash maps was made after considering the pros and cons of both options. The benefits of the CSR implementation are that it is the most memory efficient at very low densities (<40%) and that it has similar speeds to the standard version at within this range. However, most of its methods grow linearly with the size of the array; construction requires nested loops; it performs very poorly at only high density matrices; and the array must either be resized manually or alternatively if the prebuilt ArrayList class is used, the autoboxing must be used which is slow and memory intensive. The hash map implementation showed consistent improvements over J48 at a larger range of densities, while maintaining the same 'Big O' time complexities as the original version. Unfortunately, this implementation also had tradeoffs: occasionally the maps must rehash which is slow; it is not as efficient as the CSR implementation at very low densities (<10%); and most operations, despite being constant, are slightly slower than the original version.

Through weekly meetings with Senior Software Architect, Mitch Rybczynski, conducted throughout the design process, it was determined that the hash map was the best solution to implement since it was the best balance of improved performance as well as consistency. As previously mentioned, the custom CSR implementation is much more error prone while only providing benefits when data density is within a certain threshold. Consistency was prioritized and so from the pros and cons listed, the hash map implementation was the clear winner.

However, despite the hash map implementation being the most favourable, additional functionality was built into J49 to allow for comparison between different distributions quickly and easily. The Distribution methods were extracted into an interface, so that the implementation is fully abstracted from the functionality in J49. The logic of selecting the Distribution type was then encapsulated into a factory – 'DistributionFactory' [8].

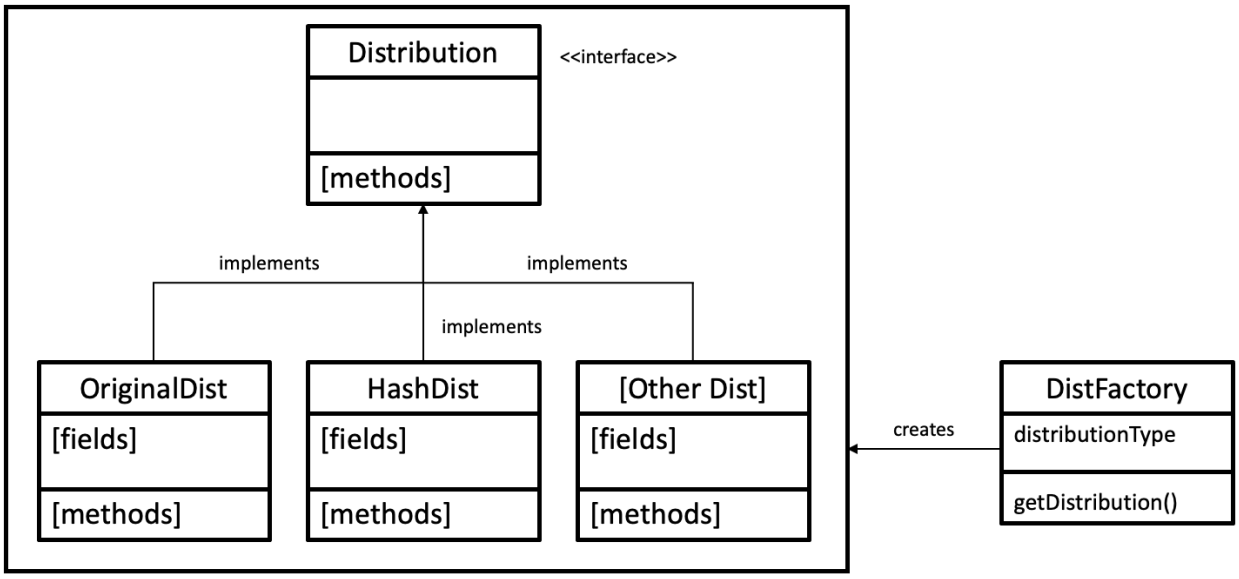


Figure 3. Diagram showing new organization of Distributions using a factory design pattern [Source: MF, 2022]

Within J49, all references to the original Distribution were changed to the interface, and constructors to the corresponding 'getDistribution()' method. One Distribution Factory is created per J49 model that determines the structure of Distributions throughout the entire tree by storing a reference at each node. While implementing this design pattern was more work up front compared to just replacing the Distribution class, it was very worthwhile when implementing and testing multiple solutions simultaneously and allows for easy reimplementation in the future.

## IV. DESIGN VERIFICATION AND VALIDATION

Test cases for the designed solution were written with three different objectives: to unit test the new Distribution implementation; to test the functionality of the overall J49 model; and to test for functional improvements in efficiency over J48. Unit tests were written to ensure individual methods of the new Distribution versions and the factory worked as expected. To test the functionality of the model, predictions were compared with a J48 model trained on the same data. This was repeated with all sample WEKA datasets as well as on two larger datasets taken from the company's SIEM. For validation, models were built and tested on varying amounts of SIEM data in a hashed and raw format. During these tests, the build time, classification time, runtime size, and serialized model size were recorded. To make the results more understandable, they were graphed, and key benchmarks were also converted to factors of improvement relative to the J48 model.

Finally, the J49 model could be loaded into the WEKA GUI to allow for easy testing on different datasets. Table 2 shows the results obtained after building models on the same data in hashed and raw (un-hashed) formats.

Table 2. Results from tests in WEKA GUI (n=2) on SIEM data [Source: MF, 2022]

| | Hashed Data | | | Raw Data | | |
|---|---|---|---|---|---|---|
| Model | Build Time | Test Time | Model Size | Build Time | Test Time | Model Size |
| J48 | 13s | 8s | 16 MB | 11s | 4s | 291 MB |
| J49 | 15s | 8s | **1 MB** | 2s | 5s | **3 MB** |

From Table 2 it is clear: J49 is vastly more efficient in terms of size, using about 16x less memory on the hashed dataset and close to 100x less on the raw data! The build and test times were comparable, with J49 being slightly slower in all tests except for the build time on the raw dataset. This is a positive result because it means J49 will at the very least be a more efficient replacement for J48 and in some cases also reduce computation time!

Currently in production, J48 models are trained on hashed data to avoid running out of memory. This hinders the understandability of the decision trees reasoning. These results prove that the raw data can be used, improving model understanding, build time, test time, and memory efficiency.

### a) LIMITATIONS OF METHODS USED AND DESIGNED SOLUTION

A major limitation to the method of testing is that many of the datasets were small and generic – only two of the sample datasets were taken from production. This limits the level of

confidence in the solution because the scope of testing is quite smaller. However, in the proof-of-concept design stage, many custom datasets were created and yielded positive results in testing. Further, the datasets from production are from one of the models that suffered the most from high cardinality attributes.

A limitation of design of J49 is that when high cardinality splits are made, there may be hash maps initialized that are never filled which leads to unnecessary overhead. This risk is quite low, however, because the gain-ratio metric prevents bias for splitting on high cardinality attributes. Further, the implementation has the interface to allow for simple and efficient transition to new implementations of the Distribution. Another downside to the current design is the risk of rehashing when instances are added individually. When the Distribution is constructed with all instances, it is optimized to fit the number of instances. However, if each instance is added individually then there is the chance that the map will need to rehash multiple times which takes up extra time and memory. Linked-lists could be used in place of the contiguous implementation of arrays, but this introduces the new problem of auto-boxing of primitive integers into objects.

## b) CONCLUSIONS

The designed solution, J49, is much more efficient than J48. With increasing levels of cyber risk, it is ever more important to have ML models that can separate signals from noise. To achieve this, the amount of data that must be considered in training has increased, so although J48 works well on small datasets, J49 allows much larger trees to be built. The tests showed that J49 memory footprint is 1-2 order of magnitudes smaller than J48 depending on the specific tree structure while maintaining speed. This decrease in memory usage allows fewer machines to run the same number of models as well as an increase in the size of these models. This improvement removes the need to use feature hashing which in turn improves the explainability of the model's output. Furthermore, the code structure of J48 was improved to allow for smooth customization in future. J49 can do everything J48 could, and more, while being more efficient and better structured for future improvements.

## ACKNOWLEDGMENTS

I am very thankful to my supervisor and teammates who provided valuable insight and guidance throughout the design process. Specifically, my manager, Mitch Rybczynski, and teammates: Owen Fairholm, Andrew Lukasik, Rudra Sharma, and Katy Zhao. Your continued support is strongly appreciated.

# REFERENCES


[1] Stealth Labs, "Number of Cyberattacks in 2021 Peaked All-Time High," 22 January 2022. [Online]. Available: https://www.stealthlabs.com/news/cyberattacks-increase-50-in-2021-peaking-all-time-high-of-925-weekly-attacks-per-organization/. [Accessed April 2022].

[2] S. Bhatt, P. K. Manadhata and L. Zomlot, "The Operational Role of Security Information and Event Management Systems," *IEEE Security & Privacy,* vol. 12, no. 5, pp. 35-41, 15 October 2014.

[3] M. A. H. I. H. W. Eibe Frank, Data Mining: Practical Machine Learning Tools and Techniques, vol. Fourth Edition, Burlington, Massachusetts: Morgan Kaufmann, 2016.

[4] S. P. Farhad Alam, "Comparitive Study of J48, Naive Bayes amd One-R Classification Technique for Credit Card Fraud Detection using WEKA," *Advances in Computational Sciences and Technology,* vol. 10, pp. 1731-1743, 6 November 2017.

[5] S. Pissanetzky, Sparse Matrix Technology, vol. 1, Cambridge, Massachusetts: Academic Press, 1984, p. 336.

[6] P. F. A. T. S. Adil Yousef Hussein, "Enhancement Performance of Random Forest Algorithm via One Hot Encoding for IoT IDS," *Periodicals of Engineering and Natural Sciences,* vol. 9, no. 3, pp. 579-591, August 2021.

[7] C. S. Q. S. A. v. d. H. D. S. Guosheng Lin, "Fast Supervised Hashing with Decision Trees for High-Dimensional Data," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition,* pp. 1963-1970, June 2014.

[8] R. H. R. J. J. V. Erich Gamma, Design Patterns: Elements of Reusable Object-Oriented Software, Boston: Addison-Wesley Professional, 1994.

[9] Amazon, "Amazon EC2 On-Demand Pricing," Amazon Web Services, 2022. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/. [Accessed April 2022].


Image References:

- *Images presented in this document created by the authors are indicated by author initials and the year of creation.*