TECHNISCHE UNIVERSITÄT DRESDEN

Dep. of Computer Science Institute for System Architecture, Database Technology Group

Study Thesis

LEARNED SELECTION STRATEGY FOR LIGHTWEIGHT INTEGER COMPRESSION ALGORITHM PARAMETERIZATIONS

Moritz Pflügner

Matr.-Nr.: 4677226

Supervised by:

Prof. Dr.-Ing. habil. Dirk Habich

and:

Dr.-Ing. Claudio Hartmann

Submitted on 20 October 2021

CONFIRMATION

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, 20 October 2021

ABSTRACT

With the continuing growth of stored data, data compression has become a common task in database systems regarding query processing or optimization. Among the large variety of existing lightweight integer compression algorithms, there is no single-best one. Thus, a selection strategy for finding suitable algorithms is necessary. In addition to the compression algorithm itself, the algorithm parameterization also influences the compression results. Hence, we present a *Learned Selection Strategy for Lightweight Integer Compression Algorithm Parameterizations* which extends existing Machine Learning approaches by considering both, the selection of the best-fitting algorithm and the parameterization leading to the best compression result. We evaluate our strategy against a baseline, point out advantages of our approach and explain the behavior of our Machine Learning models by analyzing feature importances. We show that the usage of our *Learned Selection Strategy for Lightweight Integer Compression Algorithm Parameterizations* lead to better compression results than using the simplest algorithm with a standard parameterization.

CONTENTS

1 INTRODUCTION

The amount of stored data has been increased exponentially during the last decade, reached 33 Zettabytes in 2018 and will grow to 175 Zettabytes by 2025 as the IDC [18] predicts. This growth leads to new challenges for data processing tasks in different applications like database systems [1] or Machine Learning [8]. Compressing the data and reducing the physical size of it has therefore become a crucial step for query executions or analytical tasks. Regarding database systems, column-organized data is usually encoded as sequence of integers [19]. Hence, the query processing is applied only to these integer sequence encoding. Compressing integer values prior to query processing leads to a better performance in terms of compression rate and ratio [20]. Although a large variety of lightweight integer compression algorithm exists, there is no single-best one [5, 6] as they all behave differently regarding input data and hardware properties. Additionally, integer compression algorithms can be initialized with certain parameters like the input format, output format, or the maximum bit width of the data being compressed. The parameterization also influences the behavior of the algorithm. Selecting the best combination of compression algorithm and parameters requires a suitable selection strategy. Certain approaches have been proposed that aim to solve this task. On the one hand, rule-based strategies are based on decision trees leading to the best-fitting algorithm [1]. Cost-based approaches on the other hand define cost functions for each algorithm and choose the one with the lowest costs [6]. Strategies based on Machine Learning do not require knowledge about the algorithm behavior as it is considered as a black box. The Machine Learning model learns the behavior from training data. Regarding the algorithm selection, Machine Learning approaches lead to the best results in comparison to a cost-based and rule-based approach [20]. However, none of the strategies considers the best-fitting parameterization.

In this thesis, we propose an extension of the *Learned Selection Strategy for Lightweight Integer Compression Algorithms* [20]. This extension additionally considers the best-fitting algorithm parameterization. For this, we describe the steps that are necessary in order to generate representative training data, derive features from them, create and tune the Machine Learning model, and evaluate the approach against a baseline. We show that applying a strategy which considers algorithm and parameterization leads to better compression results than using the simplest algorithm with parameters covering the largest range of data.

The thesis is structured in five chapters. Firstly, we describe general concepts of Lightweight Integer Compression, Algorithm Selection and Machine Learning in Chapter 2. Subsequently, we present the concepts and methods that will be used for the data generation process, feature engineering and hyperparameter tuning of the Machine Learning models in Chapter 3. In Chapter 4, we describe the specific implementation by presenting pipelines for the processes of data generation and validation. We evaluate the approach in Chapter 5 by using real-wold data on the one hand and by comparing it against a baseline strategy on the other hand. Finally, we conclude the thesis and give an outlook in Chapter 6.

2 PRELIMINARIES

The following chapter reviews and discusses related work of integer compression algorithms and adequate selection strategies in order to describe the problem the strategy presented in this thesis aims to solve.

2.1 LIGHTWEIGHT INTEGER COMPRESSION

Lightweight integer compression algorithms is a subject of current research mainly focusing on two different objectives. On the one hand, a lot of research aims to optimize the algorithm performance with different approaches like FPGA acceleration [17, 12] or novel kinds of query processing models [7]. On the other hand, the research focuses on analyzing the behavior of lightweight integer compression algorithms and finding of selection strategies [1, 6, 20]. Damme et. al [5, 6] have shown that among the large variety of lightweight integer compression algorithms, there is no single-best one. They all behave differently depending on the input data and hardware properties.
BitPacking (BP) is one of the most frequently used algorithms in this field. One advantage that leads to good compression rates is the adaptability to different bit widths [6].
The algorithm belongs to the group of null-suppression algorithms which means that the basic idea is the omission of leading zeros in the integer binary representation. Firstly, the binary representation has to be stored as a sequence. The omission of leading zeros in every sequence element results in blocks with different sizes for every value. The size is determined by the bitwidth of the largest value in the block. *BP* can be divided into *StaticBP* where every block has a predefined size and *DynamicBP* where the block size is determined dynamically.
Being able to analyse, compare and select lightweight integer compression algorithms in a specific context requires a non-technical abstraction. Hildebrandt et. al [11] developed the COLLATE metamodel and the description language COALA for lightweight integer compression algorithms and showed that every algorithm can be modeled with it. Their work resulted in the implementation of a software framework that allows the implementation of algorithms with the COALA language, their application to input data, and the measurement of properties like compression rate or (de)compression runtime. Furthermore, the algorithms can be defined with data-dependent

and data-independent parameters which change their behavior. The software framework offers the technical functionalities to observe the performance of an algorithm but for the selection of the best-fitting one, it is necessary to test all possible candidates with different parameter combinations manually and compare them according to compression rate or (de)compression runtime.

2.2 ALGORITHM SELECTION

As there is no single-best algorithm, it is necessary to determine the one that fits best for certain input data and hardware properties. In order to achieve that, an adequate selection strategy is necessary. In the literature, there are three major concepts. The rule-based strategy by Abadi et. al [1] is based on a decision tree for compression schemes. Certain data properties lead to specific compression algorithms. As Woltmann et. al [20] have already analysed, an advantage of their approach is the small tree size. The effort for traversing the tree is low, but the staticness of it makes it difficult to extend it to the current landscape of algorithms.
Damme et. al [6] developed a cost-based approach which is based on calibration measurements for each algorithm in order to retrieve the impact of data and hardware characteristics. This strategy has also been part of the analysis of Woltmann et. al [20]. They noted on the one hand, that it can take a wide range of algorithms into account, but adding new algorithms would always require manual effort on the other hand.
Based on their analysis, Woltmann et. al [20] proposed a third strategy combining the advantages of the rule-based one and the cost-based approach. Their main objective was the reduction of manual effort if a new algorithm is to be included into the selection set. Therefore, they designed their strategy as a black-box approach which does not need any information about the behavior or characteristics of the algorithm. In order to implement this, Machine Learning was used to train a model for each algorithm based on generated training data. After the training phase, the model was able to predict the cost of the algorithm when applied to certain input data.
As described above, lightweight integer compression algorithms behave differently due to specific input parameters. The learned selection strategy of Woltmann et. al [20] performed better than the cost-based one by Damme et. al [6], but it can only select the best-fitting algorithm and does not take parameterizations into account.

2.3 MACHINE LEARNING

Machine Learning (ML) has become a popular technique in database systems in order to manage and analyse data. Especially tasks like data sorting [15] or cardinality estimation [14] are suitable for it. Considering algorithm selection, only a few ML approaches exist. Jin et. al [13] proposed a selection strategy that considers the algorithm selection as a classification problem. During the training phase, they aggregate data with similar properties to data blocks and label them with the algorithm behavior. Afterwards, they predict the best-fitting algorithm by getting the data block with the lowest distance to the test data. Due to the usage of classification, adding a new algorithm would require a complete new training phase.
Boissier and Jenduk [3] developed a selection strategy based on Linear Regression and Gradient Boosting (GB) for the Hyrise database. Even though the GB model showed the best results, they

left out crucial validation aspects in their work.
Woltmann et. al [20] also used GB regression to train their model for the selection of the best-fitting algorithm. It is an ensemble method consisting of several weak learners, mostly decision trees. The training of every decision tree is based on the residuals of its predecessor. In comparison to the usage of Neural Networks, the times for training and forward passes are relatively low. The correct interpretation of a ML model requires the evaluation of feature importances. This aspect has not been a part of the work of Woltmann et. al [20] but would lead to a better understanding of the model decision which is crucial if a larger amount of algorithms are considered.
All existing ML approaches have in common that they only aim to select the best-fitting algorithm. None of them also considers the selection of algorithm parameterizations.

3 CONCEPTS AND METHODS

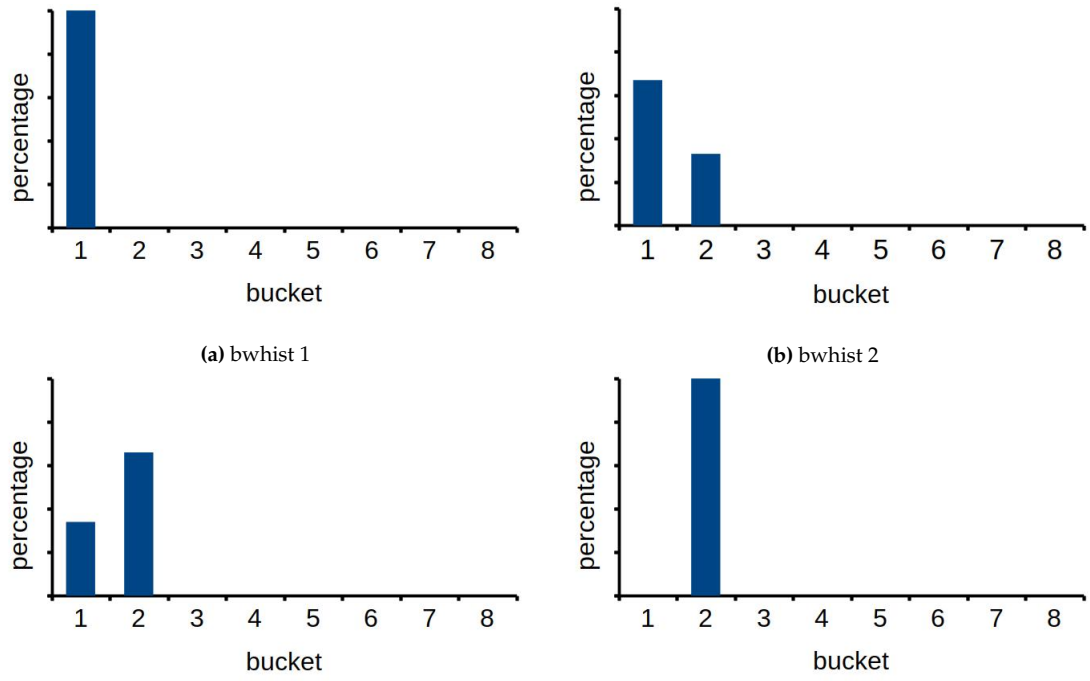
3.1 DATA GENERATION

In order to build a model which is able to predict the behavior of parameterized algorithms, representative training data is necessary. As in many other ML applications, this data does not exist initially which is called the cold start problem [20]. Solving this problem requires an adequate data generator producing representative integer values which then have to be labeled with the behavior of each considered algorithm i.e the compression rate and the compression runtime. Since all lightweight integer compression algorithms need integer sequences as input data, it is necessary to derive a representation of them. A common approach is the usage of *bit width histograms* (bwhist) [7, 20]. In an integer sequence, every value has a different bitwidth maximum depending on the size of the value. A bwhist of a sequence consists of b buckets where b is equal to the amount of bits that are used to represent the integer values of the sequence. Every bucket contains the percentual share of a specific bit width of all sequence elements.

Woltmann et. al [20] presented multiple ways to generate representative bwhists. They showed in their evaluation that the La-Ola generator performed reasonably well even though it is the most simple strategy. The generator starts with the bwhist having a full first bucket while all other buckets are empty. During each iteration step a fixed percentage (percentage shift) that has been determined previously is transferred to the next bucket. The process continues until the last bucket is filled completely. The amount of bwhists being generated is determined by the amount of buckets and the percentage shift according to equation 3.1 where b is the amount of buckets, s the percentage shift, and n the amount of bwhists:

$$n = \left\lfloor \frac{1}{s} \right\rfloor \cdot (b-1) + 1 \tag{3.1}$$

Figure 3.1 shows the first four bwhists being generated for sequences of 8-bit integers with a percentage shift of 0,33. This La-Ola configuration would generate 22 bwhists. Due to the good ratio between simplicity and performance, we use the La-Ola approach for our data generation as well. After the bwhist generation, an integer sequence of arbitrary size can be derived.



(a) bwhist 1

(b) bwhist 2

(c) bwhist 3

(d) bwhist 4

Figure 3.1: La-Ola generation example for a 8-bit integer sequence and a percentage shift of 0,33.

In order to label the generated data, the implementation of the COLLATE-Metamodel [11] was used as it supports multiple algorithms as well as different parameterizations. The current version of it includes *StaticBP* and *DynamicBP*. Therefore, only those two algorithms are considered, but the selection strategy for parameterizations can be extended for additional algorithms easily. The framework supports the input format, the output format, the maximum bitwidth, and whether the sequence is sorted or not as parameters. As all of them except the output format are data-dependent, not every possible combination of parameters is valid. Therefore, only valid combinations are added to the source data before they are getting labeled.

3.2 FEATURE ENGINEERING

Regarding ML, the process of feature engineering is considered as an important, but also most labor-consuming one [2], as the quality of the model highly depends on the feature vectors it was trained with [10].
After the generation of the bwhists and the integer sequences they are representing, it is necessary to derive features the ML model can be trained with. They can be divided in three classes. The first class contains features, that can be directly derived from the bwhist and have also been used for the selection strategy of Woltmann et. al [20]. The second class contains features that have to be derived from the generated integer sequence. Class three is formed by features representing algorithm parameters.

Class 1:

- minBucket: The lowest bucket of the bwhist that contains a percentage higher than 0.

- maxBucket: The highest bucket of the bwhist that contains a percentage higher than 0.
- minPercentage: The lowest percentage of the bwhist a bucket contains.
- maxPercentage: The highest percentage of the bwhist a bucket contains.
- averageBitwidth: The average bitwidth of the bwhist.

The average bitwidth is the weighted sum of the bitwidths the buckets are representing and its percentage.

Class 2:

- mean: The mean value of the integer sequence.
- stdev: The standard deviation of the integer sequence.
- skew: The skew of the integer sequence.
- kurtosis: The kurtosis of the integer sequence.
- isSorted: A boolean value indicating if the sequence is sorted or not.

The feature isSorted can be assigned to the following class as well, as it is also part of the algorithm parameterization.

Class 3:

- maxBitwidth: The maximum bitwidth of the integer sequence.
- inputFormat: The input format of every value in the integer sequence.
- outputFormat: The output format of every value in the integer sequence.

All classes combined lead to 13 features representing an integer sequence and algorithm parameterizations.

The COLLATE-Metamodel of Hildebrandt et. al [11] supports on the one hand the measurement of the compression runtime which is the sum of the times for compression and decompression, and the compression rate on the other hand. We used both of them as target values by training one model for the compression rate and one for the compression runtime in order to compare them.

3.3 HYPERPARAMETER TUNING

After the data generation and feature engineering process, it is necessary to train the model and tune the hyperparameters. Like Woltmann et. al [20] we decided to use GB regression to build our ML model. In comparison to Neural Networks, the training phase of a GB model is less time-consuming. This is an important aspect for our use case as we need a ML model for every combination of algorithm and target value. Due to the consideration of the algorithms *StaticBP* and *DynamicBP* as well as the target values compression runtime and compression rate, four models have to be trained. Additionally, GB regression has certain hyper parameters. The number of weak learners i.e decision trees and the maximum depth per decision tree are the parameters

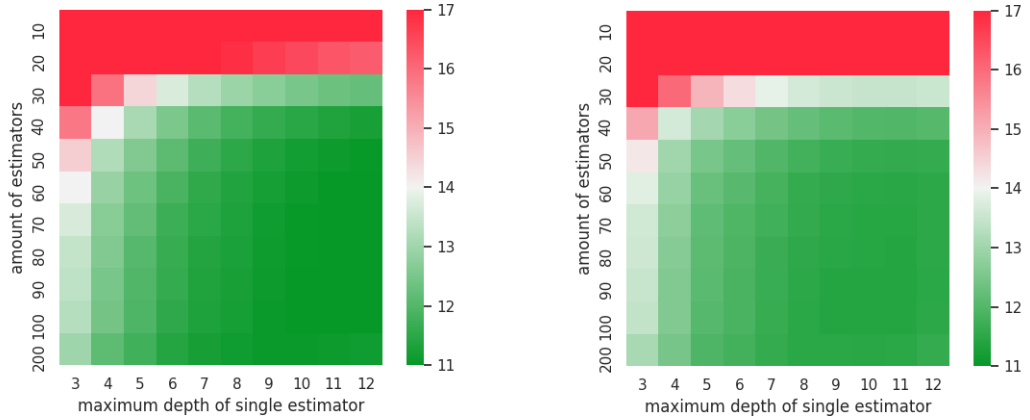
mainly influencing the model's quality [20]. Regarding the number of decision trees, we considered the search space $T = [10, 100] \cup [200]$ with a step size of 10 for the first interval. For the maximum depth per tree, the search space $D = [3, 12]$ was used.

Based on both sets, the size of the whole two-dimensional search space s is the product of the cardinalities of T and D as shown in equation 3.2.

$$s = |T| \cdot |D| = 110 \tag{3.2}$$

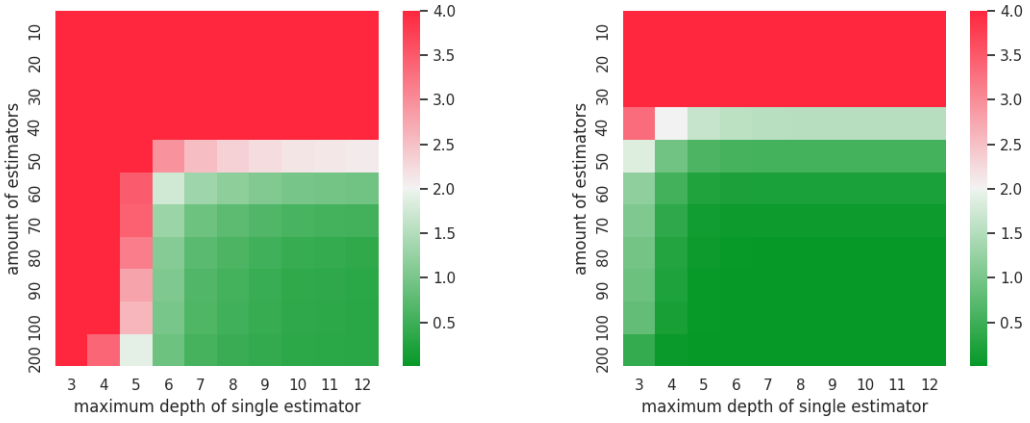
In order to compare the quality of the trained models, we decided to use the SMAPE metric which is the Symmetric Mean Absolute Percentage Error between a set P of predicted values and a set A of actual values. It is defined as follows:

$$SMAPE(A, P) = \frac{200\%}{n} \sum_{i=1}^{n} \frac{|P_i - A_i|}{|A_i| + |P_i|} \tag{3.3}$$



(a) *DynBP* **(b)** *StatBP*

Figure 3.2: Hyperparameter tuning results (SMAPE in %, compression runtime as target value).



(a) *DynBP* **(b)** *StatBP*

Figure 3.3: Hyperparameter tuning results (SMAPE in %, compression rate as target value.)

The hyperparameter tuning process has shown that for *DynamicBP*, 80 estimators and a maximum depth per tree of 10 for both, the compression rate and the compression runtime lead to reasonably good SMAPE values while not overfitting the model. For *StaticBP*, 80 estimators and a maximum depth per tree of 9 for the compression runtime and 80 estimators and a maximum depth of 6 for the compression rate lead to the best results.

3.4 CONCLUSION

Based on the presented methods of data generation, feature engineering, and hyperparameter tuning, a set of ML models can be trained with representative data which is the core element of the *Learned Selection Strategy For Lightweight Integer Compression Algorithm Parameterizations*. Having a sequence of integers, the bwhist of it can directly be derived. Given this representation, it is possible to predict the best-fitting algorithm and its parameters by passing every possible input combination to each algorithm model and selecting the one with the best compression runtime or compression rate.

As all of the concepts are general abstractions, a specific implementation is necessary for data generation, data labeling and model training on the one hand, as well as validation and forward passing on the other hand.

4 IMPLEMENTATION

The particular implementation of the general concepts for data generation, feature engineering and hyperparameter tuning explained in Chapter 3 can be divided into two parts. As each part consists of many single steps, we will explain our implementation with two pipeline models in this chapter. The first pipeline generates the dataset for each considered algorithm. The second one trains and tunes the ML models and validates them afterwards. Hence it depends on the output of the data generation pipeline. We split it up anyway due to the usage of two target values. The process of validation needs to be executed more often than the data generation process.

4.1 DATA GENERATION PIPELINE

Figure 4.1 shows the pipeline for the generation of representative training data. The process is necessary as the quality of the ML model directly depends on it. The La-Ola strategy of Woltmann et. al [20] showed the best results within their *Learned Selection Strategy for Integer Compression Algorithms*. Firstly, the La-Ola Generator runs four times being configured with a percentage shift of 0.01 and multiple bucket amounts representing the input formats 8 bit, 16 bit, 32 bit, and 64 bit one after another. According to equation 3.1, it produces 11.604 bwhists in total. Afterwards, the features of the bwhist (class 1) can directly be derived.

The next step is the generation of integer values the algorithms can compress. Integer sequences passed to the algorithms of the COLLATE implementation of Hildebrandt et. al [11] require a length which is integer divisable by the specified output format. Hence, an integer sequence with 800, 1600, 3200, and 6400 values is generated, representing the output formats 8, 16, 32, and 64 bit. This sampling process for combinations of input and output format increases the number of bwhists to 46.416. Furthermore, every sequence has to be considered sorted and unsorted in order to have an equal distribution of the isSorted feature. To avoid the generation of invalid data, the feature maxBitwidth is directly derived from the bwhist and not sampled. The whole process results in a list containing 92.832 samples. Completing the elements to feature vectors requires another feature extraction process deriving the features of the integer sequence (class 2) and the features representing the algorithm parameters (class 3). The feature vectors and its associated

integer sequences are now exported to a CSV-File.

The next step is the labeling of every feature vector with the behavior of each algorithm i.e. the compression rate and the compression runtime. Therefore, *StaticBP* and *DynamicBP* are applied three times to the integer sequence of every feature vector. During the compression, the compression rates and runtimes are measured and added to the CSV-File. In order to keep the memory footprint small, the integer array is deleted after the compression process.

The final step is the averaging of the target values of every compression process. The multiple execution and subsequent averaging ensures that the impact of outliers are flattened which results in a better ML model quality.

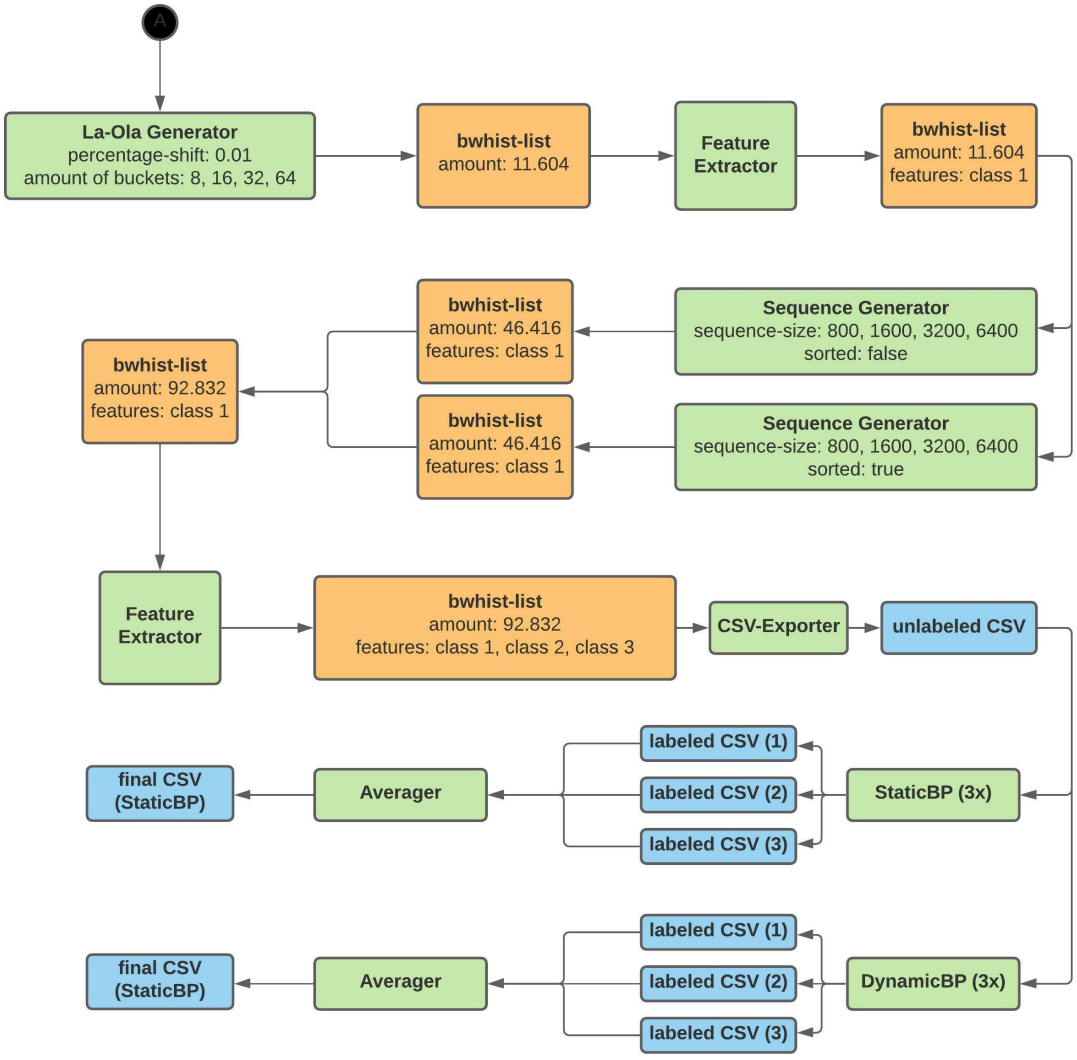


Figure 4.1: Pipeline for the generation of training data.

4.2 VALIDATION PIPELINE

After generating the final CSV-File containing the representative data, the ML model can be trained and tuned. The subsequent step is the validation of the ML models against the baseline selection strategy. Figure 4.3 shows the implementation pipeline for this process. It has to be executed once for each algorithm and for each target value. At first, a fixed subset of 20% of the entire data set is extracted. The *Algorithm Parameter Sampler* removes the features inputFormat and outputFormat as well as the target values and samples the remaining feature vector for every combination of parameters for the algorithm the data allows, for example the input format can not be smaller than the maximum bitwidth. Hence the sampling process leads to a minimum of 4 feature vectors per data sample if the maximum bitwidth is greater than 32, and a maximum of 16 feature vectors if the maximum bitwidth is smaller than 9. Figure 4.2 shows an example with 16 possible combinations.

mean	stdev	…	maxBitwidth	inputFormat	outputFormat
1	0	…	1	8	8
1	0	…	1	16	8
1	0	…	1	32	8
1	0	…	1	64	8
1	0	…	1	8	16
…					
1	0	…	1	64	16
1	0	…	1	8	32
…					
1	0	…	1	64	32
1	0	…	1	8	64
…					
1	0	…	1	64	64

Figure 4.2: Example for a feature vector sampled with 16 algorithm parameter combinations.

The next step is the prediction of the target value. Therefore, every feature vector of the sampled test data including the parameterization of the algorithm is passed to the ML model on the one hand and to the baseline on the other hand. This leads to two datasets per algorithm and target value. The first dataset contains the target value predicted by the ML model for each feature vector. The target value of the other one represents the behavior of the baseline strategy which is the selection of the most simple algorithm i.e *StaticBP* with an input format of 64 bit and an output format of 8 bit. This combination of input and output format is the most simple one as it covers all samples while having the best compression rate.

The CSV-File generated by the data generation pipeline contains a feature vector for every possible algorithm parameter combination. In order to compare the results of the ML model and the baseline, they are passed to the *Accuracy-Slowdown Processor* together with the actual data. The processor determines the minimum of the actual target values over each data sample for all algorithm parameter configuration and the minimum of both the ML model and the baseline. Now it is possible to compare, which algorithm and which parameterization would be chosen by our strategy, which one would be considered the best if the most simple algorithm is used, and which one is actually the best. Given this information, the calculation of the accuracy for the correct algorithm as well as for the correct parameterization is possible. Additionally, the slowdown can be determined, if a wrong parameterization has been chosen.

The same pipeline can be used for the validation against any data set that has been labeled with

the algorithm behavior i.e compression rate or compression runtime. For this, the final CSV-File is used in order to train the ML model. This makes sure that training and test data are disjoint. The *Algorithm Parameter Sampler* is then applied directly to the validation data. It is the source for the actual target values that are passed to the *Accuracy-Slowdown Processor*.

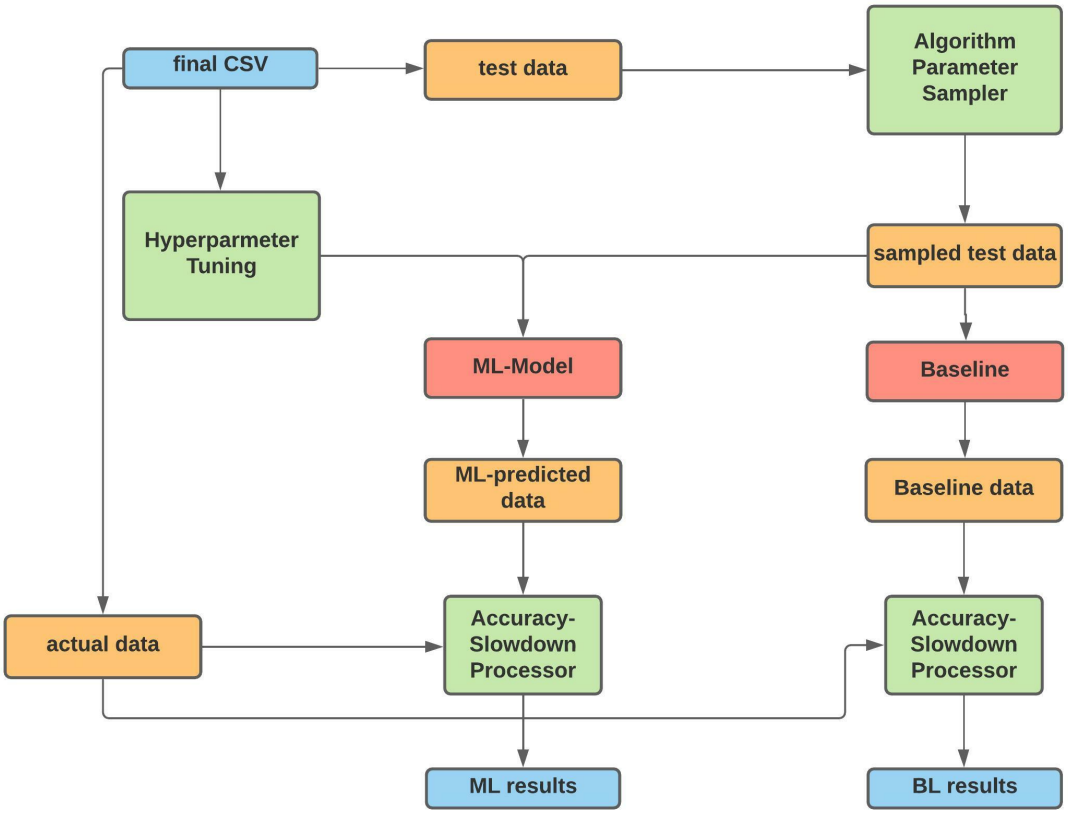


Figure 4.3: Pipeline for the validation process of the trained ML-Model.

5 EVALUATION

This chapter contains the evaluation of our *Learned Selection Strategy for Lightweight Integer Compression Algorithm Parameterizations* which is divided into three parts. Firstly, we describe our hardware setup and data sets and evaluate how long training and forward passes take. Secondly, we validate our approach against the baseline selection strategy and evaluate its performance with a validation data set. Lastly, we take a look at the feature importance in order to understand the predictions of our ML models better.

5.1 SETUP

All models have been trained on a AMD Quad-Core A10-9700 APU system with 16 GB memory. The same system has been used to label the feature vectors and for application and testing of the models. The process of data labeling was necessary once by running all examples with *StaticBP* and *DynamicBP*. The implementation of the COLLATE metamodel by Hildebrandt et. al [11] considers every different parameterization as a different algorithm and generates its implementation on the fly. This results in long compiling times before the data labeling process can start. Once labeled, the training and application of the ML model is possible.

Advantages of GB regression are the relatively fast times for training and forward passes in contrast to other methods like Neural Networks. On average, one training phase took 17.3s. Due to the fact that it was necessary to train 110 models during the hyperparameter tuning for each combination of algorithm and target value, the whole process took 127 minutes. For the forward passing process we measured an average duration of 390μs.

For the following validation measures, we used different data sets. One the one hand, we used a fixed 20% of the set generated by the La-Ola generator labeled with the compression runtime and the compression rate. This dataset is called generated data set (GDS). On the other hand, we evaluated how our strategy performs on real-world data. Therefore we used a further data set (pBI) which consists of representative samples from the Public BI benchmark. The Public BI benchmark is a user-generated benchmark and consists of real world data represented in different tables [9].

5.2 VALIDATION

The ML model of our selection strategy is able to predict the optimal compression algorithm as well as the best-fitting parameters. In order to validate the quality of the predictions, we use multiple indicators. The *algorithm accuracy* (aa) is the relative frequency of true positive algorithm predictions *aTP* for a data set *S*.

$$aa(aTP, S) = \frac{aTP}{|S|} \cdot 100\% \tag{5.1}$$

Regarding the *parameterization accuracy* (pa), we use the true positive predictions of the the algorithm parameters *pTP*.

$$pa(pTP, S) = \frac{pTP}{|S|} \cdot 100\% \tag{5.2}$$

Besides the calculation of the accuracy, we also consider the slowdown. It measures how much performance would be lost if the wrong prediction is chosen. As our strategy focuses on the parameterization, we decided to exclusively consider the slowdown of the wrong parameters. Having a set G containing the target values (runtimes or compression rates) of all wrongly predicted parameterizations, a set H containing their actual target values, and the amount of all wrongly predicted parameterizations n, the slowdown can be calculated using the SAMPE metric.

$$slowdown(G, H) = \frac{200\%}{n} \sum_{i=1}^{n} \frac{|G_i - H_i|}{|G_i| + |H_i|} \tag{5.3}$$

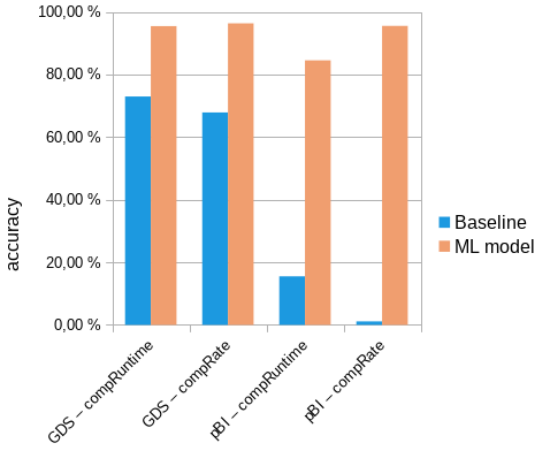
In order to evaluate the quality of our ML model, a baseline is necessary. We decided to compare our model to the most simple algorithm with a parameterization allowing the compression of integer values within a range as large as possible. In particular, this baseline strategy uses *StaticBP* with an input format of 64 bit and an output format of 8 bit.

The hyperparameter tuning process has shown that the hyperparameters for *DynamicBP* and *StaticBP* are similar. Regarding the compression rate, a more complex model is necessary for *DynamicBP* in order to make predictions with the same accuracy as for *StaticBP* due to the fact that the compression rate of *StaticBP* can be directly derived from the parameters.

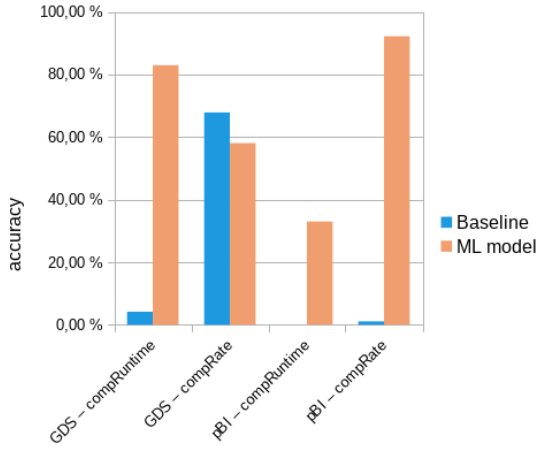
algorithm	target value	maxDepth of estimator	amount of estimators
StaticBP	compression runtime	9	80
	compression rate	6	80
DynamicBP	compression runtime	10	80
	compression rate	10	80

Figure 5.1: Result table of the hyperparameter tuning process.

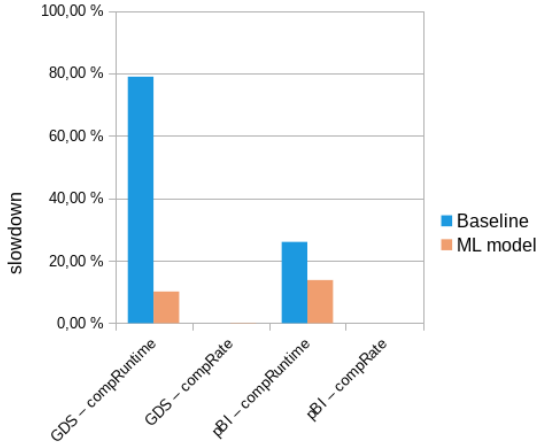
Our ML models have been trained on integer sequences with lengths of 800, 1600, 3200 and 6400 depending on the output format as described in Section 4.1. However, the input format of every integer sequence of the pBI dataset is 64 bit, but the lengths of the arrays are much bigger than the lenghts we chose for our training data. Though the sequence lengths of the pBI dataset differ for almost every sample, all of them are integer divisable by 8 what makes it possible to compress them using the COLLATE metamodel implementation by Hildebrandt et. al [11]. Due to the smaller lengths used for the training data, it is necessary to scale the predicted target value if it is proportional to the amount of integers being compressed. This is only the case for the compres-



(a) Accuracy of algorithm selection (aa).



(b) Accuracy of algorithm parameter selection (pa).



(c) Slowdown of incorrect parameter selection (slowdown).

Figure 5.2: Evaluation results against the generated data set (GDS) and Public BI data (pBI) considering the compression rate (compRate) and the compression runtime (compRuntime) as target values.

sion runtime. The compression rate is independent of the data length. Given the output format of, the unscaled compression runtime ur, and the length of the integer sequence l, we calculate the scaled runtimes (sr) for the pBI dataset according to the following equation.

$$sr = \frac{ur}{of \cdot 100} \cdot l \tag{5.4}$$

The validation results presented in figure 5.2 and its corresponding table 5.3 show that for the compression runtime as the target value, our ML model performs better than the baseline when using our test data as well as the Public BI data regarding the algorithm selection accuracy, the algorithm parameterization selection accuracy and the slowdown. Considering the compression rate, the baseline performs better for the accuracy of the algorithm parameter selection. The baseline accuracies of 67,85% for the test data and 1,1% for the Public BI data are equal to the values of the algorithm selection accuracy. The reason for this lies in the way *StaticBP* works. The combination of an input format of 64 bit and an output format of 8 bit always results in the best compression rate. Thus, if the baseline algorithm, i.e *StaticBP* is chosen and the compression rate is used as target value, the baseline parameterization is always the best. This is also the reason

for the corresponding slowdown of 0%.

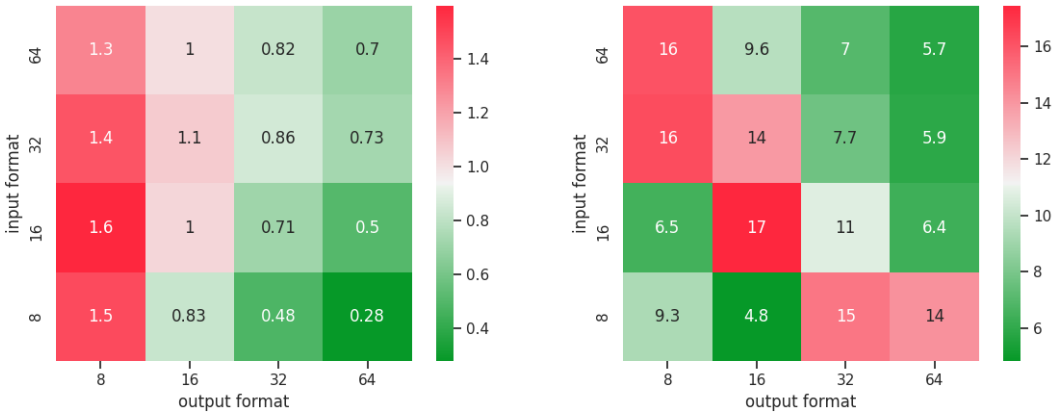
		GDS – compRuntime	GDS – compRate	pBI – compRuntime	pBI – compRate
accuracy algorithm selection	ML model	95,42 %	96,30 %	84,50 %	95,50 %
	Baseline	72,96 %	67,85 %	15,50 %	1,10 %
accuracy parameterization selection	ML model	82,95 %	58,08 %	33,01 %	92,20 %
	Baseline	4,20 %	67,85 %	0,00 %	1,10 %
slowdown incorrect parameterization selection	ML model	10,12 %	0,13 %	13,80 %	0,00 %
	Baseline	78,90 %	0,00 %	26,01 %	0,00 %

Figure 5.3: Validation table.

The GDS is naturally more systematic and contains less outliers than the pBI dataset. Regarding the accuracy of parameterization selection, we note that our ML model performs better on the GDS when trained for the compression runtime. In contrast, it performs better on the pBI dataset when trained for the compression rate. Due to the fact that the pBI data has different structures and properties, the model has to abstract more from the training data. Predicting the target values is therefore more difficult what results in the lower accuracies for the pBI data set than for the GDS. Since the compression rate leads to better results for the pBI data in comparison to the compression runtime, we conclude that it is easier for the ML model to predict the compression rate than the compression runtime. The measurement of the compression runtime is more prone to higher variations in the resulting data. The values for the compression rate on the other hand fluctuate less.

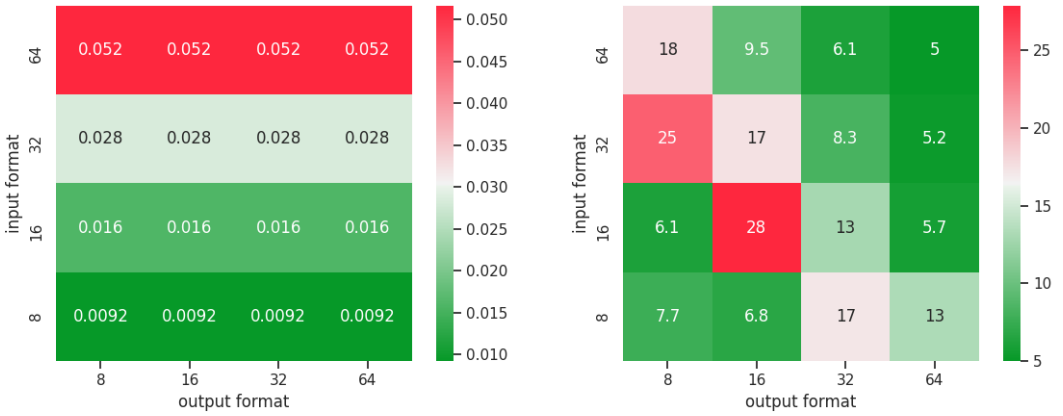
Furthermore, we observed that the ML model always reached better accuracies for the selection of the correct algorithm than for the algorithm's best parameters. The algorithm selection seems to be the easier task for the ML model than the prediction of the parameters. We expected this observation due to the different amount of possible results. While there are at worst 16 possibilities for the algorithm parameters (see Figure 4.2), there are only two for the correct algorithm.

Thus far, we evaluated our approach by considering the performance of our ML model as a whole, regardless of certain algorithm parameter combinations. In order to examine which parameter combinations lead to the best prediction results, we calculated the SMAPE for every possible combination of the input format and the output format. We do not consider the maximum bitwidth as it fully depends on the data. There are 16 different algorithm parameter combinations for the GDS. As all integer sequences of the pBI dataset have an input format of 64 bit, therefore only 4 combinations are possible.



(a) compression rate **(b)** compression runtime

Figure 5.4: Heatmap (SMAPE in %) for single parameter combinations on the GDS for *DynamicBP*.



(a) compression rate **(b)** compression runtime

Figure 5.5: Heatmap (SMAPE in %) for single parameter combinations on the GDS for *StaticBP*.

Figures 5.4 and 5.5 show the SMAPE values for each possible parameter combination within the GDS. Regarding the compression runtime, the lowest SMAPE values representing better prediction results have been calculated for combinations of small input and output formats on the one hand, and for combinations of large input and output formats on the other hand. The highest SMAPE values indicating worse prediction results were determined for combinations of small input formats with large output formats and vice versa. This behavior applies equally for both algorithms. The prediction quality of the ML model hence increases if either combinations of large input and output parameters or combinations of small input and output parameters are considered. If a new data set is passed to the ML model in order to predict the best fitting algorithm and its parameters with the compression runtime as target value, it is useful to adjust the possible range of output format values depending on the input format(s) of the data being analysed.
In contrast to that, the prediction results of the ML model using the compression rate as target values show a different behavior for both algorithms. While the ML model performs better for *DynamicBP* the larger the value for the output format, the output format has no influence on the prediction result of the ML model for *StaticBP*. The reason is that the compression rate of *StaticBP* is determined statically with the input format and the max bitwidth of the integer sequence.

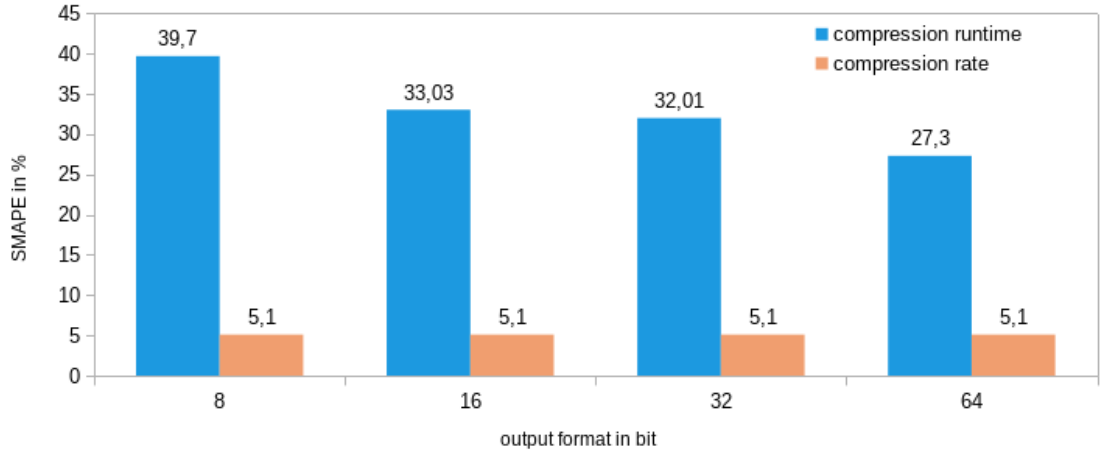


Figure 5.6: SMAPE barchart for possible output formats (pBI dataset, *StaticBP*, input format = 64 bit).

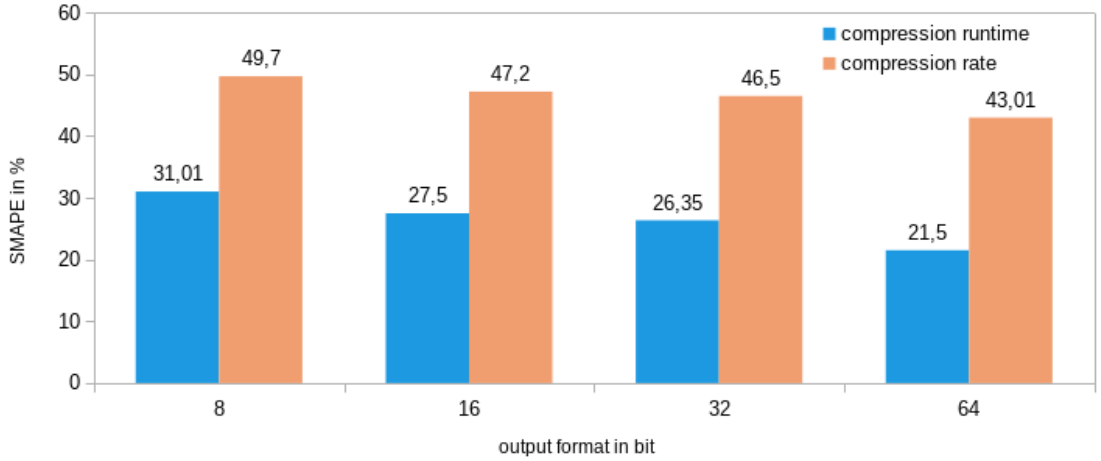


Figure 5.7: SMAPE barchart for possible output formats (pBI dataset, *DynamicBP*, input format = 64 bit).

Figure 5.6 and 5.7 show heatmaps containing the SMAPE values of each possible parameter combinations within the pBI data set. We can observe a similar behavior to the GDS. For every target value and algorithm, the SMAPE decreases the higher the output format is. The input format of the pBI dataset is 64 for every integer sequence. The increase in prediction quality with larger output format values fits the observation made for the GDS for an input format of 64 bit regardless the target value.

One exception is the compression rate when using *StaticBP*. Figure 5.6 shows again an equal SMAPE value for every possible output format. The reason is the same as already found for the GDS, as the functionality of the algorithms is the same for all data.

5.3 FEATURE IMPORTANCE

Being able to interpret the ML models of our *Learned Selection Strategy for Lightweight Integer Compression Algorithm Parameterizations* requires the consideration of feature importance. The importance of a certain feature indicates what influence it has on the model's quality. There are different ways to determine the importance of a feature. The indicator we used is the permutation feature importance [4].

To determine the individual importance, the following process is applied successively to every feature [4]. Firstly, the feature values get shuffled randomly over all samples which results in a decrease of the quality of the ML model. As the shuffling process breaks the relationship between the feature and the target value, the decrease is an indicator how much the target value depends on the feature. This process leads to a value indicating the decrease of the model quality and hence the importance of the feature (feature importance value).
We used the implementation of scikit-learn[1] which is outlined as follows: Given the ML model m and the training data set D, a reference score s is calculated indicating the quality of the model. Now, the values for each feature j get shuffled multiple times k which results in a corrupted version of the data set $\widetilde{D}_{j,k}$. Now, the reference score $s_{j,k}$ is calculated for the model m on the corrupted data set $\widetilde{D}_{j,k}$. The importance of each feature is the difference of the reference score belonging to the original data set and the average of the corrupted reference scores.

$$i_j = s - \frac{1}{K} \sum_{k=1}^{K} s_{k,j} \tag{5.5}$$

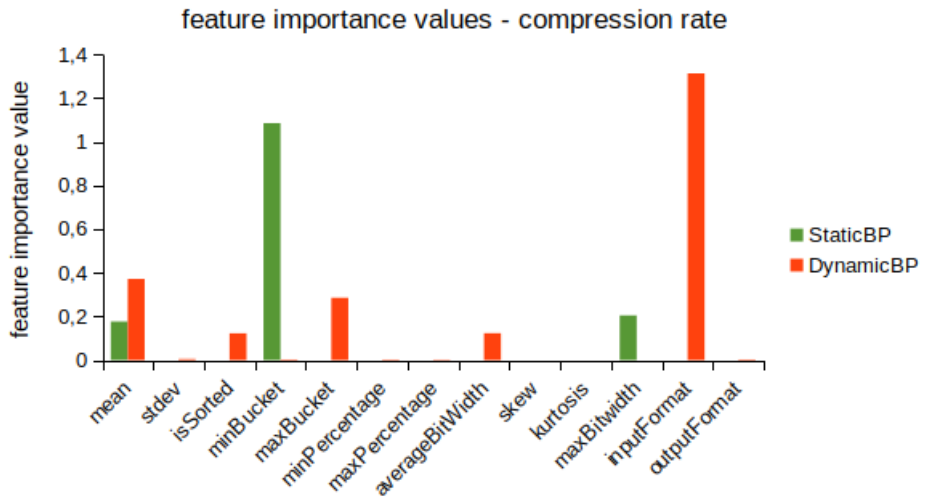


Figure 5.8: Feature importances for compression rate as target value.

Figure 5.8 shows that the ML model for *StaticBP* mostly depends on the minBucket feature and the maxBitwidth feature when trained to predict the compression rate. The model for *DynamicBP* mostly depends on the input format. Both observations are indicators that the ML models

[1] https://scikit-learn.org/stable/modules/permutation_importance.html, last access: 01.10.2021

represent the behavior of the algorithms. Considering *StaticBP*, the block size is fixed and determined by the maxBitwidth parameter. As the minBucket feature is inversely proportional to the maxBitwidth feature, the model can derive the algorithm behavior from both features. Regarding *DynamicBP*, the bit width of a block's largest value is chosen to represent all data elements in the block [20]. The input format correlates with the bit width of a block's largest value which is the reason for its importance.

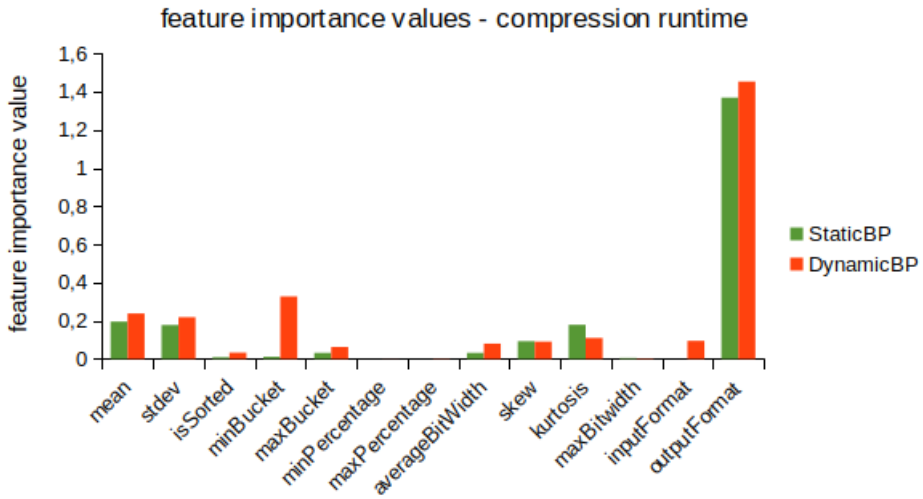


Figure 5.9: Feature importances for compression runtime as target value.

Figure 5.9 shows the feature importances on the ML models when trained on the compression runtime. For both *DynamicBP* and *StaticBP* the outputFormat is the feature with the highest feature importance value. The output format is an indicator for the length of the integer sequence being compressed. Hence we can conclude that the longer the sequences are, the more time it takes to compress them which is an intuitive behavior. This is also the reason why it was necessary to scale the predicted compression runtime value of the pBI dataset accordingly. Besides the output format, the features mean, stdev and minBucket have relatively high feature importance values. All three features are correlated to the size of the integer values the sequence contains. In addition to the length of the sequence, the size of the values also affects the compression runtime.

5.4 CONCLUSION

The evaluation of the *Learned Selection Strategy for Lightweight Integer Compression Algorithm Parameterizations* has shown that its usage results in better compression results (i.e compression runtime and compression rate) than the application of the simplest approach for generated data as well as for real world data. Additionally, our learned approach results in a lower slowdown if the incorrect parameterization is chosen than the most simple approach does.

6 SUMMARY AND OUTLOOK

Within the wide range of algorithms for lightweight integer compression, no single-best one exists [5, 6]. As they all behave differently depending on hardware and data properties, multiple selection strategies have been proposed [6, 20]. All of them have in common that they can determine the best-fitting algorithm but do not consider possible algorithm parameterizations. Hence, we evaluated a *Learned Selection Strategy for Lightweight Integer Compression Algorithm Parameterizations* which extends the strategy of Woltmann et. al [20]. Firstly, it was necessary to generate representative data for training and testing with the La-Ola strategy, because our strategy is a ML based approach. From the bwhists generated by the La-Ola generator we derived 13 features representing an integer sequence. Subsequently, we applied the compression algorithms *StaticBP* and *DynamicBP* and labeled the data set with the compression runtime and the compression rate representing the behavior of the algorithm. For this, we used the COLLATE implementation of Hildebrandt et. al [11]. With the labeled data sets, our ML models for every combination of algorithm and target value could be trained and their hyperparameters tuned. We used GB regression which requires relatively low times for training and forward passes. After the training phase, we evaluated the quality of our approach in comparison to a baseline strategy always choosing the most simple algorithm and the parameters that are covering the largest range of data. A test data set containing generated data and a real world data from the Public BI benchmark have been used. In order to increase the transparency of our ML models, we calculated the impact of each feature on the prediction result using the permutation feature importance approach.
We could show in our evaluation that a Learned Selection Strategy based on Machine Learning is an effective way to predict suitable parameters in addition to the best-fitting compression algorithm for certain input data. Regarding the selection results, our approach outperforms the baseline strategy in almost every case. Hence, the application of our ML based approach leads to faster or better compression results of integer values, depending on whether the compression runtime or the compression rate was used as the target value. With the analysis of the feature importance we could derive certain behaviors of our ML models. Knowing these behaviors makes it now possible to consider them during the application on new data.

We evaluated our strategy using two different integer compression algorithms. Due to the fact that our approach is an extension of the black box strategy of Woltmann et. al [20], no information about the algorithm behavior is necessary what makes it possible to add further algorithms with-

out a complete new training phase. Regarding our strategy, it is firstly necessary to apply the new algorithm to our generated data. This step leads to a new labeled data set which is used to train the new ML models. In contrast to modelling the problem as classification task, our regression approach only requires a training phase for every ML model that belongs to the new algorithm. The new model can subsequently be used to predict the behavior of the new algorithm and hence to extend the amount of possible selection results. Another aspect is the extension of the considered parameters. Our data generation pipeline generates all valid algorithm parameter combinations. To add a new one, it is necessary to generate a new data set based on the new possible combinations. A new data set would also lead to a new labeling process and hence to a new learning phase for every combination of algorithm and target value. A more efficient way to add new parameters would be an interesting aspect of future work. In our evaluation, we only considered the permutation feature importance. Besides, the usage of Shapley Additive Explanations (SHAP) is a common approach [16]. While the calculation of SHAP values is more time-consuming, they can further increase the transparency of an ML model as they reveal if a feature negatively or positively influences the prediction result. Analyzing the feature importances with SHAP could also be part of future research.

BIBLIOGRAPHY

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 671–682, 2006.

[2] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

[3] Martin Boissier and Max Jendruk. Workload-driven and robust selection of compression schemes for column stores. In *Advances in Database Technology - EDBT*, volume 2019-March, pages 674–677, 2019.

[4] Leo Breiman. Random Forests. *Machine Learning*, 45:5–32, 2001.

[5] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. Lightweight data compression algorithms: An experimental survey experiments and analyses. In *Advances in Database Technology - EDBT*, volume 2017-March, pages 72–83, 2017.

[6] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. From a Comprehensive Experimental Survey to a Cost-based selection strategy for lightweight integer compression algorithms. *ACM Transactions on Database Systems*, 44(3), jun 2019.

[7] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. Morphstore: Analytical query engine with a holistic compression-enabled processing model. *Proceedings of the VLDB Endowment*, 13(11):2396–2410, apr 2020.

[8] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. Compressed linear algebra for largescale machine learning. *Proceedings of the VLDB Endowment*, 9(12):960–971, 2016.

[9] Bogdan Ghit, Cwi Amsterdam, NL Diego Tomé, and Peter Boncz. White-box Compression: Learning and Exploiting Compact Table Representations. *Cidr*, 2020.

[10] Jeff Heaton. An empirical analysis of feature engineering for predictive modeling. In *Conference Proceedings - IEEE SOUTHEASTCON*, volume 2016-July. Institute of Electrical and Electronics Engineers Inc., jul 2016.

[11] Juliana Hildebrandt, Dirk Habich, Thomas Kuhn, Patrick Damme, and Wolfgang Lehner. Metamodeling lightweight data compression algorithms and its application scenarios. *CEUR Workshop Proceedings*, 1979:128–141, 2017.

[12] Nusrat Jahan Lisa, Tuan Duy Anh Nguyen, DIrk Habich, Akash Kumar, and Wolfgang Lehner. High-Throughput BitPacking Compression. In *Proceedings - Euromicro Conference on Digital System Design, DSD 2019*, pages 643–646. Institute of Electrical and Electronics Engineers Inc., aug 2019.

[13] Yingting Jin, Yuzhuo Fu, Ting Liu, and Lan Dong. Adaptive compression algorithm selection using LSTM network in column-oriented database. In *Proceedings of 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference, ITNEC 2019*, pages 652–656. Institute of Electrical and Electronics Engineers Inc., mar 2019.

[14] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR 2019 - 9th Biennial Conference on Innovative Data Systems Research*, 2019.

[15] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. The Case for a Learned Sorting Algorithm. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1001–1016, jun 2020.

[16] Scott M. Lundberg and Su In Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems*, volume 2017-Decem, pages 4766–4775, 2017.

[17] Mahmoud Mohsen, Norman May, Christian Färber, and David Broneske. FPGA-Accelerated compression of integer vectors. In *Proceedings of the 16th International Workshop on Data Management on New Hardware, DaMoN 2020*, New York, NY, USA, 2020. ACM.

[18] David Reinsel, John Gantz, and John Rydning. Data Age 2025: The Digitization of the World From Edge to Core. 2018.

[19] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'neil, Pat O'neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 553–564, Trondheim, Norway, 2005. ACM.

[20] Lucas Woltmann, Claudio Hartmann, Dirk Habich, Wolfgang Lehner, and Patrick Damme. Learned Selection Strategy for Lightweight Integer Compression Algorithms. *preprint*, 2021.