SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Exploring Performance of B-Tree Configurations Using Machine Learning Techniques

Roland Haidari

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Exploring Performance of B-Tree Configurations Using Machine Learning Techniques

Untersuchung der Performance von B-Baum-Konfigurationen durch Techniken des maschinellen Lernens

Author:	Roland Haidari
Supervisor:	Prof. Dr. Viktor Leis
Advisor:	M.Sc. Marcus Müller
Submission Date:	15.03.2024

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Garching, 15.03.2024 Roland Haidari

Abstract

This thesis aims to identify potential improvements for B+ tree implementations using Explainable Artificial Intelligence (XAI) on Machine Learning (ML) models. It starts by generating a dataset using the Yahoo! Cloud Serving Benchmark (YCSB) benchmark, preprocessing the results, training ML models, and finally analyzing the results of XAI algorithms. Through this process, the thesis delves into various stages of the data science life cycle. Additionally, pattern recognition algorithms, Apriori and association rules, are described and used, which further verify some insights found by the XAI approach.

Building on the work of Müller et al. [1] the findings of this thesis contribute to our understanding of B+ trees by highlighting the impact of data skewness, size of values, number of records and the page size of leaf nodes on the performance. Beyond the domain of B+ trees, this thesis, concludes that the currently available XAI algorithms can be utilized to enhance the interpretability of outputs originating from ML models.

Contents

1 Introduction

In computer systems, good responsiveness of systems is desired to achieve fast retrieval of information. Where and how data is stored, can greatly impact the latency of data storage systems. Traditionally, systems would offload data to disk, which would lead to slower performance. More modern hardware technologies allow storing bigger amounts of data in-memory, which brings lower latency. To optimize storage, there exist numerous data structures and ways to store the data. One important factor, however, is the need to be able to efficiently offload data to secondary storage, once the memory limits are reached.

The B+ tree structure enabled managing data in-memory and supports disk storage as well. In their paper, Müller et al. [1] aim to optimize the performance of B+ trees by exploring various possible configurations and optimizations on different datasets and statistically analyzing the performance impacts of the configurations.

Nowadays, according to Davenport [2], ML and Artificial Intelligence has taken more relevance in the data science world. The idea of gaining insights with smart algorithms and large amounts of data is naturally intriguing. This thesis attempts to utilize machine learning to find potential optimization techniques of B+ trees. To do so, multiple steps need to be taken. These steps, as explained by Stodden in [3], include data collection, data exploration, data preprocessing, model training and model evaluation. Together these steps form the data science life cycle.

Once a well-performing model is trained, it can be attempted to analyze and explain the insights that the ML model has learned. XAI is a field in the data science that focuses on this idea. This thesis, tries to apply common XAI techniques to the trained models to verify existing insights or gain new - possibly unseen - information that could help optimize B+ tree performance.

In addition to the classical model training and XAI, a data-mining pattern recognition approach is taken, where two algorithms are used to gain insights by computing frequent itemsets and association rules.

The code base, excluding the datasets, is published in the GitHub repository at `https://github.com/rolandhaidari/thesis`.

2 B+ Tree Implementations

2.1 B+ Tree

2.1.1 Structure

The original paper on B-Trees [4] describes a data structure, where data is managed in a tree-like structure, with a node containing data, and having children with one node being on top, called the root. According to Graefe [5] it has self-balanced mechanisms that ensure that the data is remained sorted and lookups require an efficient amount of comparisons by the utilizing binary sort techniques. In Figure 2.1 the root node contains three values and references the children in a sorted manner.

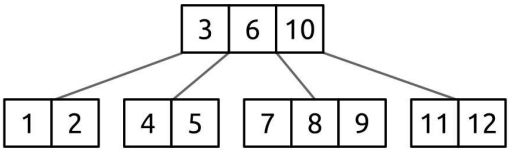


Figure 2.1: Simple B-Tree example [6]

Graefe [5] further discusses that B+ tree is a variant of the B-tree. Instead of storing the values inside the all nodes it stores the values in the lowest layer, called the leaf nodes. The inner nodes keep track of the keys and reference child nodes, in a sorted manner. To be able to traverse key ranges fast, the leaf nodes can contain a reference to the neighboring leaf node. The implementation by Müller et al. [1] does not implement these references. The figure 2.2 outlines a simple example of this variation. The data is structured by keys and values. For example, the number 3 represents the key 3 and the middle leaf node contains the reference to the corresponding value d_3. In this figure, the references to neighboring leaf nodes are visualized in dark red.

2.1.2 Operations

The B+ tree, similarly to the B-tree, has, among others, the following operations: Insertion, Search, Deletion, and Range Queries. The differences between the B+ tree and the

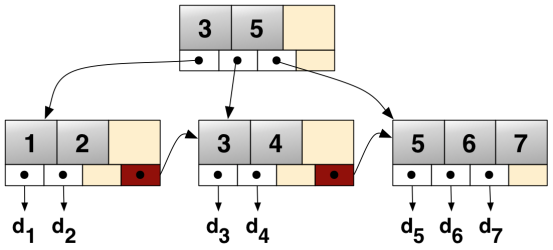


Figure 2.2: Simple B+ tree example [7]

B-tree lie in the nature of the structures, and therefore the operations have differences in memory usage and performance. Some operations are more suited in a B+ tree than a B-tree. The table 2.1 outlines the differences in regard to the operations. Overall, a notable downside of B+ trees is the generally bigger storage use, since a B-tree is able to store values in the inner nodes. Graefe [5] reported that this downside, however, simplifies some operations, bringing an overall, better performance, making the B+ tree the default structure in modern times. This thesis focuses on the B+ tree and the operations of insertion and range querying.

In this thesis, we consider the operations described in by Müller et al. [1], namely **insert, remove, lookup, scan** and more specifically the **insert**, **lookup** and **scan** operations.

2.2 Optimizations

The paper by Müller et al. [1] uses an implementation of the B+ tree that is implemented in the programming language C++. The paper outlines certain optimizations and configurations that can be utilized with the aim to improve operation runtime performance. The following optimizations have been proposed:

- **Prefix Truncation** refers to truncating the prefixes that are shared among keys stores in a node. An example for such truncation is cutting off the prefix "*https://*" in url data. This brings benefits to storage use and the cache is better utilized. The key transfer between nodes requires an extra operation, which adds complexity, however as this operation is rare, the runtime cost is minimal.

- **Heads** is the idea of taking the first 4 bytes of the key, named *head*, and using those for the binary search. This enables better use of the cache. The figure 2.3 represents this idea by using the first letter of the keys as heads. On the other hand, the slots with heads would require more bytes to be aligned. The implementation by Müller et al. [1] decided on doing unaligned access, since the negative impact is believed to be negligible.

Operation	Description	B-tree	B+ tree
Search (**lookup**)	Finding a given key in the tree.	Binary search through the tree	Same as B-tree.
Insertion (**insert**)	Adding a key-value pair	Values are also stored in internal nodes.	Keys are stored in internal nodes, Values only in leaf nodes.
Deletion (**remove**)	Removing a key-value pair	Since values are stored in the internal nodes, merging and redistribution operations are more complex.	Simplified merging and redistribution.
Range Queries (**scan**)	Retrieving a range of keys or values	Less suitable due to values stored in internal nodes.	More suitable because of leaf nodes containing pointers to their succeeding leaf node, which allows sequential access.
Traversal	Iterating through all keys or values in sorted order.	Follows a dept-first approach.	Reduced complexity, since values are only in linked leaf nodes, only requiring one big range query.
Splitting Nodes	Full nodes are divided into two nodes.	May redistribute keys and values.	Only internal nodes (keys) are affected. Values are not redistributed.
Merging Nodes	During deletion, nodes may become too empty. Nodes need to be merged as a consequence.	Keys and values need to be redistributed	Simplified as only keys are affected.

Table 2.1: Comparison of Operations in B-trees and B+trees

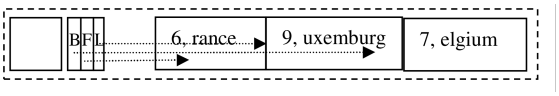


Figure 2.3: Heads example [5]

- **Hints** takes the idea of **heads** one step further, by adding a structure called hint array to each node header. The array contains the heads of equidistant slots, as visualized in figure 2.4. The implementation by Müller et al. [1] sets the hint array to the size of 16 hints, and the array is updated only at or after the insertion.

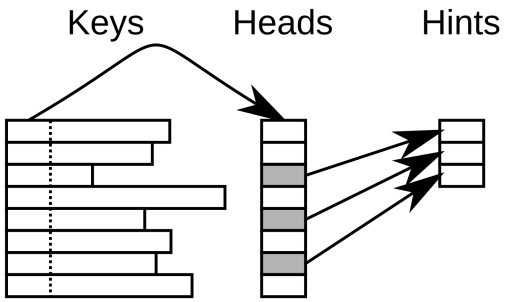


Figure 2.4: Hints from heads [1]

- **Fingerprinting** introduces an array of key hashes in the heap of the node. The figure 2.5 is an example for such an array. The array contains a sorted and an unsorted part. During insertion, the values are appended to the end - the unsorted part. The fingerprint array only gets sorting during scan, node splitting, merging operations. If used in combination with **prefix truncation**, the prefixes get truncated before calculating the hash. Additionally, to improve performance of search operations on nodes the SIMD instructions with a range of 256 bit are used.

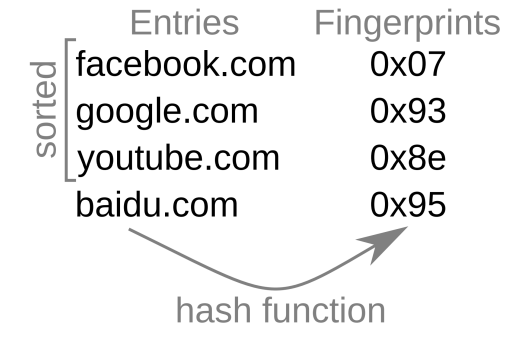


Figure 2.5: Fingerprinting array [1]

- **Dense Leaves** is a new approach introduced by Müller at al. [1], based on the idea of using frame-of-reference encoding to keys. Frame-of-reference encoding takes

reference keys and calculates the offsets for the following values. This idea is meant for keys that are dense enough, thereby mitigating the inefficiency associated with underutilized storage space. Two approaches are presented, namely **Semi Dense Leaves** and **Fully Dense Leaves**. As presented in figure 2.6, Semi Dense Leaves contain pointers at each possible offset to the node's heap containing the keys and values. The Fully Dense Leaves implementation presumes that the values are of the same size, which is a common characteristic for specific types of data. Additionally, a partition mechanism is implemented, which recognizes dense keys originating from different sources and therefore ensuring that the tree stores data more densely.

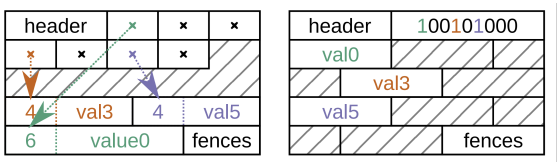


Figure 2.6: Example of Semi (left) and Fully (right) Dense Leaves with three values [1]

2.3 Adaptable B+ tree

Müller et al. [1] further discusses the topic of creating a B+ tree, which is adaptable, meaning that it can adjust the type of leafs to circumstances and the type operations. One approach is proposed, and shows considerable performance improvements in its benchmarks. The figure 2.7 describes, in which circumstances, leafs are transitioned into a different type.

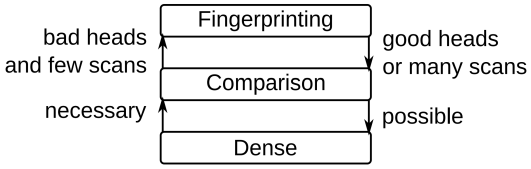


Figure 2.7: Decisions on leaf layout [1]

This thesis takes this idea and aims to get more insights into the possible approaches of configuring B+ trees. The goal is to improve on the idea of adaptable B+ trees, by finding factors impacting the performance in various configuration combinations. Particularly, the focus lies on the page size of leaf nodes, since this configuration is easily adaptable during runtime.

3 Preprocessing

This chapter describes, how the data has been collected and what steps have been done to transform the data, so that it is suitable for training models.

3.1 Data collection

3.1.1 Environment

During creating benchmarking data, it is necessary to create the same conditions, at each run of the program. Preferably, the data generation occurs on dedicated machines, where little to no other background processes can impact the performance of the benchmarking program. This approach has been taken in this thesis, where the data generation, occurs only at times, where no other user is using the same machine concurrently. This was ensured, by registering who uses the machine at a table on a shared tabular scheduling sheet. The table 3.1 describes the hardware and software characteristics of the server machine running the data generation program.

Characteristic	**Value**
CPU Product	AMD Ryzen 9 7950X 16-Core Processor
CPU Vendor	Advanced Micro Devices [AMD]
CPU Cores	16
CPU MHz	3360.195
Total Memory	64 GB
OS Description	Ubuntu 23.10
Architecture	x86_64
Linux Kernel version	6.5.0-15-generic

Table 3.1: Benchmarking Server System Information

3.1.2 Binary programs

After making sure that the data can be collected consistently, the subsequent phase involves thinking about how the data is to be collected. For the purpose of this thesis, program binaries have been provided, which execute the B+ tree configuration behavior as defined by Müller et al. [1]. Each binary file executes a b-tree configuration using the inputs and produce output a multi-line string in the comma-separated format containing

input configuration together with the measured outputs. More details on the operations being benchmarked are provided in section 3.5. The input and the output values are further outlined in feature tables, table 3.2 and table 3.3. In the feature tables, for numerical and boolean features, if only one value is provided as an example, this feature is not varied in further context.

Each run of a single binary gives an output in three lines. The first line describes the header of the comma-separated values. The second and third contain the inputs and output in the comma-separated format of the initialization of the operation and the actual operation, respectively.

When calling a binary program, some of the input parameters are embedded in the binary itself, in which case changing it requires a call to a different binary. This is the case for the inner and leaf node page size and the configuration (eg. hints, head). The rest of the parameters are passed to the binary program by setting environment variables beforehand.

Feature	**Description**	**Example Value(s)**
bin_name	Name of the binary	`bins/-DPS_I=4096 -DPS_L=4096/hints-n3-ycsb`
config_name	set of optimizations	hash, hints, baseline, prefix, dense3
const_basicHintCount	Basic hint count	16 , 0
const_enableBasicHead	Enable basic head optimization	1
const_enableDense	Enable dense optimization	0 , 1
const_enableDense2	Enable dense 2 optimization	0
const_enableDensifySplit	Enable densify split	0 , 1
const_enableHash	Enable hash optimization	0 , 1
const_enableHashAdapt	Enable adapation optimization	0
const_enableHeadNode	Enable head node optimization	0
const_enablePrefix	Enable prefix truncation optimization	1
const_hashSimdWidth	SIMD operation width	32
const_hashSortUseStdMerge	Hash sort use standard merge	1
const_hashUseCrc32	Hash use CRC32	0
const_hashUseSimd	Hash use SIMD	1
const_headNode4HintCount	Head node 4 hint count	16
const_headNode8HintCount	Head node 8 hint count	16
const_pageSizeInner	Inner node page size	2048, 4096, 8192
const_pageSizeLeaf	Leaf node page size	2048, 4096, 8192
data_name	Name of the data	data/urls, int
data_size	Size of the data, key count	5181932, 2813666
data_sorted	Data sorted status	0
density	Density	0.5, 0.75, 1
op	Operation	ycsb_c, ycsb_c_init, ycsb_e, ycsb_e_init
payload_size	Size of values	8, 256
rand_seed	Random seed	1703378455, 1703378098
run_id	Run ID	fc16dbaa-104b-4777-8087-bbe953b81b16
ycsb_range_len	YCSB range length	100
ycsb_zipf	YCSB zipf	0.9, 0.651367

Table 3.2: Input Features with Example Values

Feature	Description	Example Value(s)
time	Time in seconds	1.0512e-06, 3.126e-07
nodeCount_X	Node count of type X	19755, 20879
counted_final_key_count	Final key count	5798920, 3175985
cycle	Average number of CPU cycles pro operation	2197.381, 1087.615
instr	Average CPU instructions pro operation	1927.571 1151.517
X_miss	Average misses pro operation in cache type X	43.867, 9.918
task	Kernel execution time	392.291, 194.439
scale	Scale	5798920, 10000000
IPC	Instructions Per Cycle	0.877, 1.059
CPU	CPU usage	1, 0.993, 0.99
GHz	CPU frequency	5.601, 5.594

Table 3.3: Output Features with Example Values

3.1.3 Data generation program

Collecting the data is done through a Python script that runs in a loop, that terminates after a configured timeout. Inside the loop, the input parameters are generated either with a constant, or depending on the use case randomly from a given set, or a random value in a range. The outputs are collected and appended to three different files: the error logging file, the output file and the logging of the script file. Each file serves a purpose. The error logging file is useful to identify issues with the binaries that are called. The output file contains the data that is later analyzed. The logging of the script file is useful to identify issues of the Python script itself.

Two versions of this script exist that set the generation of the input parameters for the specific use case. For general model building more values are generated at random, whereas for the analysis of the page size, the values are more fixed to be able to be able to better make use of discrete data mining algorithms.

The Python script is called in a Poetry [8] environment and to make the script run in the background the *nohup* [9] command is utilized.

Once the script is finished running, the output files are checked to verify a successful run. Since the following steps do not require a consistent environment, the output files are copied to a local device, where the output is further processed, analyzed and other experiments are run.

ML model program

For the use case of training the ML models, the input parameters are set to the following values:

- **ycsb_zipf**: Random value between 0 and 1.5

- **op**: The operations *c* with *c_init* are chosen twice as much as *e* with *e_init*
- **data_name**: Set statically to *data/urls*
- **const_pageSizeLeaf**: Randomly varied in powers of two between 2048 and 16384
- **const_pageSizeInner**: Randomly varied in powers of two between 2048 and 16384
- **payload_size**: Set statically to 8
- **density**: Generated with `1 / 2 ** random()`
- **config_name**: Randomly chosen between *dense3*, *hints* and *hash*
- **data_size**: Generated by `floor((10 ** (random() * 0.5 + 8.25))/ 70.280)`. `70.280` corresponds to an estimated average size of a key.

Pattern recognition program

For the use case of using data mining techniques, and reasons stated in section 4.4, the input parameters are varied the following way:

- **ycsb_zipf**: Set statically to 1
- **op**: The operations *c* with *c_init* are chosen twice as much as *e* with *e_init*
- **data_name**: Set statically to *data/urls*
- **const_pageSizeLeaf**: Randomly varied in powers of two between 2048 and 8192
- **const_pageSizeInner**: Randomly varied in powers of two between 2048 and 8192
- **payload_size**: Randomly varied in powers of two between 4 and 256
- **density**: Set statically to 1
- **config_name**: statically set to *hints*
- **data_size**: Randomly generated in 1 million steps between 1 and 6 million.

3.2 Aggregate data

After the files are collected in a subfolder, all following steps are done in Jupyter Notebooks [10], to enable quick step-by-step running of the programs. There are two steps that are done:

In the first step, the multiple output files, originating from multiple data generation script runs, are merged into one file. This step is done by traversing the subfolders and finding files matching the naming convention defined for the specific types of the

generation script. The files are then merged by appending the lines of each file to an array and then writing the array into a new file.

The second step, takes the new merged file and removes the excessive header lines that are caused by each run of a binary creating its own header line. Additionally, in this step excessive spaces are removed, to make the output format in accordance with a comma-separated values (.csv) file type. This way libraries like *pandas* [11] and consequently *scikit-learn* [12], and *mlxtend* [13] can read and process the given data.

3.3 Cleanup data

After aggregating the data into one file, the data needs to be cleaned up, to be more suitable for ML models. For the use case of this thesis, multiple aspects need to be considered to make sure that the ML models, do not learn from data, that is invalid, wrongly interpreted or unavailable for the considerations of the use-case. These considerations are implemented using the *pandas* library [11]. After reading the data into a DataFrame, the following steps are done:

- **Remove failed runs**: Binary program executions that have run into an error during benchmarking occasionally produced outputs, that are to be considered outliers and irrelevant for model training. These executions are easily identified, because of the feature *time* being equal to 0. Therefore, these records are filtered out of the DataFrame.

- **Rescale time into time per op**: One important aspect to consider is the feature *time*. This feature is the total time that the program took to perform the benchmark. This, however, is highly correlated to the number of records being inserted or read. The total time, is not representative of the time, which this thesis aims to optimize. Instead, this needs to be transformed into the time per operation. This is done by dividing the feature *time* by the feature *scale*. For simplicity, the result overwrites the values for the feature *time*.

- **Reduce complexity of input**: When training some ML models, it is best practice to reduce the number of dimensions to reduce the number of records needed to get a performant model. There are ways to do this is with algorithms that reduce the dimensionality by projecting the higher dimensionality space into lower dimensions, trying to preserve the main information. One example, according to Bishop [14], is PCA, but another, simpler way is to perform feature selection, as proposed by Murphy [15]. This refers to dropping features that based on domain knowledge are known to not contribute to the end result. In this case, for the input features, it is only the feature *bin_name*.

- **Drop output features**: Another necessary step in ML is to ensure that the models are not getting information, that in the use-case, it would not have without knowing the result. In the case of this thesis, it is important to not feed models information, that is generated during the output of the binary program executions. This means that all output features, except the target feature *time*, are dropped out of the DataFrame. The importance of this step is further outlined in section 4.1.1 on correlation.

3.4 Encoding categorical data

Since some features are not numeric and are categorical, in order to be able to train ML models on the preprocessed data, they need to be converted into numerical values. The approach chosen in this thesis is **one-hot encoding**, as described by Murphy [15]. As a consequence, the amount of features is increased, however since only a few features are affected, and the features themselves, only contain a few categories, the number of features increased from 28 to 33.

3.5 Separate YCSB operations

The binary programs execute YCSB benchmarking programs as introduced by Cooper et al. [16]. YCSB can be configured for different workloads, which the figure 3.1 describes together with an example, where such a distribution of operations can occur. In this thesis, the feature *op* describes which workload is being measured. The main focus for this thesis is the workload C and secondarily the workload E is considered as well. This corresponds to the values *ycsb_c* and *ycsb_e* in feature *op*. The initial population of the B+ tree is also benchmarked and represented by values *ycsb_c_init* and *ycsb_e_init* respectively in feature *op*.

Workload	Operations	Record selection	Application example
A—Update heavy	Read: 50% Update: 50%	Zipfian	Session store recording recent actions in a user session
B—Read heavy	Read: 95% Update: 5%	Zipfian	Photo tagging; add a tag is an update, but most operations are to read tags
C—Read only	Read: 100%	Zipfian	User profile cache, where profiles are constructed elsewhere (e.g., Hadoop)
D—Read latest	Read: 95% Insert: 5%	Latest	User status updates; people want to read the latest statuses
E—Short ranges	Scan: 95% Insert: 5%	Zipfian/Uniform*	Threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id)

*Workload E uses the Zipfian distribution to choose the first key in the range, and the Uniform distribution to choose the number of records to scan.

Figure 3.1: Definition of types of YCSB workloads [16]

One early observation during the initial analysis was that the operation being benchmarked has a big influence on the measured *time*. The idea of the thesis, however, is to find interpretable insights into which configurations influence the *time* and how. Since the *op* has a significant impact, and different configurations might benefit some

operations more than others, the dataset is split into parts according to the feature *op* and all the models are trained based on the respective parts. The box plot in figure 3.2 represents the measured time of records for each operation type. The visible shift for each operation type, further supports the reason for splitting the data set, by highlighting the impact of single feature *op* on the feature *time*.

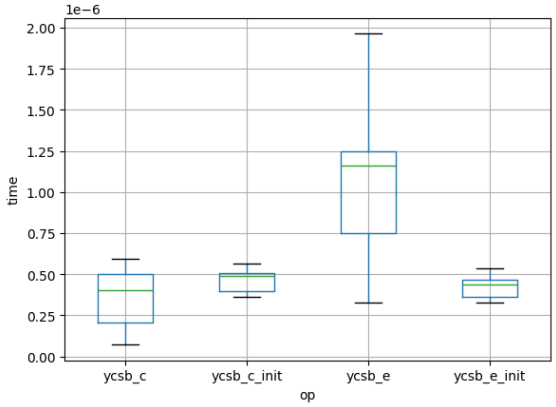


Figure 3.2: Box plot of time grouped by feature *op*

4 Approaches

In this chapter, the various approaches taken on getting insights using classical statistical, ML and Data Mining techniques are explained. The actual outcomes of the ML models and Data Mining algorithms are outlined in chapter 6.

4.1 Data exploration

Before training ML models it is important to understand what type of underlying data one is working with. The section 3 on preprocessing explains the individual features and explains why the split of the data based on feature *op* has been done. These findings originate from exploring the data in the first place, and understand which preprocessing steps are to be done to gain insights that are meaningful. One intuitive way to understand the relations between the features is by exploring the correlation between them.

4.1.1 Correlation

A correlation, as the name suggests, describes the relationship between two features. In this thesis, the Pearson product-moment correlation coefficient is covered as defined by Salkind [17]. This correlation coefficient measures the relationship between two values, giving a value in the range $[-1, 1]$. Positive values for features X and Y indicate a relationship, where if the value of a feature X increases, the feature Y increases too. Negative values indicate the that the decrease of feature X results in an increase of feature Y.
The coefficient r between two features X and Y is calculated using the formula 4.1.

$$r = \frac{n \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{n \sum_{i=1}^{n} (x_i - \bar{x})^2 \cdot n \sum_{i=1}^{n} (y_i - \bar{y})^2}} \tag{4.1}$$

where:

r is the Pearson correlation coefficient,
n is the number of data points,
x_i and y_i are individual data points of X and Y,
$\bar{x}$ is the mean of X,
$\bar{y}$ is the mean of Y.

This correlation can be calculated comparing all features, resulting in the correlation matrix as described by Papoulis et al. [18]. It is symmetric along the diagonal, as the Person correlation is symmetric in nature as well. The figure 4.1 describes the correlation in the dataset collected in this thesis. Non-numeric features have been excluded for better viewability. It can be observed that the lower right section of the matrix displays high absolute values. This can be easily explained, since these features are output features, which are highly correlated between each other, including the target feature *time*.

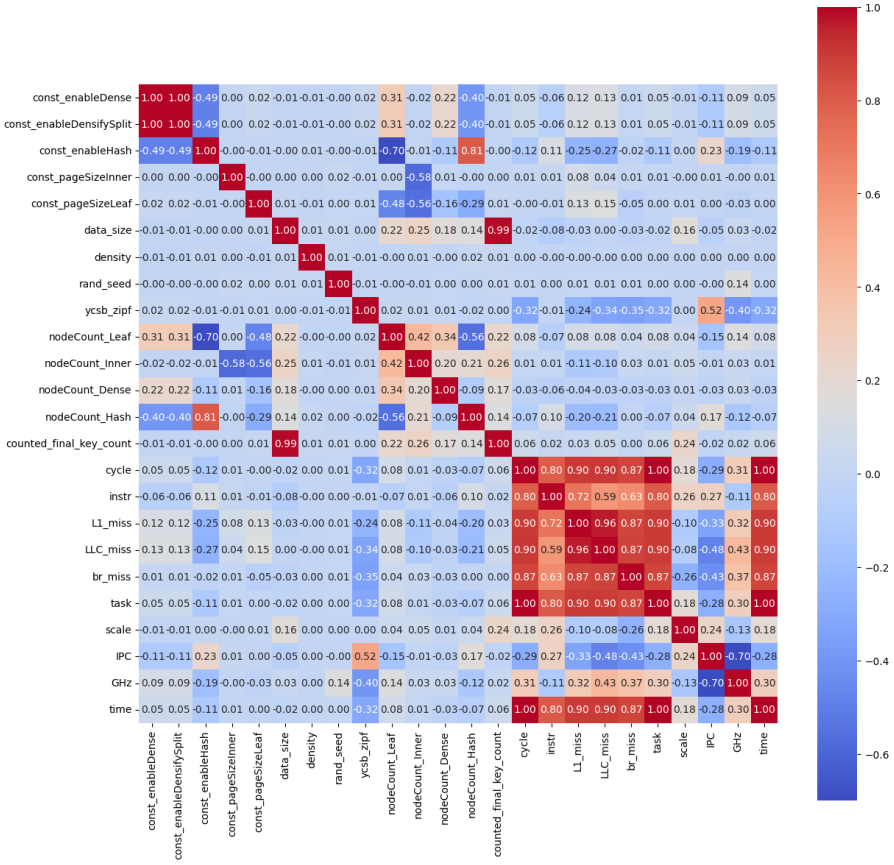


Figure 4.1: Correlation matrix of the full dataset

Figure 4.2 represents the correlation coefficients between the feature *time* and all other features. Some features are omitted from this figure, due to their constant values resulting in undefined behavior. The figure further highlights the high correlation between the output features and time, by displaying the output features in red and input features in blue. This highlights the importance of excluding these features as inputs to the ML

models. Including these features, would make it much easier for the ML models to calculate the feature *time* with high precision. However, such a scenario would be an inaccurate representation, since the output features are not available for new values and the goal of this thesis is to find correlation between input features and the feature *time*.

Additionally, it can be observed that looking at the input features and their statistical correlation to feature *time*, does not give promising insights, which is why ML models are used to learn more complex relationships between the input and the feature *time*.

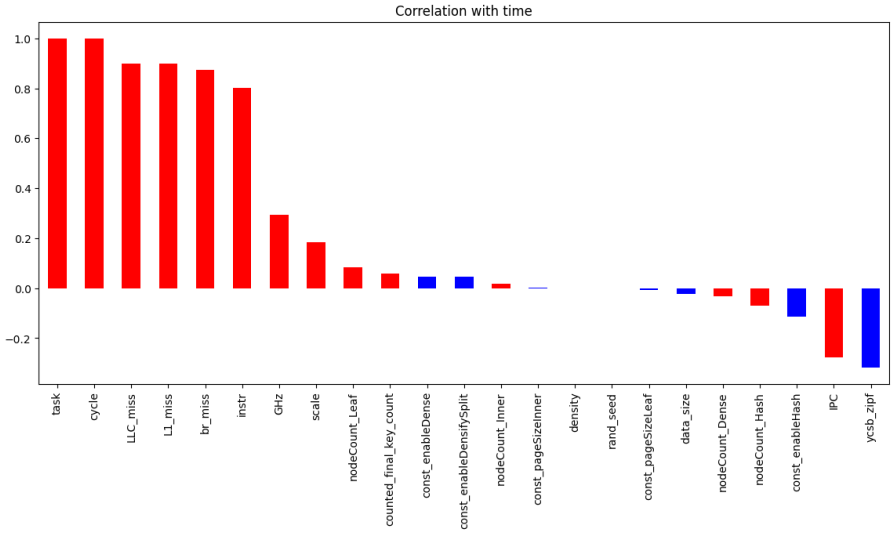


Figure 4.2: Correlation of features with feature *time*

4.2 Classification models

This section describes which classification models are used and what they do in a general sense. Before the individual ML models are used, however, one needs to define, what is classified and what the metrics are used to distinguish good from bad models. In this thesis, the classification problem is defined as follows:

Since the feature *time* is a continuous value, it is not reasonable to classify each possible value as one class. Therefore, the values are split into 10 sections, based on the values. The class 1 indicates that the value is in the lowest 10% of the data, meaning that it is among the fastest values. The class is stored as the feature *percentile bracket*. For this, the following code is used to create the feature:

```
df['percentile bracket'] = pd.qcut(df['time'], q=10, labels=False,
                                   duplicates='drop') + 1
```

To make sure that the ML model can be evaluated, a metric needs to be used on data, which the model has not been fed beforehand. Therefore, the data is split into a train- and a test set. This thesis uses an 80:20% train/test split, which according to Joseph [19] is a common ratio. In the paper [19], Joseph suggests using a ratio depending on the number of parameters, however, since this thesis aims to get insights into the data and not the best performing model, the simpler, static ratio split is applied.

Using the test set, a metric can calculate the usefulness of the model. The metric being used in this thesis is the accuracy score, as defined by *scikit-learn* in [20]. This score calculates the fraction of the correctly classified samples from the test set.

Since the data has been split between the four values of feature *op*, there are four models trained and therefore four metrics are compared per model type.

4.2.1 Decision Tree

The Decision Tree classification, according to Zhou's definition in "Machine Learning" [21], is a ML model, which classifies data by creating a set of rules in a tree structure. The rules can be thought of if-then rules. At the leaf of the tree, the class of the value is defined. The Decision Tree can be built using a recursive algorithm, which continuously splits the set of data points in a node based on defined splitting rules until one of the termination conditions is met. In the book "Machine Learning" [21] Zhou defines the following three conditions on a node:

- All data points are of the same class.
- All data points have the same feature values, or there is no more feature to split.
- There is no data point in the node.

The implementation that is used in this thesis [22] allows further terminating conditions, such as the maximum depth of the tree, the maximum number of leaf nodes or a minimum number of data points required for a split of a node.

In the book "Machine Learning" [21], Zhou further outlines that in order to determine the split, a higher *purity* of the new child nodes is attempted to be achieved. The *purity* is commonly measured by the *information gain*, given in formula 4.3. Calculating the *information gain* requires the *entropy* of data, which is given in formula 4.2.

$$\mathrm{Ent}(D) = -\sum_{k=1}^{|y|} p_k \log_2 p_k \tag{4.2}$$

where:

$$\begin{aligned} \text{Ent}(D) &\text{ is the entropy of the data set,} \\ |y| &\text{ is the number of classes,} \\ p_k &\text{ is the proportion of the } k\text{th class} \end{aligned}$$

It is assumed that the classes are labelled $k = 1, 2, ..., |y|$.

$$\text{Gain}(D, a) = \text{Ent}(D) - \sum_{v \in \text{Values}(a)} \frac{|D_v|}{|D|} \text{Ent}(D_v) \tag{4.3}$$

where:

$$\begin{aligned} \text{Ent}(D) &\text{ is the entropy of the dataset } D \\ \text{Values}(a) &\text{ is the feature} \\ |D_v| &\text{ is the number of instances for which feature } a \text{ has value } v \\ \text{Ent}(D_v) &\text{ is the entropy of the dataset } D_v \end{aligned}$$

The library *scikit-learn* [22] by default uses the splitting criterion *Gini impurity* instead of *entropy*. The *Gini impurity*, according to Zhou [21] is defined in formula 4.4.

$$\text{Gini}(D) = 1 - \sum_{k=1}^{|y|} p_k^2 \tag{4.4}$$

where:

$$\begin{aligned} \text{Gini}(D) &\text{ is the Gini impurity of the dataset } D \\ |y| &\text{ is the number of classes} \\ p_k &\text{ is the proportion of the } k\text{th class} \end{aligned}$$

During experimentation of this thesis, practically, the results between using the criterion *entropy* and *Gini impurity* yielding minimal differences below a 0.5% accuracy. Since no criterion yielded consistently better results, for the context of this thesis the criterion *Gini impurity* is chosen.

Zhou [21] states that these formulae assume discrete values, which is why for continuous values a simple discretization technique is used, where the middle index of the sorted values is taken. This middle index is then used for splitting the feature values into two intervals, which can be taken as discrete values to calculate the splitting criterion.

4.2.2 Random Forest

The Random Forest classifier by *scikit-learn* [23] takes the idea of a Decision Tree classifier and trains multiple Decision Trees with varied configurations. According to Zhou [21],

the variations vary the available features at each step. Additionally, the available data points are varied, with repeating data points across variations.

Zhou [21] further explains, that this method is part of the idea of Ensemble learning, where multiple models are trained and each value is then evaluated on all models and later the results are combined to give a final result. After training a given number of Decision Trees, the class is determined by giving one vote to each model. This method provides the benefit of better controlling over-fitting the models, since each time only a subset of data points from the training set is used.

4.2.3 Gradient Boosting

According to Masui [24], Gradient Boosting is an ensemble technique, which is built on the idea of training a "weak" decision tree and iteratively improving on it, by taking the loss function into account and using gradient descent. The loss function used in this thesis is the *log_loss*, which is the idea used in Logistic Regression 4.2.5 and according to *scikit-learn* [25] is a good idea for classification with probabilistic outputs. The figure 4.3 visualizes the flow of a Gradient Boosting model. The weights assigned to each "weak" decision tree is based on the loss function. The mechanism of how gradient descent is used in each iteration during training is out of the scope of this thesis.

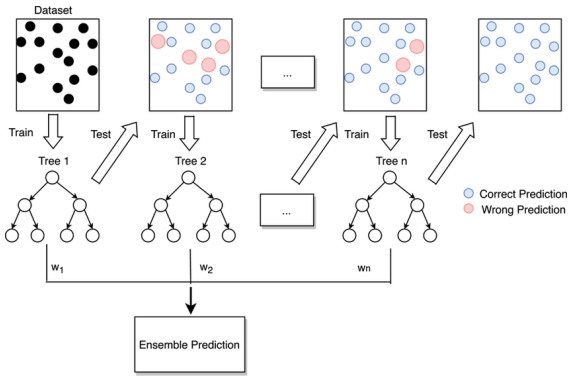


Figure 4.3: Gradient Boosting flow diagram [26]

4.2.4 Neural network

In the book [21] Zhou describes that there exist a number of types of neural networks, like Radial Basis Function networks, Adaptive Resonance Theory networks or Self-Organizing Map networks. However, the one described and used in this thesis is the Multilayer Perceptron (MLP) Classifier as defined by *scikit-learn* [27]. According to Bishop [14] the MLP is considered the most successful model, as it is of practical value for statistical pattern recognition. The MLP is also known as a feed-forward neural network. An example for an MLP is visualized in figure 4.4. The layer on the left is known as the input layer, where each node, also known as a neuron, represents one

feature. The rightmost layer, the output layer, contains the output features, which in this example case is a scalar value, which in the case of MLP Classifiers would be the class of the value. The middle layer represents the hidden layer. There can be multiple hidden layers defined, each with a different amount of neurons. Additionally, each layer contains a *bias*, typically referred to as the 0th weight, w_0, in the layer. As stated by

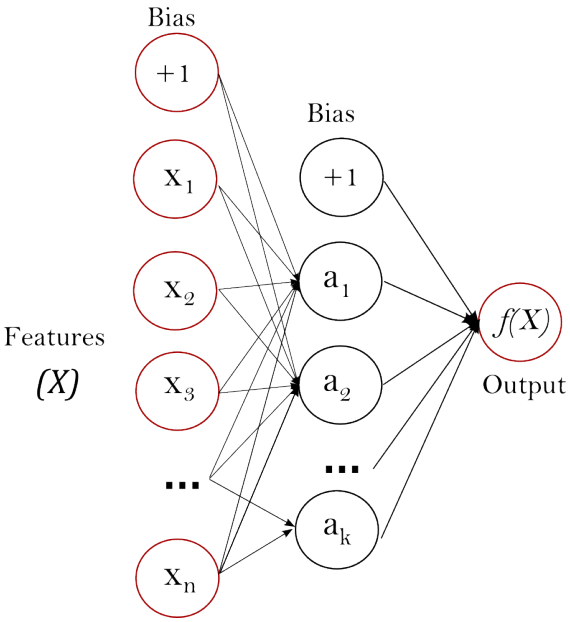


Figure 4.4: MLP with one hidden layer [27]

Bishop [14] and *scikit-learn* [27], each neuron, take the values from the previous layers and calculate a weighted sum $w_0 + w_1x_1 + w_2x_2 + ... + w_mx_m$, which consequently is put into a typically nonlinear activation function. Choosing the activation function can be done based on domain knowledge about the underlying data and its distribution.

During experimentation of this thesis, the following activation functions yielded results with the highest accuracy. The *ReLu* (Rectified Linear Unit) function $f(x) = max(0, x)$ and the *Identity* function $f(x) = x$. In the end, *ReLu* resulted in a slightly higher accuracy, therefore the results of this activation function are presented in the chapter on findings 6.

The neural network training process can be defined as follows:

1. **Initialization**: The weights and biases are assigned random values.

2. **Forward Pass**: At each layer the data from the previous layer is weighted summed, transformed and passed forward to the consequent layer.

3. **Loss Calculation**: A loss function compares the calculated output for a validation set, which is not used in forward pass, with the actual output.

4. **Backpropagation**: The loss is then propagated backwards through the network by calculating gradients using mathematical techniques like the chain rule in calculus.
5. **Gradient Descent**: The gradients calculated in backpropagation are used to adapt the parameters in the loss-minimizing direction. Multiple Gradient Descent algorithms can be used, including *Adam*, defined in [28], and *Stochastic Gradient Descent*.
6. **Iterations**: Steps 2 - 5 are iterated, gradually improving the network's performance. The training is terminated once the loss stops to decrease or starts to increase. Other terminating conditions can be set to the maximum number of iterations, time limit or by integrating other performance metrics.

This thesis does not delve into the detailed mathematical principles underlying backpropagation and gradient descent techniques. However, one characteristic of MLP Classifiers is that it is sensitive to feature scaling, which is why *scikit-learn* [27] suggests using the *StandardScalar*, as defined in [29], which transforms the data in such a way that each feature has a mean of zero and a standard deviation of one. This indeed improved the performance of some models trained in this thesis. In one case, this improvement increased the accuracy from around 4% to more than 80%. The precise results are outlined and discussed in chapter 6.

4.2.5 Logistic regression

Despite what the name suggests, logistic regression actually is a classification model. The base case for logistic regression considers a binary classification, meaning that only two classes are considered, true and false. The logistic function, otherwise known as logistic sigmoid or sigmoid function, is used to identify the probability of an input belonging to a class. The logistic function is outlined in the formula 4.5.

$$g(z) = \frac{1}{1 + e^{-z}} \tag{4.5}$$

The figure 4.5 visualizes the binary case of a logistic regression with one feature on the x-axis. The threshold value is set to 0.5 implying that results of the logistic function above 0.5 are classified as the class 1.

The idea of binary classification case, can be extended to support multiple classes. The implementation used in this thesis extends the case by using *multinomial logistic regression* with *cross-validation*. This thesis does not go into the details of the multinomial logistic regression model, however the main idea can be summarized as follows.
To predict the class of a value the model takes a weighted sum of the input features for each possible class, puts the output into the softmax function, as defined in 4.6. These results are then put into the logistic function, which are then interpreted as the probability for each class. The most likely class is chosen to classify the value.

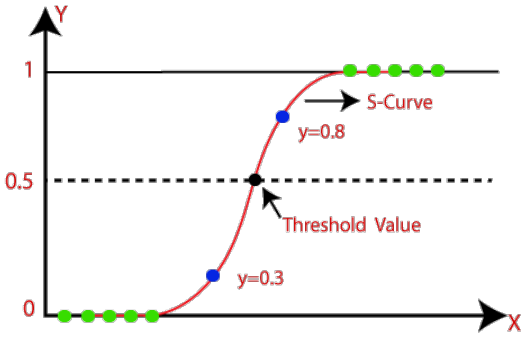


Figure 4.5: Logistic regression example [30]

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{4.6}$$

Following the suggestion of [31], to prevent overfitting and performing poorly on unseen data the cross-validation *scikit-learn* implementation [32] is used in this thesis. The concept is visualized in figure 4.6. The training set is taken and split into k disjoint subset. Afterward, each time a different subset is omitted from the training set and used as the testing set for that iteration. Following this, the results are averaged to create an evaluation result. This thesis set the parameter k to 10, as it is, according to Zhou [21], the most commonly used value.

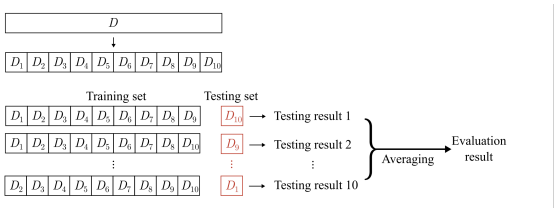


Figure 4.6: Cross validation example [21]

4.3 Regression models

This section describes which ML regression models have been used and gives an overview of how the individual models work. Since regression models are able to target a continuous feature, the target feature is the feature *time*, without the need for more preprocessing.

To evaluate the performance of the models, the same procedure as with classification models with an 80/20 train/test split of data is employed. In contrast to the classification models, this thesis uses two metrics to evaluate the individual models.

RMSE stands for Root Mean Squared Error, which, as the name suggests, is the root of the **MSE** metric. The MSE metric itself is the most commonly used metric, according to Zhou [21], and it takes the mean of the squared errors, as defined in 4.7.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4.7}$$

Despite the fact that RMSE is a metric in the same units as the target variable, this thesis uses the metric MSE. This is due to the already minuscule scale of the feature *time*.

Additionally, since very small numbers are expected as errors, it is a good idea to make sure that the models actually perform better than the simplest model possible, which would be taking the average of the entire dataset. The R^2, as described by *scikit-learn* in [33], also known as the coefficient of determination, typically yields results between 1, the perfect predictions, and 0, the average of the dataset. It can, however, also yield negative values, when the models perform more poorly than the average-case. The R^2 is defined in formula 4.8

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{4.8}$$

where:

R^2 is the coefficient of determination
n is the number of data points
y_i is the actual value of the i^{th} record
$\hat{y}_i$ is the predicted value of the i^{th} record
$\bar{y}$ is the mean of actual values of all records

4.3.1 Linear regression

According to Bishop [14] the simplest linear model, and also the one used in this thesis, is the linear regression model. The model attempts to create a linear combination of input variable to create an output value. The model itself can be defined as in formula 4.9.

$$y = w_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n \tag{4.9}$$

where:

y is the target feature
$x_1, x_2, \ldots, x_n$ are the features
$w_0, w_1, \ldots, w_n$ are the weights representing the slopes of the regression line

As Alto describes in [34], to estimate the weights of the Ordinary Least Squares (OLS) method is used, which minimizes the sum of squared differences. The details of this method include differentiating the sum of squared differences with respect to each coefficient, setting the derivatives to zero, and solving the equation.

The implementation by *scikit-learn* [35] assumes multiple characteristics. The main assumption is that there is a linear relationship between the input features and the target feature. Additionally, it is assumed, that each record is independent of the other records, which is the case in this thesis.

4.3.2 Random forest regression

According to *scikit-learn* [36], decision trees can be used for classification, as well as regression. The key lies difference is that instead of taking the most occurring class in the leaf node, the mean of the target feature is taken. Similarly, to the decision tree for classification, the one for regression, can also face issues with over-fitting. The figure 4.7, outlines this issue, by highlighting how defining the maximum depth can influence the outcome of the mode.

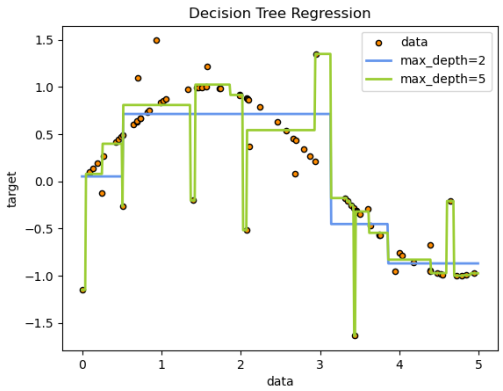


Figure 4.7: Example for decision tree regression model [37]

To avoid the overfitting issue, the model RandomForestRegressor [36], is trained which works on the same idea as described for the Random Forest Classification.

4.3.3 Gradient boosting regression

Combining the idea behind Decision Tree Regression and Gradient Boosting Classification, the Gradient Boosting Regression model can be built. The major difference Gradient Boosting Classification is the fact that the underlying trees being built are regression in nature.

4.4 Pattern recognition

As described in the blog [38], pattern recognition is a set of algorithms that identify repetitive patterns in data. The pattern recognition algorithms described in this thesis differ from the classification and regression models in a major aspect. Instead of predicting target features on an unseen dataset, they bring insights into the underlying dataset, by highlighting patterns. This is very useful for the use-case of this thesis, due to the explainable nature.

Before the algorithms described in this section are used, it is to be highlighted that a different dataset is used and additional preprocessing steps are executed on top of the ones described in chapter 3. Since the algorithms deal with discrete data with binary features, the dataset needs to be converted into such data. To ease these steps, the input parameters during benchmarking are limited to a defined set of possible values, instead of taking a random value in a given range. Additionally, less features are varied during the benchmarking, to avoid bloating the dimensionality of the data. To achieve the binary features, after dropping static features, **one-hot encoding** of all features is implemented. The exception to this is the feature *time*, which before being one-hot encoded is split into three sections, similarly to the technique used for the classification models. The choice of three buckets instead of 10 used in classification stems from the limited variations of the input. The goal for these algorithms, in the context of this thesis, is to be able to identify which page size of leaf nodes performs faster for which B+ tree optimizations.

4.4.1 Apriori

The Apriori algorithm, introduced by Agrawal et al. in [39], is used to identify frequent itemsets. These itemsets are later used in mining for association rules. The algorithm considers only one metric, the **support** which is defined as the fraction of the amount an itemset occurs in a dataset. After defining a support threshold, the algorithm returns the itemsets with a higher support.

The following example from the *mlxtend* library [40] outlines the algorithm. The transactions from figure 4.8 yield for a minimum threshold of 60% the frequent itemsets in figure 4.9.

	Apple	Corn	Dill	Eggs	Ice cream	Kidney Beans	Milk	Nutmeg	Onion	Unicorn	Yogurt
0	False	False	False	True	False	True	True	True	True	False	True
1	False	False	True	True	False	True	False	True	True	False	True
2	True	False	False	True	False	True	True	False	False	False	False
3	False	True	False	False	False	True	True	False	False	True	True
4	False	True	False	True	True	True	False	False	True	False	False

Figure 4.8: Apriori input example [40]

	support	itemsets
0	0.8	(Eggs)
1	1.0	(Kidney Beans)
2	0.6	(Milk)
3	0.6	(Onion)
4	0.6	(Yogurt)
5	0.8	(Eggs, Kidney Beans)
6	0.6	(Eggs, Onion)
7	0.6	(Kidney Beans, Milk)
8	0.6	(Kidney Beans, Onion)
9	0.6	(Yogurt, Kidney Beans)
10	0.6	(Kidney Beans, Eggs, Onion)

Figure 4.9: Apriori output example with 60% support threshold [40]

4.4.2 Association rules

The *mlxtend* library describes association rules in its documentation [41], as rules in the form of $X \rightarrow Y$ with X and Y being disjoint itemsets. The rules imply the likely existence of Y given that X occurs.

The implementation, from the *mlxtend* library [41] used in this thesis, takes the frequent itemsets created by the Apriori algorithm and generates rules based on those. It also allows limiting the output, by allowing to set a threshold for a metric. The implementation calculates various metrics for the rules, such as the ones used in this thesis support, confidence and lift.

For $A \rightarrow C$, and A meaning antecedent itemset and C meaning consequent itemset the metrics used are defined by *mlxtend* [41] as follows:

- **Support:** Equal to the support of the union of the itemsets, described in Apriori algorithm.

$$\text{support}(A \rightarrow C) = \text{support}(A \cup C), \quad \text{range: } [0, 1] \tag{4.10}$$

- **Confidence:** The conditional probability of finding the consequent item in a transaction given that the transaction contains the antecedent itemset.

$$\text{confidence}(A \rightarrow C) = \frac{\text{support}(A \rightarrow C)}{\text{support}(A)}, \quad \text{range: } [0, 1] \tag{4.11}$$

- **Lift:** The ratio of the observed support to that expected if the items were statistically independent. It indicates the strength of a rule over random chance. Statistically independent itemsets would have the lift equal to 1.

$$\text{lift}(A \rightarrow C) = \frac{\text{confidence}(A \rightarrow C)}{\text{support}(C)}, \quad \text{range: } [0, \infty] \tag{4.12}$$

4.4.3 Optimal page size of leaf nodes

To identify the most optimal size of leaf nodes, heatmaps can be generated, which visualize the features *payload_size* and *data_size* together with the best performing page size of leaf nodes. For this, two approaches are taken. One of the approaches is taking the dataset created for pattern recognition and training regression models on it. Then for each combination of the features *payload_size*, *data_size* and *const_pageSizeLeaf* the best performing page size is shown in the respective section. The other approach is skipping the regression model step and taking the empirical mean performance instead to identify the optimal page size.

The results of these approaches are outlined in chapter 6.

5 XAI

In his book "Interpretable Machine Learning" [42] Molnar describes the fundamentals of XAI. This chapter is based on Molnar's findings. The idea behind XAI is to be able to explain decisions made by an Artificial Intelligence, or ML model. The ability to understand the underlying model's decisions can be crucial for real world applications of ML models. One goal of XAI is to increase the social acceptance of models by giving insights into why a model has made a certain decision. For example, in the application of self-driving cars, in the case of an accident, it is useful to understand why the car has made the decisions it has made and possibly even prove that the behavior prior to the accident was correct.

For the case of this thesis, the goal of applying XAI to the trained ML models is to be able to understand which optimizations benefit from which configurations.

5.1 Characteristics of XAI techniques

Since the topic of XAI covers numerous algorithms, their characteristics can be broken down into whether a technique is **model-agnostic** or **model-specific** and whether it is a **local** or **global** explanation. The figure 5.1 provides an overview of these characteristics and how they relate to each other.

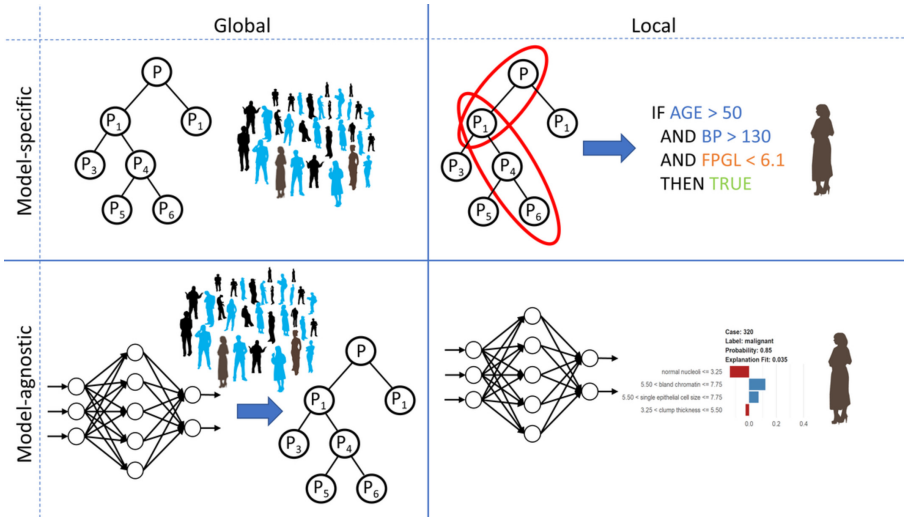


Figure 5.1: Overview of characteristics of XAI techniques [43]

Model-agnostic vs. model-specific: Model-agnostic refers to techniques that can be applied to any ML models. They consider the models as a black-box, meaning that they analyze the relationship between the inputs and the outputs and do not look deeper into the internal functionality of the models. Model-specific techniques are able to analyze the internal functionality of the models, but are limited to a specific type of models. These type of models can for example explain the weights of a linear model, or interpret the nodes in a neural network.

Local vs. global: Local XAI techniques focus on interpreting the results of individual values, while global techniques interpret the model as a whole. Local techniques are not further discussed in this thesis, since interpreting individual runs of the benchmarking program does not provide a general picture about B+ tree configurations. They are more useful for real-world applications, where individual model decisions need to be explained.

Model transparency: Some machine learning models are considered transparent, or white-box, which refers to the interpretable nature of the model itself. Transparent models are considered to be easily humanly interpretable. A common case for this is a decision tree model, which is also used in figure 5.1 to represent explainable models. Not all transparent models are necessarily easily interpretable. For example, in the case of a decision tree model, if the tree has a big depth, the interpretations, can get incomprehensible. Linear models can be globally and locally explained, by analyzing the feature weights, however, there is the assumption that all features are statistically independent of each other.

Out of the numerous algorithms that are part of XAI, this thesis focuses on the algorithms that are suited for the use-case of this thesis. Model-specific interpretations are covered in the chapter 6. It should be noted that the mathematical background of the individual algorithms is out of the scope of this thesis. Instead, this thesis aims to explain core ideas, the advantages, and disadvantages of the techniques.

5.2 Permutation Feature Importance

Permutation feature importance is a global model-agnostic method. The goal of this method is to identify, which features are important and which are less relevant. This is done by shuffling values of individual features and measuring the error of the model. One example of a permutation feature importance output, provided by the *scikit-learn* implementation [44] used in this thesis, is presented in figure 5.2. The plot highlights that the features *sex*, *pclass* and *age* have the highest importance on an example model. Since various permutations are tested per feature, a box plot represents the output.

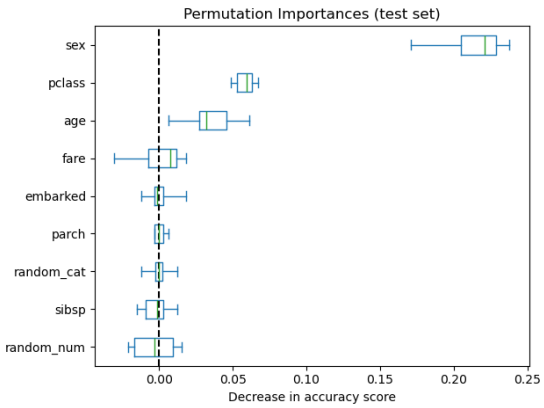


Figure 5.2: Example output for permutation feature importance [44]

5.3 SHAP - SHapley Additive exPlanations

SHAP is a technique that at its core is a local model-agnostic method. The idea behind this method is that a single record can be taken, and an individual feature is permuted to identify how the permutation impacts the result. The importance is measured in a so-called *Shapley value*. KernelSHAP [45] is an algorithm used in this thesis, which trains a special weighted linear regression model to calculate the Shapley values of a record. For this thesis, however, local explanations are not relevant.
Nevertheless, KernelSHAP is used in this thesis, since applying the algorithm on multiple samples and plotting them in a **SHAP summary plot**, can give a broader, global picture about a model. An example for such a plot is given in figure 5.3. This plot is beeswarm plot, where each dot represents a Shapley value of a record for each feature. The color of a dot indicates the value of a feature of a record from low to high. The density of the dots is visualized by stacking dots along the y-axis. The features are sorted from the highest mean of absolute Shapley values to the lowest.

One can compare the SHAP summary plot to the permutation feature importance. They both intend to outline the feature importance, however the difference lies in the underlying structure of the algorithms. Permutation feature importance calculates the error by permuting features for all values, while SHAP feature importance takes the average impacts of features for individual values.

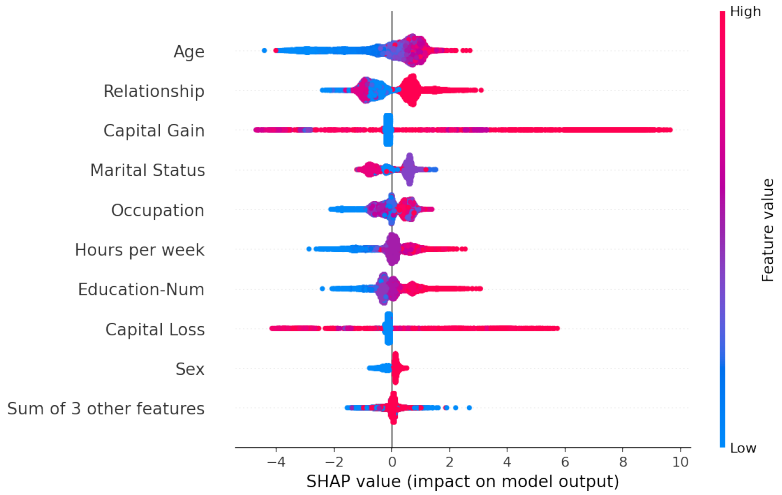


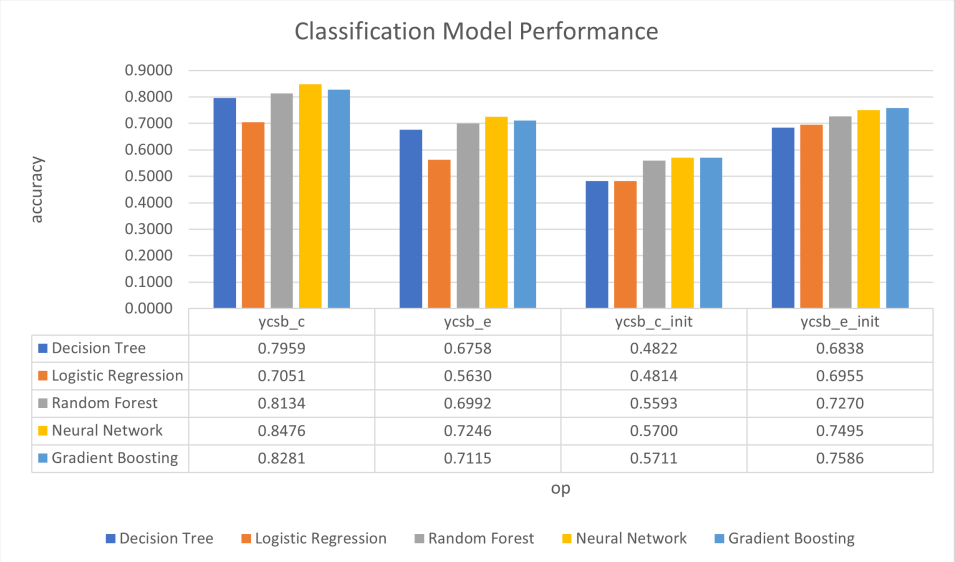
Figure 5.3: Example output for SHAP feature importance [46]

6 Findings

This chapter collects the results of the individual models, the XAI algorithms, and pattern recognition algorithms. The results are then analyzed and interpreted.

6.1 Classification Models

After doing all the preprocessing steps mentioned in chapter 3, the classification models have yielded the results visualized in figure 6.1. The results indicate that ensemble methods, random forest and gradient boosting, seem to perform better than a classic random tree. The neural network classifier, however, performs best for all operations, with the exception of *ycsb_e_init*, where gradient boosting performs better by a minimal difference of less than one percent. A surprising observation is the difference of performance between the initialization operations. This is attempted to be explained by the XAI techniques.



	ycsb_c	ycsb_e	ycsb_c_init	ycsb_e_init
Decision Tree	0.7959	0.6758	0.4822	0.6838
Logistic Regression	0.7051	0.5630	0.4814	0.6955
Random Forest	0.8134	0.6992	0.5593	0.7270
Neural Network	0.8476	0.7246	0.5700	0.7495
Gradient Boosting	0.8281	0.7115	0.5711	0.7586

Figure 6.1: Graph and data for performance of classification models

The generally best performing model in all cases, the neural network, has been further analyzed for each operation. This is done using XAI techniques mentioned in chapter 5. It should be noted that similar figures have been created for the gradient boosting mod-

els during experimentation, and therefore only the insights from the best performing models are shown and discussed.

6.1.1 Permutation Feature Importance

Figure 6.2 depicts four box-plots generated by applying the permutation feature importance algorithm on the neural networks created for the four operation types.

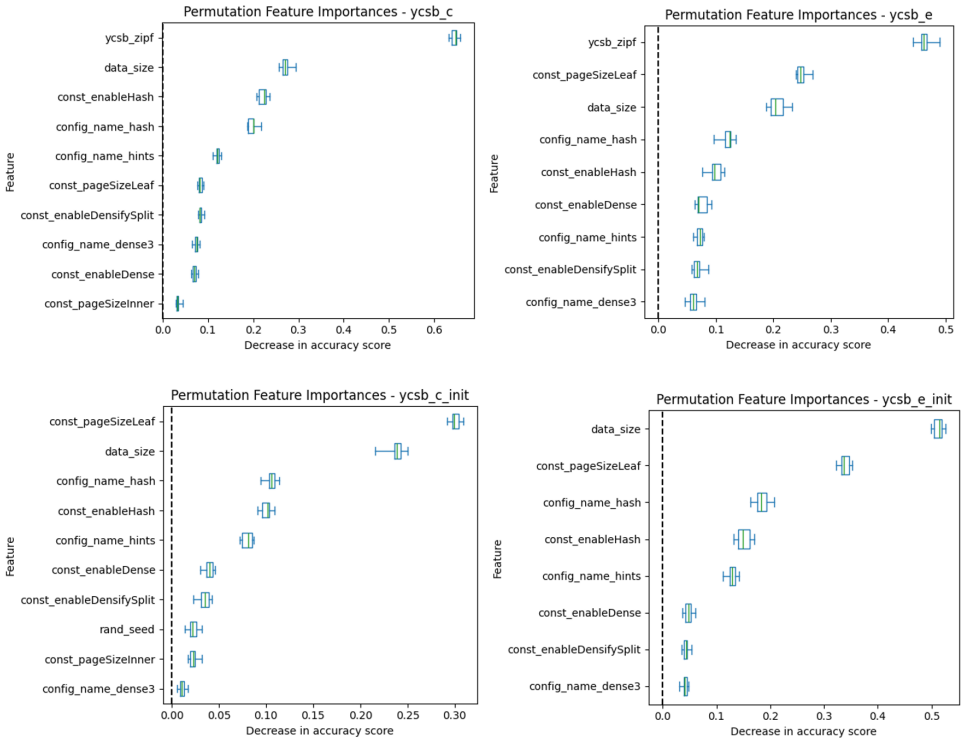


Figure 6.2: Neural network permutation feature importances

The most notable result is that the feature *ycsb_zipf*, which describes how skewed the distribution of accessed keys is, is the most important feature for the benchmarking operations. For the initialization, the feature has no importance, which can be explained by the fact that the initialization operations do not consider the skewness during the run.

Furthermore, it can be seen that for operations that perform insertions (*ycsb_e*, *ycsb_e_init*, *ycsb_c_init*) the feature *const_pageSizeLeaf* has a higher importance. The *data_size* is important for all operations, which can be explained by the fact that the B+ tree grows with more data and therefore the operations require more time to read and insert.

Since the features *const_enableHash* and *config_name_hash* are highly correlated, it is understandable why the features are assigned similar importance. The slight differences could be explained by the random permutations that occur during the creation of the plot.

One should not assume that a higher importance of a configuration name implies a better performance. For example, the higher importance of *config_name_hash* than *config_name_dense3* only implies that when the models classify more based on the *config_name_hash* than *config_name_dense3*. To understand which configurations lead to which classification, one could use local XAI techniques or use pattern recognition to identify these patterns. This, however, is beyond the objectives of this paper.

An unexplained phenomenon is the differences between the initialization operations. The feature *data_size* seems to be much more important for the operation *ycsb_e_init*, compared to *ycsb_c_init*. This could potentially explain the difference in the performance for the models. It could lie in the fact that for these two operations, the feature *scale* is calculated differently, which impacts the target feature *time* and consequently the models too. The feature *scale* is set for the *ycsb_c_init* to the number of keys, while for *ycsb_e_init* 2.5% is subtracted from that number. The subtraction is due to the fact that operation *ycsb_e* is expected to insert that amount of keys in its benchmark. However, validating this theory falls beyond the purview of this thesis.

6.1.2 SHAP Summary Plot

Figure 6.3 depicts four SHAP summary plots generated by applying the KernelSHAP algorithm and plotting the results of the neural networks created for the four operation types.

Comparing the results between permutation feature importance plots and SHAP summary plots, the SHAP summary plots corroborate the conclusions drawn from the permutation feature importance plots. The SHAP summary plots, however, provide a few more insights. The colors in figure 6.3 indicate the plot relative feature value, with low values being blue and high values being red. These colors can then be compared with the Shapley values. For example, a cluster of blue dots for a feature on the positive side of the x-axis indicates that low values of the feature result in a higher target class, meaning a lower performance.

Based on this, it can be concluded that higher values of feature *ycsb_zipf* result in better performance for the operations *ycsb_c* and *ycsb_e*. The feature *ycsb_zipf* is generated randomly for each run between 0 and 1.5, with 0 meaning a normal distribution, 1 meaning a Zipfian distribution and above meaning even more skewed data. According to Claesen in [47], a Zipfian distribution indicates that fewer values are being accessed. Therefore, it can be interpreted that higher *ycsb_zipf* values indicate the cache can be

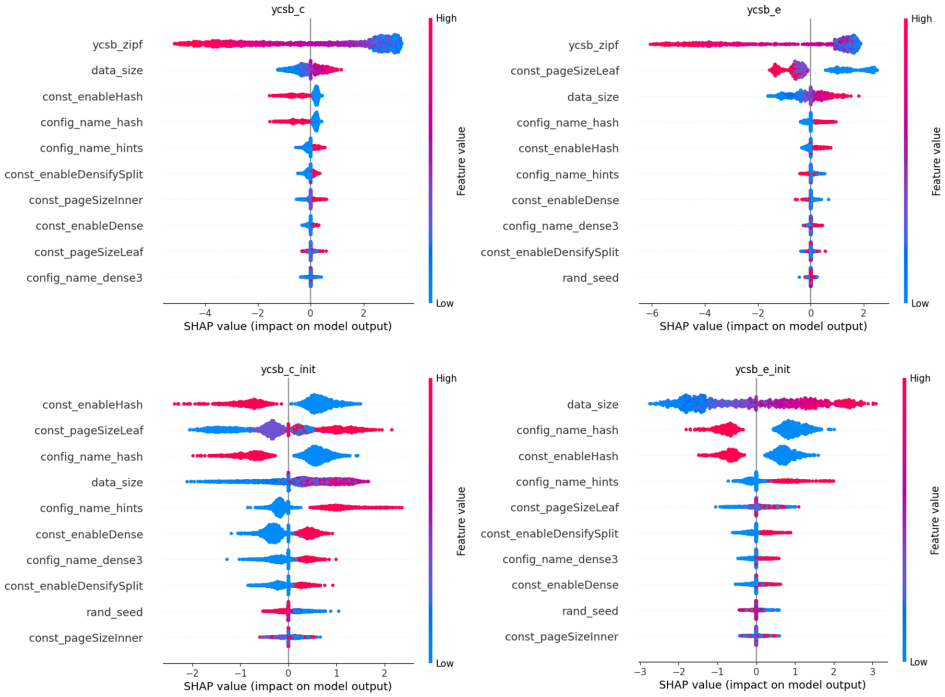


Figure 6.3: Neural network SHAP summary plots per operation

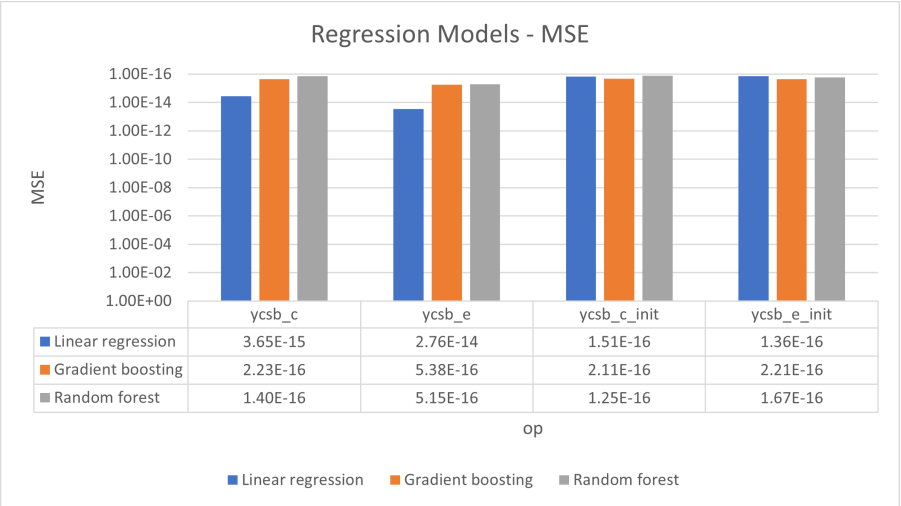
better utilized, leading to better performance.

For the feature *data_size* it is clearly visible at all operations that higher a smaller amount of data, leads to smaller trees and therefore a smaller class, meaning better performance.

Furthermore, for the initialization operations, one can notice that lower page sizes of leaf nodes result in better performance. This indicates that for the insertion operations benefit from lower leaf page sizes. Despite that, the *ycsb_e* figure could indicate a contradictory pattern, both patterns can be true. Considering that the *ycsb_e* consists of 95% scan operations and only 5% insertions, the scan operations outweigh the impact of the insertion operations. The scan operations, seem to benefit from higher leaf page sizes. This could be explained by the fact that higher leaf page sizes, lead to more consecutive values being stored in the same node and therefore during scan operations, fewer nodes need to be traversed. Insertion operations, on the other hand, might consider fewer values in a node more beneficial since it reduces the number of disk accesses required to write in a node. As a result, it can improve the performance of insertion operations, as disk I/O is often the bottleneck. This performance improvement seems to outweigh the drawback of having a higher depth and requiring more splitting operations.

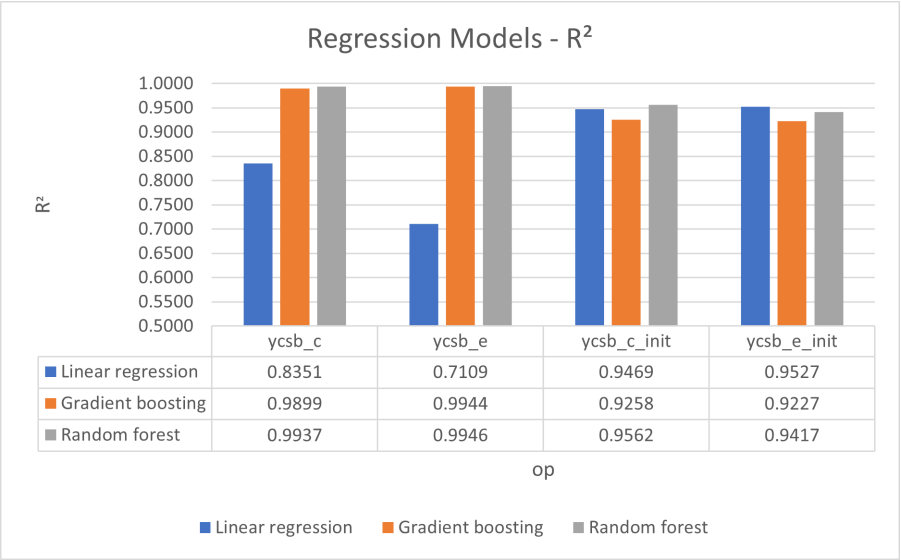
6.2 Regression Models

For regression models, the metrics have yielded the results depicted in figure 6.4 for metric MSE and figure 6.5 for metric R^2. From these figures, one can conclude that the random forest model has yielded the best results in both metrics, with the exception of *ycsb_e_init*, where the linear regression model performs better by a minimal margin.



	ycsb_c	ycsb_e	ycsb_c_init	ycsb_e_init
Linear regression	3.65E-15	2.76E-14	1.51E-16	1.36E-16
Gradient boosting	2.23E-16	5.38E-16	2.11E-16	2.21E-16
Random forest	1.40E-16	5.15E-16	1.25E-16	1.67E-16

Figure 6.4: Graph and data for MSE performance of regression models



	ycsb_c	ycsb_e	ycsb_c_init	ycsb_e_init
Linear regression	0.8351	0.7109	0.9469	0.9527
Gradient boosting	0.9899	0.9944	0.9258	0.9227
Random forest	0.9937	0.9946	0.9562	0.9417

Figure 6.5: Graph and data for R^2 performance of regression models

6.2.1 Permutation Feature Importance

Producing insightful permutation feature importance plots has presented significant challenges, primarily stemming from the scale of the metrics and target feature. As a result, this particular step has been omitted from the scope of this thesis.

6.2.2 SHAP Summary Plot

Due to the scale of the target feature *time*, the SHAP summary plots look differently, more clustered compared to the ones for classification. Nevertheless, similar outcomes can be deducted from the figure 6.6. For example, the most important features remained the same, with only minor differences in order. Additionally, the same observations regarding features *ycsb_zipf*, *const_pageSizeLeaf* and *data_size* and their impact on the target feature can be made. However, due to the low interpretability caused by the scale of the target feature, the confidence for these statements is lower.

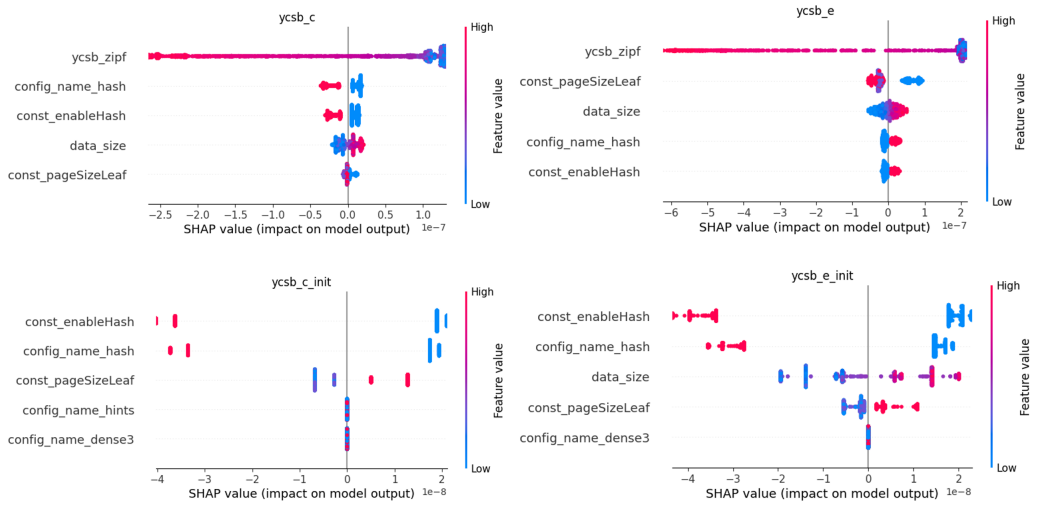


Figure 6.6: Random forest SHAP summary plots per operation

6.3 Pattern recognition

This section describes the results yielded by performing the Apriori algorithm and consequently the association rules mining algorithm. As described in chapter 3 on preprocessing, the data used in this section is generated with a different data generation script than the one used for model training. The feature *time*, is split into three buckets, as described, along with other necessary additional preprocessing steps, in section 4.4. Bucket_1 indicates the fastest times and therefore best performance.

In the following results, the Apriori threshold has been set to 5%, and the association rules threshold for confidence to 40%. The result is then filtered to association rules, that are relevant to the performance. Additionally, to simplify the output, association rules that are redundant, by being majorly explained by a smaller antecedent, have been removed from the output. Furthermore, the consequent support is not displayed, since the buckets are split equally, meaning each has a support of close to 33%. Metrics not introduced in this thesis are also filtered out.

6.3.1 Association rules

ycsb_c: As figure 6.7 outlines, the major factor found for the purely read operation benchmark is feature *data_size*. Lower values for *data_size* indicate better performance. Another notable result is that the biggest leaf page size for the benchmarking, 8192, tends to perform worst for this operation.

antecedents	consequents	antecedent support	support	confidence	lift
'data_size_1000000'	'bucket_1'	0.1695	0.1695	1.0000	2.9974
'data_size_6000000'	'bucket_3'	0.1662	0.1536	0.9241	2.7724
'data_size_2000000'	'bucket_1'	0.1671	0.1367	0.8178	2.4514
'data_size_3000000'	'bucket_2'	0.1692	0.1336	0.7896	2.3708
'data_size_5000000'	'bucket_3'	0.1643	0.1133	0.6896	2.0689
'data_size_4000000'	'bucket_2'	0.1638	0.1056	0.6449	1.9365
'payload_size_128'	'bucket_3'	0.1483	0.0704	0.4746	1.4239
'const_pageSizeLeaf_8192'	'bucket_3'	0.3462	0.1495	0.4317	1.2952
'payload_size_64'	'bucket_3'	0.1478	0.0632	0.4275	1.2826
'payload_size_4'	'bucket_1'	0.1529	0.0652	0.4267	1.2790

Figure 6.7: Association rules for *ycsb_c*

ycsb_e: For the operation *ycsb_e*, figure 6.8 displays that the size of the payload is a decisive factor. For example, executions with payload size of 256, which occur about 10% of the time in the dataset, end in the slowest bucket every time. Additionally, with the *pageSizeLeaf* set to 2048, the executions mostly end up in the slowest two buckets, which considering that it is the lowest value in the dataset further underlines the findings from the previous section.

antecedents	consequents	antecedent support	support	confidence	lift
'payload_size_256'	'bucket_3'	0.0973	0.0973	1.0000	2.9995
'payload_size_4', 'const_pageSizeLeaf_8192'	'bucket_1'	0.0500	0.0500	1.0000	2.9995
'payload_size_64', 'const_pageSizeLeaf_2048'	'bucket_3'	0.0544	0.0544	1.0000	2.9995
'payload_size_4', 'const_pageSizeLeaf_4096'	'bucket_1'	0.0521	0.0518	0.9933	2.9794
'payload_size_64', 'const_pageSizeLeaf_8192'	'bucket_2'	0.0525	0.0520	0.9900	2.9711
'payload_size_128'	'bucket_3'	0.1520	0.1378	0.9060	2.7174
'payload_size_4'	'bucket_1'	0.1555	0.1269	0.8161	2.4480
'payload_size_8'	'bucket_1'	0.1430	0.1086	0.7598	2.2789
'payload_size_32'	'bucket_2'	0.1480	0.0987	0.6667	2.0007
'payload_size_64'	'bucket_2'	0.1557	0.0832	0.5342	1.6030
'data_size_1000000'	'bucket_1'	0.2084	0.1088	0.5222	1.5663
'payload_size_16'	'bucket_2'	0.1484	0.0743	0.5006	1.5023
'payload_size_16'	'bucket_1'	0.1484	0.0741	0.4994	1.4980
'payload_size_64'	'bucket_3'	0.1557	0.0720	0.4625	1.3872
'const_pageSizeLeaf_8192'	'bucket_1'	0.3541	0.1557	0.4397	1.3188
'const_pageSizeLeaf_2048'	'bucket_3'	0.2983	0.1303	0.4366	1.3095
'const_pageSizeLeaf_2048'	'bucket_2'	0.2983	0.1254	0.4202	1.2611

Figure 6.8: Association rules for *ycsb_e*

ycsb_c_init: By looking at the figure 6.9 one can observe a similar pattern, where the size of the payload and data can give a big indication, as to which of the three performance buckets the record belongs to. The pattern being that a high payload size indicates a slower performance.

antecedents	consequents	antecedent support	support	confidence	lift
'payload_size_256'	'bucket_3'	0.0979	0.0979	1.0000	3.0006
'payload_size_128'	'bucket_3'	0.1483	0.1441	0.9715	2.9152
'payload_size_4'	'bucket_1'	0.1529	0.1197	0.7827	2.3465
'payload_size_32'	'bucket_2'	0.1465	0.1136	0.7752	2.3265
'payload_size_8'	'bucket_1'	0.1518	0.1051	0.6923	2.0756
'data_size_1000000'	'bucket_1'	0.1695	0.1121	0.6615	1.9831
'payload_size_64'	'bucket_3'	0.1478	0.0877	0.5938	1.7818
'payload_size_16'	'bucket_2'	0.1548	0.0858	0.5544	1.6638
'data_size_2000000'	'bucket_1'	0.1671	0.0820	0.4908	1.4714
'data_size_6000000'	'bucket_2'	0.1662	0.0759	0.4569	1.3713
'data_size_5000000'	'bucket_2'	0.1643	0.0747	0.4547	1.3646
'const_pageSizeLeaf_2048'	'bucket_1'	0.3007	0.1353	0.4499	1.3487
'payload_size_16'	'bucket_1'	0.1548	0.0683	0.4415	1.3235
'data_size_4000000'	'bucket_2'	0.1638	0.0675	0.4123	1.2375

Figure 6.9: Association rules for *ycsb_c_init*

ycsb_e_init: The same observation as for operation *ycsb_c_init* can be made for *ycsb_e_init*. This is understandable, considering that these operations are very similar. The differences in values can be explained by the different scaling of feature *time*.

antecedents	consequents	antecedent support	support	confidence	lift
'payload_size_256'	'bucket_3'	0.0992	0.0992	1.0000	3.0037
'payload_size_128'	'bucket_3'	0.1527	0.1268	0.8305	2.4945
'data_size_1000000'	'bucket_1'	0.2076	0.1538	0.7406	2.2152
'payload_size_4'	'bucket_1'	0.1550	0.0990	0.6390	1.9114
'data_size_2000000'	'bucket_1'	0.1904	0.1095	0.5748	1.7194
'payload_size_8'	'bucket_1'	0.1425	0.0808	0.5671	1.6962
'data_size_4000000'	'bucket_2'	0.1979	0.1015	0.5127	1.5409
'payload_size_64'	'bucket_3'	0.1552	0.0791	0.5095	1.5304
'data_size_5000000'	'bucket_3'	0.2012	0.1018	0.5060	1.5200
'payload_size_32'	'bucket_2'	0.1475	0.0742	0.5029	1.5115
'payload_size_16'	'bucket_2'	0.1479	0.0744	0.5029	1.5114

Figure 6.10: Association rules for *ycsb_e_init*

6.3.2 Optimal page size of leaf nodes

The methodology to generate heatmaps, based on regression models trained by the more discrete dataset, proved to be difficult. The output of the regression models, Random Forest and Gradient Boosting, as previous findings from the association rules indicate, greatly depends on the features *payload_size* and *data_size*. As a result, the various page sizes of leaf nodes received the same output based on features *payload_size* and *data_size*. Therefore, this methodology was not able to identify the optimal page size of leaf nodes.

Nevertheless, the heatmaps based on empirical means are generated and visualized in figure 6.11.

The findings of operation *ycsb_e* indicate the high importance of big page sizes for scan operations. The page size 8192 is the highest in the data set, meaning that potentially a higher page size might be optimal. Operations ycsb_c_init and ycsb_e_init

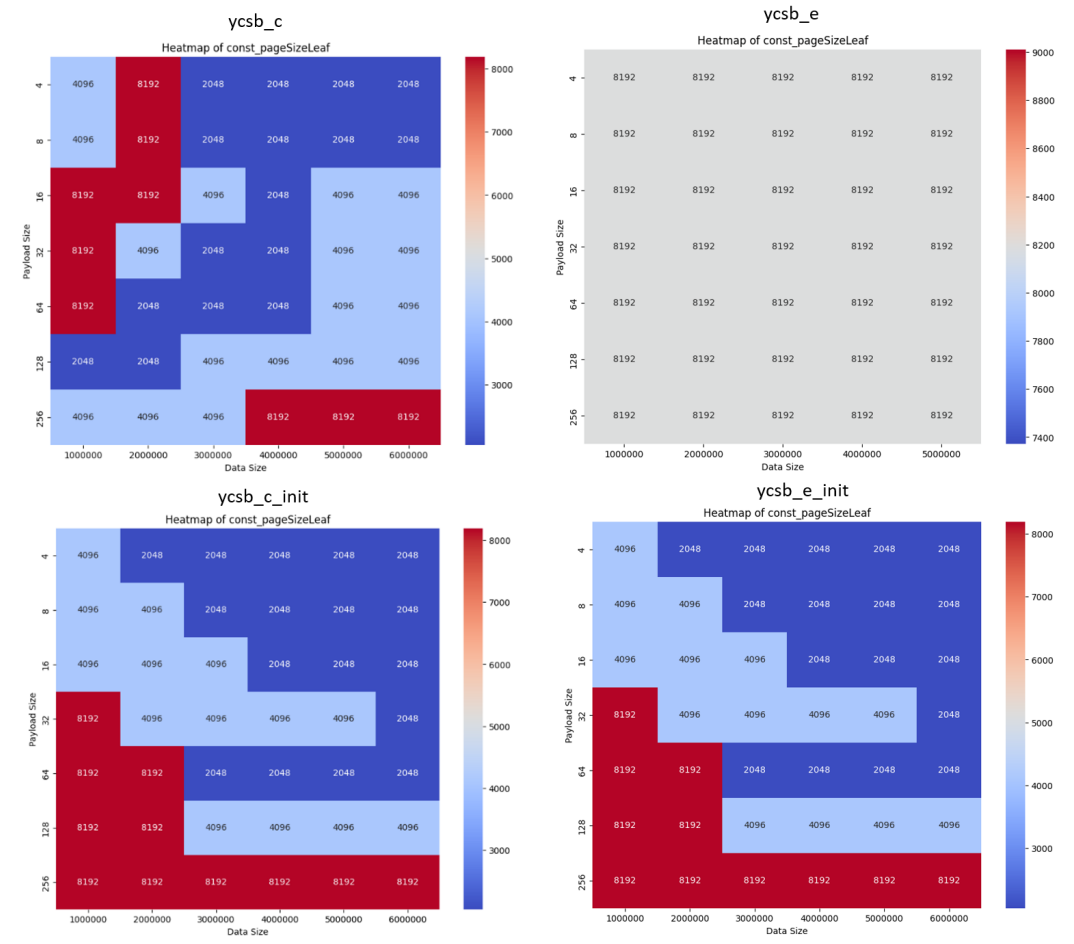


Figure 6.11: Heatmaps per operation based on empirical mean

generated the same behavior, which is expected due to their similar nature. The pattern seems to indicate that during the insertion operations, higher sizes of the payload prefer higher page sizes, however, the bigger the size of the dataset, lower page sizes are preferred. These findings should be interpreted with caution, as the optimal page sizes generally result in only an $1-3\%$ improvement compared to the next best page size. Given that the values of the feature *time* are in the order of magnitude of hundreds of nanoseconds (10^{-7} seconds), an $1-3\%$ improvement may not be considered substantial. Consequently, the heatmap for operation *ycsb_c* is not further analyzed.

7 Conclusion and Future Work

This thesis explored the use of ML models to analyze behavior of B+ trees with different optimizations and configurations. The aim of this thesis was to find insights into how one can improve the performance of B+ trees.

The methodology used in this thesis explored multiple ways of getting the desired insights. Firstly, two datasets were created, by running a benchmarking program many times with a varying B+ tree configurations, and measuring the performance of each run. These datasets were then preprocessed to fit the desired structures of the used algorithms.

The initial, statistical analysis brought meaningful insights into which features can and can not be used for trying to identify patterns between inputs and outputs.

The main approach of this thesis was to train classification models on the dataset and identify, using XAI on the best performing model, which inputs, impact the performance and how. The best performing model for the dataset was a neural network classifier, which by using on SHAP summary plots indicated the most influential parameters for performance. The skewness of the data together with the size of the data seem to impact the performance the most. For inserting and scan operations, the page size of leaf nodes can impact the performance as well. This indicates that an implementation of adaptive B+ trees, could potentially further improve performance of the data structure.

The second approach was to use regression models, which managed to get good results in terms of their R^2 metrics. However, using XAI algorithms on the models and getting insights from those proved to be mode difficult due to the scale of the feature *time*. Nevertheless, some insights gained from the first approach, were further supported.

Another approach was to use data mining pattern recognition techniques, to find patterns in the data. The association rule mining algorithm was used on a dataset, which has more discrete feature values, with fewer variations. This approach ultimately confirmed the claim that a bigger size of payload or data, impacts the performance negatively. Retrospectively, one could generate the dataset by not varying these parameters, and instead focus more on varying the page size of the leaf node and other optimizations. This way, one could identify more relevant patterns for an adaptive B+ tree.

Lastly, it was attempted to take train regression models on the dataset for pattern recognition techniques and generate heatmaps to create an overview of which payload and data size combination benefit from which page size of leaf nodes the most. This approach, however, failed due to the models not assigning enough importance to the page size. Nevertheless, a heatmap based on the empirical means and not the regression models was generated and discussed.

One key observation, outside the scope of B+ trees, is that using XAI on ML models is a viable way to gain valuable insights into a dataset.

In future works, this thesis could be expanded to use other types of data, instead of restricting itself to urls. Additionally, more parameters could be varied, and a bigger dataset created, to yield more confident results. Using the approaches demonstrated in this thesis, one could potentially identify more helpful patterns.

Abbreviations

XAI Explainable Artificial Intelligence

ML Machine Learning

YCSB Yahoo! Cloud Serving Benchmark

List of Figures

List of Tables

Bibliography

[1] M. Müller, L. Benson, and V. Leis, "B-trees are back: Engineering fast and pageable node layouts [experiment, analysis benchmark]," unpublished.

[2] T. H. Davenport and R. Bean, *Five key trends in ai and data science for 2024*, https://sloanreview.mit.edu/article/five-key-trends-in-ai-and-data-science-for-2024/, (Accessed on 03/10/2024), Sep. 2024.

[3] V. Stodden, "The data science life cycle: A disciplined approach to advancing data science as a science," *Communications of the ACM*, vol. 63, pp. 58–66, Jun. 2020. DOI: 10.1145/3360646.

[4] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control - SIGFIDET '70*, Boeing Scientific Research Laboratories, Jul. 1970, p. 107.

[5] G. Graefe, "Modern b-tree techniques," *Foundations and Trends in Databases*, vol. 3, no. 4, pp. 203–402, 2010. DOI: 10.1561/1900000028.

[6] R. Horvick, *B-tree - data structures succinctly part 2*, https://www.syncfusion.com/succinctly-free-ebooks/datastructurespart2/b-tree, (Accessed on 02/16/2024), Jul. 2014.

[7] Grundprinzip, *Example of a simple b+ tree with 7 key entries pointing to data d1 to d7.* CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=10758840, Jun. 2010.

[8] *Poetry - python dependency management and packaging made easy*, https://python-poetry.org/docs/, (Accessed on 02/23/2024).

[9] *Nohup(1) - linux manual page*, https://man7.org/linux/man-pages/man1/nohup.1.html, (Accessed on 02/23/2024).

[10] *Project jupyter | home*, https://jupyter.org/, (Accessed on 02/23/2024).

[11] *Pandas documentation — pandas 2.2.0 documentation*, https://pandas.pydata.org/docs/, (Accessed on 02/23/2024).

[12] *Scikit-learn: Machine learning in python — scikit-learn 1.4.1 documentation*, https://scikit-learn.org/stable/, (Accessed on 02/23/2024).

[13] *Mlxtend*, https://rasbt.github.io/mlxtend/, (Accessed on 03/11/2024).

[14] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer New York, NY, 2006, ISBN: 978-0-387-31073-2.

[15] K. P. Murphy, *Machine Learning: a Probabilistic Perspective*. MIT Press, 2012.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC'10)*, Indianapolis, Indiana, USA: Association for Computing Machinery, Jun. 2010, pp. 143–154, ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807152.

[17] N. J. Salkind, *Statistics for People Who (Think They) Hate Statistics*, 4th. Thousand Oaks, California, USA: Sage Publications, 2011, ISBN: 9781412979597.

[18] A. Papoulis and S. U. Pillai, *Probability, Random Variables, and Stochastic Processes*, 4th, international ed. Boston, USA: McGraw-Hill, 2002, pp. 250–251, ISBN: 9780072817256.

[19] V. R. Joseph, "Optimal ratio for data splitting," *Stat. Anal. Data Min.: ASA Data Sci. J.*, vol. 15, pp. 531–538, 2022. DOI: 10.1002/sam.11583.

[20] *Sklearn.metrics.accuracy_score — scikit-learn 1.4.1 documentation*, https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html, (Accessed on 02/29/2024).

[21] Z.-H. Zhou, *Machine Learning*, 1st ed. Springer Singapore, 2021, ISBN: 978-981-15-1967-3. DOI: 10.1007/978-981-15-1967-3.

[22] *Sklearn.tree.decisiontreeclassifier — scikit-learn 1.4.1 documentation*, https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html, (Accessed on 03/01/2024).

[23] *Sklearn.ensemble.randomforestclassifier — scikit-learn 1.4.1 documentation*, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html, (Accessed on 03/02/2024).

[24] T. Masui, *All you need to know about gradient boosting algorithm - part 1. regression | by tomonori masui | towards data science*, https://towardsdatascience.com/all-you-need-to-know-about-gradient-boosting-algorithm-part-1-regression-2520a34a502, (Accessed on 03/03/2024).

[25] *Sklearn.ensemble.gradientboostingclassifier — scikit-learn 1.4.1 documentation*, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html, (Accessed on 03/03/2024).

[26] T. Zhang, W. Lin, A. Vogelmann, M. Zhang, S. Xie, Y. Qin, and J.-C. Golaz, "Improving convection trigger functions in deep convective parameterization schemes using machine learning," *Journal of Advances in Modeling Earth Systems*, vol. 13, May 2021. DOI: 10.1029/2020MS002365.

[27] *1.17. neural network models (supervised) — scikit-learn 1.4.1 documentation*, https://scikit-learn.org/stable/modules/neural_networks_supervised.html, (Accessed on 03/02/2024).

[28] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG].

[29] *Sklearn.preprocessing.standardscaler — scikit-learn 1.4.1 documentation*, https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html, (Accessed on 03/02/2024).

[30] *Logistic regression in machine learning - javatpoint*, https://www.javatpoint.com/logistic-regression-in-machine-learning, (Accessed on 03/02/2024).

[31] *Cross validation in machine learning - geeksforgeeks*, https://www.geeksforgeeks.org/cross-validation-machine-learning/, (Accessed on 03/03/2024).

[32] *Sklearn.linear_model.logisticregressioncv — scikit-learn 1.4.1 documentation*, https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html, (Accessed on 03/03/2024).

[33] *Sklearn.metrics.r2_score — scikit-learn 1.4.1 documentation*, https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html, (Accessed on 03/03/2024).

[34] V. Alto, *Understanding ordinary least squares (ols) regression | built in*, https://builtin.com/data-science/ols-regression, (Accessed on 03/03/2024).

[35] *1.1. linear models — scikit-learn 1.4.1 documentation*, https://scikit-learn.org/stable/modules/linear_model.html, (Accessed on 03/03/2024).

[36] *Sklearn.ensemble.randomforestregressor — scikit-learn 1.4.1 documentation*, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html, (Accessed on 03/03/2024).

[37] *1.10. decision trees — scikit-learn 1.4.1 documentation*, https://scikit-learn.org/stable/modules/tree.html, (Accessed on 03/03/2024).

[38] *Frequent pattern mining in data mining - geeksforgeeks*, https://www.geeksforgeeks.org/frequent-pattern-mining-in-data-mining/, (Accessed on 03/04/2024).

[39] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994.

[40] *Apriori - mlxtend*, https://rasbt.github.io/mlxtend/user_guide/frequent_patterns/apriori/, (Accessed on 03/04/2024).

[41] *Association rules - mlxtend*, https://rasbt.github.io/mlxtend/user_guide/frequent_patterns/association_rules/, (Accessed on 03/03/2024).

[42] C. Molnar, *Interpretable Machine Learning, A Guide for Making Black Box Models Explainable*, 2nd ed. 2022.

[43] G. Stiglic, P. Kocbek, N. Fijacko, M. Zitnik, K. Verbert, and L. Cilar, "Interpretability of machine learning-based prediction models in healthcare," *WIREs Data Mining and Knowledge Discovery*, vol. 10, no. 5, e1379, 2020. DOI: https://doi.org/10.1002/widm.1379. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1379.

[44] *4.2. permutation feature importance — scikit-learn 1.4.1 documentation*, `https://scikit-learn.org/stable/modules/permutation_importance.html`, (Accessed on 03/04/2024).

[45] *Shap.kernelexplainer — shap latest documentation*, `https://shap-lrjball.readthedocs.io/en/latest/generated/shap.KernelExplainer.html`, (Accessed on 03/05/2024).

[46] *Bar plot — shap latest documentation*, `https://shap.readthedocs.io/en/latest/example_notebooks/api_examples/plots/bar.html`, (Accessed on 03/05/2024).

[47] C. Claesen, A. Rafique, D. Van Landuyt, and W. Joosen, "A ycsb workload for benchmarking hotspot object behaviour in nosql databases," in *Performance Evaluation and Benchmarking*, R. Nambiar and M. Poess, Eds., Cham: Springer International Publishing, 2022, pp. 1–16, ISBN: 978-3-030-94437-7.