Using Machine Learning to Predict Effective Compression Algorithms for Heterogeneous Datasets

Brandon Alexander Burtchell and Martin Burtscher

Department of Computer Science
Texas State University
San Marcos, TX 78666, USA
{burtchell,burtscher}@txstate.edu

Abstract

Heterogeneous datasets are prevalent in big-data domains. However, compressing such datasets with a single algorithm results in suboptimal compression ratios. This paper investigates how machine-learning techniques can help by predicting an effective compression algorithm for each file in a heterogeneous dataset. In particular, we show how to train a very simple model using nothing but the compression ratios of a few algorithms as features. We named this technique "MLcomp". Despite its simplicity, it is very effective as our evaluation on nearly 9,000 files from a heterogeneous dataset and a library of over 100,000 compression algorithms demonstrates. Using MLcomp to pick one lossless algorithm from this library for each file yields an average compression ratio that is 97.8% of the best possible.

1 Introduction

A heterogeneous dataset is a dataset consisting of files or groups of files that are best compressed with distinct algorithms. Heterogeneity is prevalent in many big-data domains (e.g., medicine [1] and IoT [2]) where the need for effective data compression is high. However, compression algorithms tend to exploit patterns and redundancies that are specific to a particular type of data. In fact, there exists no lossless algorithm that compresses all files well [3]. Thus, when attempting to compress a heterogeneous dataset with a single algorithm, the average compression ratio is likely lower than it would be if each type of file were compressed with a specialized algorithm.

This paper explores simple techniques to predict effective compression algorithms for each file in heterogeneous datasets. Moreover, it investigates ways to make such predictions without knowledge of the type of data being compressed to demonstrate the method's generality. We named our approach "MLcomp" as it is based on machine-learning (ML) techniques—namely sequential feature selection (SFS) [4] and nearest-neighbor (1NN) classification [5]. The result is a model that can pick an effective compression algorithm for an input file out of a large selection of compression algorithms. On our evaluation dataset, MLcomp reaches 97.8% of the compression ratio (CR) achieved when exhaustively searching for the best of 103,488 algorithms. This paper makes the following main contributions.

- It outlines a simple method for narrowing down a search space of over 100,000 algorithms to a single well-performing algorithm for any given input.

- It demonstrates that the CRs of a few short compression pipelines make surprisingly effective features for predicting near-optimal longer pipelines.
- It shows that prudently selected features can accurately distinguish the type of each file in a heterogeneous dataset.

The remainder of this paper is organized as follows. Section 2 describes MLcomp in detail. Section 3 discusses related work. Section 4 outlines the evaluation methodology. Section 5 analyzes our results. Section 6 provides a summary and conclusions.

2 MLcomp Approach

This section describes the operation of MLcomp. Subsection 2.1 explains the training and Subsection 2.2 the prediction. Subsection 2.3 walks through a simple example.

2.1 Training Procedure

Suppose we have a heterogeneous dataset that we want to compress. Further suppose we have a library of algorithms from which we are allowed to choose any algorithm to compress each file in the dataset. We could exhaustively run each algorithm on every file and keep the smallest output, but this would be too slow for large libraries. Instead, we train an ML model off-line to minimize the amount of computation needed later when we want to predict an effective compression algorithm for a given file.

We perform the following steps to train the model. First, we select a subset of the files from the dataset and split the subset into a training and a validation set. For each file in the training set, we run all compression algorithms from our library to determine the one that maximally compresses the file. We use the term target to refer to a file's best compression algorithm. We also extract features from each file. They can be any quantitative property (e.g., the entropy, the file length, or the number of '1' bits). The features we use in MLcomp are discussed further down.

Once the feature values and target algorithm of each training file have been computed, we begin the SFS procedure to greedily choose a small combination of features that yields the best predictions. First, we separately train a 1NN model (explained below) for each feature on the entire training set. Next, we use each resulting model to predict an algorithm for each file in the validation set. Finally, we pick the feature whose 1NN model yields the highest geometric-mean CR. Once we have identified the best feature in this manner, we repeat the procedure by combining the chosen feature with every other feature, one at a time, to find the best pair of features. We continue until we have enough features selected. The end result is our feature vector. The rest of this subsection describes key aspects of the training procedure in more detail.

In MLcomp, the library of compression algorithms is generated by CRUSHER [6], a tool that synthesizes compression and decompression algorithms by chaining coders into pipelines. For this purpose, CRUSHER contains a collection of encoders and matching decoders. Our study uses 57 encoders, of which 33 are compressors. Note that a pipeline's last stage must be a compressor. Our target pipelines have a length $l = 3$ (excluding `NUL`, the identity coder), yielding $56 \times 56 \times 33 = 103{,}488$ compression

algorithms. The features we use in MLcomp are also CRUSHER-generated pipelines but of length $p = 2$ (including `NUL`), yielding $57 \times 33 = 1881$ feature pipelines.

The feature value for a given file is the inverse CR, that is, the compressed file size divided by the uncompressed size. Using the inverse normalizes features values to the range $(0, 1]$, which ensures every feature is informative in the 1NN search.

Training a 1NN model simply consists of computing and storing the feature vector and target of each training file. To make a prediction for a new input, we compute the input's feature vector and search the trained model for the nearest neighbor using the Euclidean distance. Then, we retrieve the target algorithm stored in that nearest neighbor and use it to compress the input.

Note that the number of distinct pipelines stored in the 1NN model is necessarily less than or equal to the number of training files. In our case, the number of pipelines is, therefore, much lower than the size of CRUSHER's search space. This means that the best pipeline for an input may not be present in the model. In other classification problems, this would be a serious issue. However, our goal is not necessarily for a prediction to match the input's target but only to find a pipeline that achieves a CR that is as close as possible to that of the target. Consequently, as long as the training data is representative of the dataset and we select the right features, the trained 1NN model should predict pipelines that compress near the target CR.

For 1NN classification problems, one should minimize the number of features to avoid the curse of dimensionality [7]. Moreover, not every feature is guaranteed to help. In MLcomp, reducing the number of features has the added benefit of lowering the amount of computation needed at prediction time since each feature pipeline must be run on the input. For these reasons, we perform the following SFS procedure.

We start with an initial set of features $X = \{x_1, x_2, ..., x_m\}$. Via forward SFS, we greedily select the subset Y with $n \leq m$ features from X that yields the highest score. To begin, we initialize the working feature set to $Y = \emptyset$. In every iteration, we consider each feature in X a *candidate feature*. We train a corresponding candidate 1NN model on the training set with the current working feature set Y plus the candidate feature. We evaluate the resulting candidate 1NN model by scoring it on the geometric-mean CR of its predictions on the validation set. The candidate feature that maximizes the score is added to the working set Y and removed from X. We repeat this procedure until $|Y| = n$, that is, until Y contains n features. Backward SFS works similarly but starts with $Y = X$ and iteratively removes features until $|Y| = n$. The resulting feature set Y is used to train the final 1NN model.

2.2 Prediction Procedure

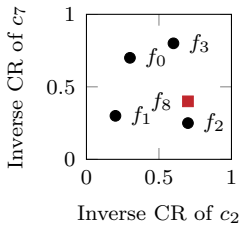
Since most of the computation has been offloaded to the training phase, the prediction procedure for an input is relatively quick. For an input file, we compute the feature vector (i.e., the values of the features in Y) by running each feature pipeline on the input and recording the inverse CRs in a vector. Based on this feature vector, we search the trained model for the nearest neighbor by computing the Euclidean distance between the feature vector and each feature vector recorded in the model. Finally, we compress the input using the pipeline stored in the nearest neighbor.

2.3 Walkthrough

Suppose we have a heterogeneous dataset with 12 files $\{f_0, f_1, ..., f_{11}\}$ and 10 compressing coders $\{c_0, c_1, ..., c_9\}$ from which we can generate pipelines. In this example, we set the length of the feature pipelines to $p = 1$, the length of the target pipelines to $l = 2$, the size of the feature vectors to $n = 2$, and perform forward SFS. With 10 compressing coders and $p = 1$, we generate 10 feature pipelines $\{c_0, c_1, ..., c_9\}$. Similarly, with $l = 2$, we generate a library of 100 pipelines $\{c_0c_0, c_0c_1, ..., c_9c_9\}$.

For the sake of this example, we split the dataset evenly into the training set $\{f_0, f_1, f_2, f_3\}$, the validation set $\{f_4, f_5, f_6, f_7\}$, and the test set $\{f_8, f_9, f_{10}, f_{11}\}$. For each training file, we first determine the target (i.e., the pipeline that compresses the best) by running all 100 pipelines from the library and compute the feature values by running the 10 feature pipelines and recording the resulting inverse CRs.

Next, we perform forward SFS. We start with the feature set $X = \{c_0, c_1, ..., c_9\}$, the working feature set $Y = \emptyset$, and consider each feature in X a candidate feature. We train a corresponding candidate 1NN model on the training set with the current feature set Y (which is empty in the first iteration) plus the candidate feature. We evaluate the candidate 1NN model by scoring it on the geometric-mean CR of its predictions on the validation set. Suppose that, in the first iteration, c_2 is the candidate feature that scores the highest. We add c_2 to Y and remove it from X. Since $|Y| < n$, we perform another iteration. Suppose that c_7 scores the highest in the second iteration. We add c_7 to Y and remove it from X. Now the SFS terminates as $|Y| = n$. The resulting feature set $Y = \{c_2, c_7\}$ is used to train the final 1NN model.



(a) 1NN Feature Space

Training File	Target Pipeline
f_0	c_2c_3
f_1	c_4c_7
f_2	c_6c_1
f_3	c_2c_3

(b) Training File to Target Pipeline

Figure 1: Visualization of MLcomp Model after Training

Fig. 1 shows the information in the final model. In Fig. 1a, the four files from the training set are represented as four points at their coordinates (c_2, c_7). Table 1b shows the target pipeline of each training file. In this example, the library of 100 pipelines is reduced to 3 distinct in-model pipelines (fewer than the number of training files).

Now, we use the trained model to compress each input in the test set. We illustrate this on input f_8. To start, we compute the feature vector by running c_2 and c_7 on f_8 and utilize the resulting inverse CRs as the feature values. Assume the resulting feature vector for f_8 to be (0.7, 0.4), represented by the square in Fig. 1a. Next, we search the model for the nearest neighbor of f_8, which is f_2 based on the Euclidean distance. According to Table 1b, the target pipeline of f_2 is c_6c_1, so we compress f_8 with the algorithm c_6c_1. We repeat the process for the remaining test files.

3 Related Work

This section summarizes prior work on lossless compression of heterogeneous files or datasets, usage of ML for lossless compression, and prior applications of CRUSHER.

Hsu and Zwarico present a technique for automatically synthesizing compressors for heterogeneous files [8]. Each file is separated into chunks, then a statistical method is employed to determine what algorithm to use on each chunk. This is akin to how MLcomp computes features to determine the best algorithm. However, we do this at file granularity and use simple compression algorithms as features.

Kattan and Poli address heterogeneity by using genetic programming to combine standard compression algorithms [9]. Our work differs in that MLcomp selects algorithms from a library of synthesized pipelines and does not combine compressors.

Townsend et al. propose BB-ANS, which, with a trained latent variable model, is able to losslessly compress image datasets with impressive compression ratios [10]. However, BB-ANS does not target heterogeneous files or datasets.

Coplin et al. used an earlier version of CRUSHER to generate lossless pipelines for the same satellite dataset [11] (see Section 4). They proposed a method in which CRUSHER is periodically run on Earth to determine the best pipeline for each type of data based on recently downloaded files. The resulting set of pipelines is then transmitted to the satellite to be used until the next set is sent. Our work differs by only training once, avoiding repeated CRUSHER runs. Also, our approach works at a finer granularity, predicting effective pipelines per file rather than per type of data.

Burtscher et al. developed FPcrush, a modified version of CRUSHER that synthesizes lossless pipelines for floating-point data in real-time using a genetic algorithm [6]. MLcomp differs in that it predicts and selects algorithms from a fixed library.

4 Evaluation Methodology

To test our approach, we use data from the THEMIS-B spacecraft [12]. THEMIS-B has six scientific instruments that operate in multiple modes, producing 27 distinct data packet types, each of which is identified by a set of three hexadecimal digits. Every day, the satellite sends the recorded packets to Earth. The space probe also sends small housekeeping packets, which we exclude from our experiments.

THEMIS-B uses four on-board compression algorithms, each assigned to compress a different set of packet types. We include comparisons against these compressors to demonstrate the viability of using MLcomp on real data.

For our tests, we focus on measuring the compression ratio (CR), which is the original uncompressed data size divided by the compressed size. Our experiments use data collected in 2013 and 2014. We reserved January and February of 2013 (1,406 files) for training and March (775 files) as the validation set. To compare CR consistently, we tested the models on all data packets from 2014 (8,916 files).

We performed our experiments on a system with a Xeon E5-2687W v3 CPU running at 3.10 GHz and a main-memory size of 128 GB. The operating system is Fedora Linux 37. CRUSHER was compiled using GCC version 12.2.1 with the "`-O3`" flag. The MLcomp code is written in Python 3.11.1 and uses scikit-learn 1.2.2 [13].

5 Results

This section presents and discusses the experimental results. Subsection 5.1 compares the performance of our trained MLcomp model against several baselines. Subsection 5.2 analyzes the effects of SFS direction and feature pipeline length. Subsection 5.3 illustrates the correlation between the type of a data packet and the predicted pipelines. Subsection 5.4 compares the performance of MLcomp to that of the on-board THEMIS-B compressors on notable packet types.

5.1 Compression Ratio

We trained MLcomp to predict pipelines of length $l = 3$ using forward SFS with $n = 4$ features and a feature pipeline length of $p = 2$. To evaluate its effectiveness, we compare MLcomp against three categories of baselines: the best pipeline from our library for each file in the test set, the single best pipeline across the entire test set, and the single median pipeline across the entire test set. The median pipeline reflects the expected "average" performance when randomly picking a pipeline from the library. For each category, we perform two experiments: using the entire CRUSHER library of 103,488 pipelines and using just the 90 in-model pipelines. We also compare against the compressors used by the THEMIS-B spacecraft.

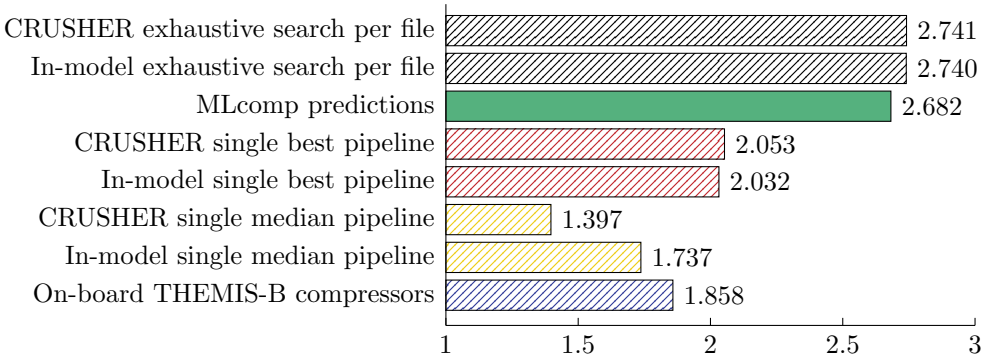


Figure 2: Geometric-mean Compression Ratio of MLcomp and Baselines

Fig. 2 displays the results. Longer bars indicate more compression. The various baselines are shaded. Since the results of the two exhaustive searches represent upper bounds, we show MLcomp in the third position.

Comparing the first three bars, we find that MLcomp achieves CRs that are very close to the upper bounds. In fact, it reaches 97.8% of the CR that is obtained when separately picking the best of our 103,488 pipelines for each file. Comparing the two upper bounds, selecting the best pipeline for each file from the 90 pipelines in the model reaches 99.9% of the CR achieved when picking from the full CRUSHER library. These results highlight three key points. First, even though our approach is simple (it uses one of the simplest ML models (1NN), only four features, and readily-available simple features), it performs close to optimal. Second, the training is effective and identifies 90 important pipelines out of over 100,000. Third, the prediction mechanism accurately picks 1 pipeline out of 90 that yields a near-optimal CR.

Next, we compare MLcomp to the two single best pipelines, that is, to using just one of the 103,488 algorithms in CRUSHER or one of the 90 algorithms in the model to compress all files. This yields two important observations. First, the best in-model pipeline results in a CR that is remarkably close to CRUSHER's best pipeline, which again shows that the training is accurate and able to identify a key pipeline. Second, using a single algorithm for compressing all files results in a much lower CR (just 77%) than MLcomp. This demonstrates that our dataset is heterogeneous, that there is no single pipeline in the library that predicts all our inputs well, and, most importantly, that the predictions of MLcomp are not trivial yet still pick effective pipelines.

Looking at the performance of the single median pipelines, we find that randomly selecting a pipeline, even from the models's reduced search space of 90 pipelines, is not an adequate method to achieve acceptable CRs. Moreover, it is again evident that the training procedure is effective and yields in-model pipelines that, on average, substantially outperform those from the complete library. Most importantly, the comparison of MLcomp to the median pipelines illustrates the need for and benefit of an intelligent prediction/selection scheme such as the one we present in this paper.

Lastly, we compare MLcomp's CR to that of the compression algorithms used by the THEMIS-B space probe. All of them run with only 64 kilobytes of memory [12]. We reflect this constraint by allowing CRUSHER to only generate algorithms that also adhere to this limit. Recall that THEMIS-B selects one algorithm out of 4 according to packet type, whereas MLcomp selects one algorithm out of 90 but without knowledge of the packet type. Nevertheless, MLcomp outperforms the THEMIS-B algorithms, demonstrating that its performance is good not only relative to various subsets of the 103,488 pipelines generated by CRUSHER but also in absolute terms.

5.2 Feature Vector Size, SFS Direction, and Feature Pipeline Length

In general, the direction of SFS affects the training time and the quality of the predictions. Whereas backward SFS takes much longer when selecting just a few features out of many, it can retain relationships between features and tends to yield better predictions. Fig. 3 shows the effect of SFS direction and the feature pipeline length p on the geometric-mean CR over the test set for different feature vector sizes n. We evaluate 33 feature pipelines for $p = 1$ and 1881 pipelines for $p = 2$.

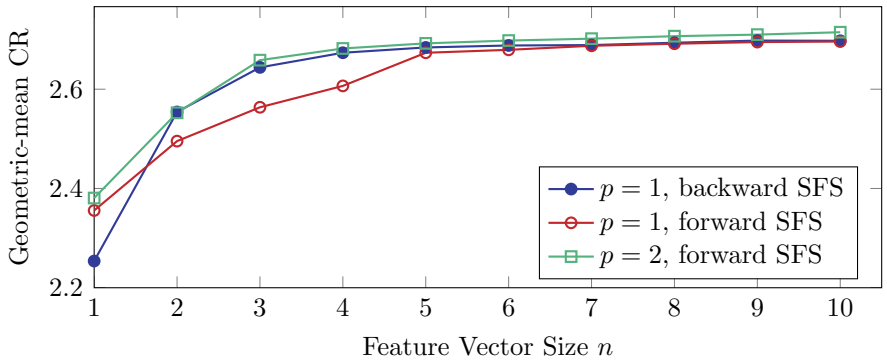


Figure 3: Performance of Three SFS Configurations for Different Numbers of Features

Regardless of the configuration, the geometric-mean CR increases as we increase n. However, the performance plateaus at 4 or 5 features. Note that we want to minimize n while achieving an acceptable CR since n determines the number of feature pipelines that must be run at prediction time. The $p = 1$ configurations demonstrate that, for small n, backward SFS performs better as expected. However, the forward SFS configuration with $p = 2$ beats both $p = 1$ configurations, so we chose it for MLcomp. Note that backward SFS with $p = 2$ would require over 1.77 million evaluations of every training input (compared to 7518 for forward SFS). We selected $n = 4$ since a larger feature-vector size would gain almost nothing but slow down the predictions.

5.3 Correlation between Packet Type and Predicted Pipeline

To explore the degree of heterogeneity our model retains from training, we generated the matrix in Fig. 4, which maps each packet type to the pipelines predicted by MLcomp during testing. We sorted the packet types (rows) by their hexadecimal value and the predicted pipelines (columns) alphabetically by their name. There are only 26 rows as one of the training packet types does not appear in 2014.

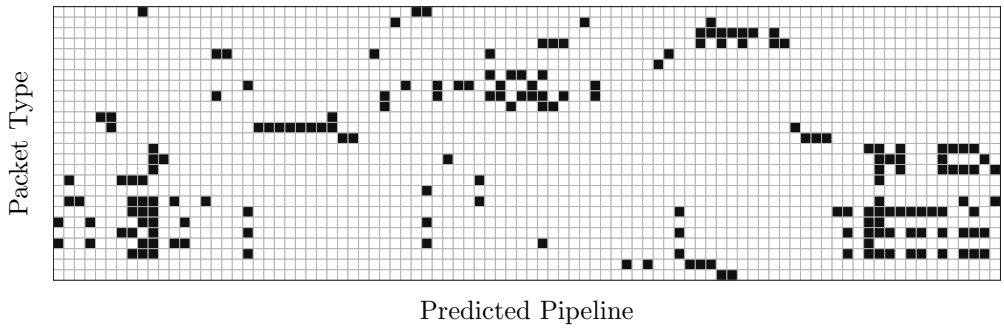


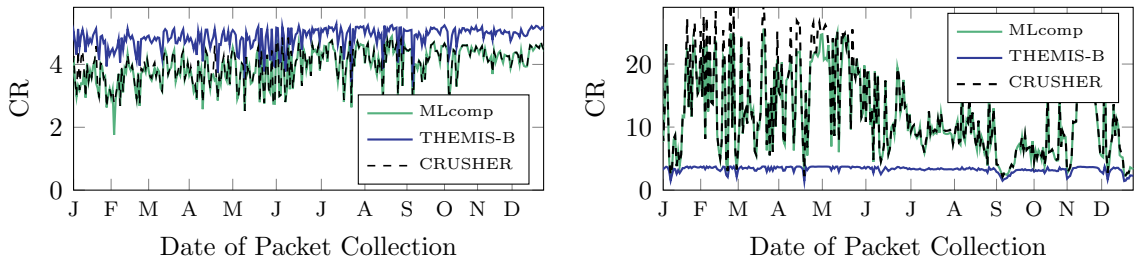
Figure 4: Correlation between Packet Type and Predicted Compression Pipeline

Out of the 90 available pipelines, MLcomp predicts a maximum of 16, a minimum of 1, a median of 7.5, and an average of 7.03 distinct pipelines per packet type. Considering that some packet types likely have similar features (e.g., because they are produced by the same instrument), overlap between packet types is to be expected. For example, there are several packet types towards the bottom of the matrix that tend to select similar pipelines to each other. That said, it is interesting to see that the distribution in Fig. 4 maintains such a strong discreteness between packet types.

5.4 Comparison with THEMIS-B Compressors per Packet Type

THEMIS-B uses a hardcoded algorithm for each packet type, but we purposely do not provide this information to MLcomp. Nevertheless, MLcomp achieves a higher average CR than THEMIS-B on average. Fig. 5a shows the CRs of each file with packet type '449' across the entire test set. 449 is the type where THEMIS-B beats the geometric-mean CR of MLcomp by the highest factor (1.2× higher). Fig. 5b shows the same for packet type '45f', where MLcomp beats THEMIS-B by the highest factor

(3.0× higher). We also plot MLcomp's upper bound (CRUSHER) in both charts. MLcomp reaches on average 98.9% of the upper-bound CR for 449 and 96.4% for 45f.



(a) Packet Type 449

(b) Packet Type 45f

Figure 5: Compression Ratio of Packet Type across Test Set

In Fig. 5a, THEMIS-B consistently beats MLcomp. Moreover, even the exhaustive search is unable to surpass THEMIS-B. This means that the sub-par performance of MLcomp is not due to a weakness of MLcomp but due to a limitation of the library of compressors, which does not contain any competitive algorithms for 449.

In Fig. 5b, MLcomp generally attains high CRs. Interestingly, the benefit over THEMIS-B fluctuates substantially. This is due to MLcomp selecting many different algorithms for compressing these files. In fact, 45f is tied for having the second-highest number (15) of predicted pipelines. Since the upper bound exhibits the same fluctuations, we know that the files within this packet type are themselves quite heterogeneous. As we can see, MLcomp is able to automatically exploit this property.

6 Summary and Conclusions

With access to a large library of compression algorithms (103,488 in our case), it is natural to desire an ML approach that learns how to predict/select an effective algorithm for compressing any given file. This is particularly important in heterogeneous environments where no single algorithm compresses all files well. However, choosing quantitative features that map to effective compression algorithms is non-trivial.

We demonstrate that compression algorithms themselves can be used for this purpose. With the straightforward SFS procedure, we are able to narrow down our 1881 simple candidate algorithms to just the 4 most effective features, essentially without loss in compression ratio. Based on these four feature algorithms, we train an ML model by recording, for each of our 1406 training inputs, the CRs of the feature algorithms along with the best compression algorithm from our library. This yields a 1406-entry table, where each entry contains 4 feature values and 1 algorithm. This makes predicting a good compression algorithm for a new input easy. We evaluate the CRs of the four simple feature algorithms on the new input and search the table for the nearest entry based on the Euclidean distance. The algorithm stored in that entry is then used to compress the new input.

We show that this approach yields a near-optimal CR on 8,916 unseen heterogeneous data packets from the THEMIS-B space probe. More broadly, we demonstrate

that a prediction scheme like ours is very useful on heterogeneous datasets, where using any single compression algorithm results in poor CRs. We also show that our approach surpasses the CR of the set of compressors used aboard THEMIS-B.

We believe the technique presented in this paper to be general and applicable to other heterogeneous datasets in archival (i.e., compress once and access many times) contexts. We hope our work will inspire others to explore methods of utilizing explainable ML to further improve data compression.

Acknowledgements

We thank Andrew Poppe from the Space Sciences Laboratory at UC Berkeley for providing the THEMIS-B data and codes. This work has been supported in part by the Department of Energy, Office of Science under Award Number DE-SC0022223.

References

[1] K. J. Cios and G. W. Moore, "Uniqueness of medical data mining," *Artificial Intelligence in Medicine*, vol. 26, no. 1, pp. 1–24, Sept. 2002.

[2] L. Wang, "Heterogeneous Data and Big Data Analytics," *Automatic Control and Information Sciences*, vol. 3, no. 1, pp. 8–15, Aug. 2017.

[3] J. A. Storer, *Data compression: methods and theory*, Number 13 in Principles of computer science series. Computer Science Press, Rockville, Md, 1988.

[4] F. J. Ferri, P. Pudil, M. Hatef, and J. Kittler, "Comparative study of techniques for large-scale feature selection," in *Machine Intelligence and Pattern Recognition*, vol. 16, pp. 403–413. Elsevier, 1994.

[5] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, Jan. 1967.

[6] M. Burtscher, H. Mukka, A. Yang, and F. Hesaaraki, "Real-Time Synthesis of Compression Algorithms for Scientific Data," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016, pp. 264–275, IEEE.

[7] R. Bellman, *Dynamic programming*, Princeton University Press, Princeton, NJ, 1957.

[8] W. H. Hsu and A. E. Zwarico, "Automatic synthesis of compression techniques for heterogeneous files," *Software: Practice and Exp.*, vol. 25, no. 10, pp. 1097–1116, 1995.

[9] A. Kattan and R. Poli, "Evolution of human-competitive lossless compression algorithms with GP-zip2," *Genetic Programming and Evolvable Machines*, vol. 12, no. 4, pp. 335–364, Dec. 2011.

[10] J. Townsend, T. Bird, and D. Barber, "Practical lossless compression with latent variables using bits back coding," in *International Conference on Learning Representations*, 2019.

[11] J. R. Coplin, A. Yang, A. R. Poppe, and M. Burtscher, "Increasing Telemetry Throughput Using Customized and Adaptive Data Compression," in *AIAA SPACE 2016*. American Institute of Aeronautics and Astronautics.

[12] V. Angelopoulos, "The THEMIS Mission," *Space Science Reviews*, vol. 141, no. 1-4, pp. 5–34, Dec. 2008.

[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.