technische universität
dortmund

24.11.
Gutachten

Master's Thesis

Automated Generation of Evaluation Tasks for Machine Learning Models

Fabian Dillkötter
September 2022

Supervisors:
Prof. Dr. Katharina Morik
M. Sc. Sascha Mücke

TU Dortmund University
Faculty of Computer Science
Chair VIII Artificial Intelligence
https://www-ai.cs.uni-dortmund.de

Contents

1 Introduction **1**

1.1 Goal 1

1.2 Related Works 2

1.3 Structure of this Work 3

2 Background **5**

2.1 SVM Models 5

2.2 Decision Trees 7

2.3 Confusion Matrix 10

3 Conceptual Framework for Test Generation **13**

3.1 Workflow of the Framework 14

3.2 Evaluation Tasks 15

3.3 Database 15

3.4 Assembly of Evaluations 16

3.5 Result Management 17

4 Implemented Framework **19**

4.1 MongoDB 19

4.2 MLflow 22

4.2.1 Elements of MLflow 23

4.2.2 Application in this Work 24

4.3 Task Generation 24

4.3.1 Input Processing 24

4.3.2 Assignment of Possible Tasks 32

4.3.3 Compilation of Task Packets 33

4.4 Evaluation Tasks 35

4.4.1 File Structure 35

4.4.2 Program Structure 36

5 Implemented Tasks **37**
5.1 Confusion Matrix Evaluation 37
5.2 SVM Robustness Evaluation 39
5.2.1 Description of the Tool 40
5.2.2 Implemented Software 41
5.3 Theoretical Bounds for SVMs 42
5.3.1 Theoretic Background 43
5.3.2 Implementation of the Calculations 45
5.3.3 Generation of a Report 46
5.4 Energy Consumption Measurement 48
5.4.1 Description of the Tool 49
5.4.2 Integration as a Task 50
5.4.3 Outputs and Interpretation 53

6 Evaluation **57**
6.1 Test Criteria 57
6.2 Experiment Composition 59
6.3 Experiment 1: SVC on MNIST 61
6.3.1 Confusion Matrix Evaluation 61
6.3.2 Robustness Evaluation 61
6.3.3 Energy Consumption Evaluation 62
6.4 Experiment 2: SVR on California Housing 62
6.4.1 Energy Consumption Evaluation 63
6.5 Experiment 3: Linear SVC on Ionosphere 63
6.5.1 Confusion Matrix Evaluation 64
6.5.2 Robustness Evaluation 64
6.5.3 Energy Consumption Evaluation 65
6.5.4 Bounds Report Evaluation 65
6.6 Experiment 4: Decision Tree on Iris 65
6.6.1 Confusion Matrix Evaluation 66
6.6.2 Energy Consumption Evaluation 67
6.7 Assessment of the Implementations 67
6.7.1 Achieved Goals 68
6.7.2 Software Quality 69

7 Discussion **73**
7.1 Results 73
7.2 Future Work 74

Table of Figures **77**

Bibliography **79**

A Outputs of the Evaluations **91**

A.1 RBF SVC on MNIST 91

A.2 RBF SVR on California Housing 99

A.3 Linear SVC on Ionosphere Dataset 100

A.4 Decision Tree on Iris 105

Chapter 1

Introduction

In the past years, Machine Learning (ML) has become omnipresent in both scientific and industrial applications. Due to this development, people of different backgrounds are presented with the task of evaluating ML models. Scientists, developers and managing employees have an interest in testing models for metrics like accuracy, robustness or energy consumption [99].
Possible situations where such an evaluation is needed include the assessment of models other researchers have developed in a scientific environment. When researchers are presented with foreign ML models, execution of independent tests can help to acquire an unbiased rating. Other scenarios are the comparison of models during development and the evaluation of a model during the decision making whether a model shall be deployed. These different groups all desire a fast method for evaluating ML models, however the conventional workflow is currently not optimized. The process for testing aspects of a model is complex and includes multiple challenges.
The first challenge in evaluating ML models is finding tests that are applicable to the model type, dataset and specific model in question. The resulting tests may not be directly executable and require implementation in order to work on the supplied model. Afterwards, the tests have to be executed manually and individually. Only after combining the outputs of different tests with multiple parameter variations, the user can acquire a meaningful assessment.
This process requires expert knowledge of the used tools and methods and is neither time nor resource efficient. The users desire a simple tool for the evaluation of diverse models in regards to multiple metrics.

1.1 Goal

The main goal of this work is the development of a framework that enables an automated evaluation of ML models. The aim however is not a fully developed universally applica-

ble framework, but rather a baseline on which future additions and improvements can be made. In order to allow a meaningful determination of features, the limitations of the implementation have to be declared from the beginning.
The framework must allow users to input ML models, datasets and additional information about the desired evaluations. Furthermore, the framework must contain a database for storage of evaluation techniques and specific details of their execution. These two sources of information have to be combined by the framework to result in the generation of tests from the database that fit the inputs of the user. These tests must then be independently runnable to analyse different metrics. The outputs of this framework are then stored in a central hub so they could then be used to construct a rating of the model, for example a Care Label [72].
As discussed before, the development of a complete framework with full functionality for all possible models is not feasible in a single work. Due to this reason, formulating some limitations is unavoidable. The first strong element of this is the restriction to Support Vector Machine (SVM) models. While a Decision Tree based ML model is used as a comparison, this is not the focus of this work. Another limitation is the amount of supported evaluation techniques, which is not exhaustive, as the starting set of evaluations can be expanded upon in the future.
The goal of this work is centered around the structure of an evaluation framework for ML models and not the specific implementation of tests. Therefore the elements of the framework must be detailed and explained.
As a fully implemented framework for evaluating all possible models in a variety of aspects is beyond the scope of this work, the focus is rather on the development of a baseline for future expansion. Therefore the structure of the framework and the conceptualization of a workflow is the focus of this thesis, and the main research question is:
What elements are needed for an automated generation of Evaluation Tasks for ML models?

1.2 Related Works

As the evaluation of ML models constitutes a highly relevant research topic, multiple works with a focus on automated execution exist. Automating evaluation of software in general is an established research area. Even the automated evaluation of high level features like the usability of user interfaces have already been explored over 20 years ago [48]. In the context of ML models, the evaluation of models takes a special role, as it is closely connected to the tuning of hyperparameters.
Prior work has been done to automate the process of hyperparameter tuning for example with the Waikato Environment for Knowledge Analysis (WEKA) [44]. While an automated hyperparameter tuning is only possible with automated evaluation, the focus in software

like WEKA is not on providing a comprehensive evaluation and rather on testing fast measures (like accuracy) for comparing different configurations. Furthermore, this evaluation is not applicable to models trained using other frameworks.
Some tools have been developed that offer support of models trained using multiple training frameworks [64, 75, 83]. The frameworks that allow a fully automated testing are however all limited in some way.
The CleverHans library [75] is restrained to evaluating robustness using adversarial examples. Similarly, the Foolbox library [83] is also focused only on evaluating robustness. Other works include the SHapley Additive exPlanations (SHAP) framework for the identification of the relevant features for predictions [64].
The software developed in these works falls short of the goals formulated in this thesis, as only a limited set of evaluation metrics aimed at a specific aspect of the models can be evaluated. In contrast, this thesis aims to create a framework capable of assigning evaluation methods of arbitrary variety to given models.
One part of an automated processing of ML models is the development of a unified model format. The framework ONNX[1] offers a format for ML models designed specifically to facilitate the connection and interoperability of different tools and frameworks. This development is needed because popular existing frameworks often only support exporting and importing models as JavaScript Object Notation (JSON) [29] or using the python object serialization pickle [76].
The development of a framework that allows the management of all parts of the model lifecycle has been approached with the software MLflow [98]. This product along with the subsequent improvements of Chen et al. [28] can be useful with any desired framework and will therefore be used in this work (see chapter 4.2).
The evaluation of ML models is closely connected to the research topic of Explainable Artificial Intelligence (XAI) [85]. ML methods are evaluated for different aspects like accountability [8], responsibility [33] or robustness [92] with the goal of a more transparent learning process and more explainable decisions.
The combination of different metrics to produce an easily interpretable rating has also been explored, for example with the Care Label concept [72]. This concept produces certifications for ML models according to a set of aspects. The framework developed in this work could be used to evaluate the measures needed for such a certification.

1.3 Structure of this Work

First of all, the ML techniques SVM and Decision Tree are introduced with some mathematical background in chapter 2. This is done to provide a basis to apply the framework to.

[1] `https://onnx.ai`

In the following chapter, the concept of a framework that enables an automated evaluation of ML models is introduced. The workflow of a framework like this is described and the necessary database technology discussed. The basic elements of such a framework are defined.
Chapter 4 then shows the structure and workings of the developed framework in detail. This topic is split into sections about the set up database, the usage of the framework MLflow[2], the process of generating tasks and the conceptualized structure for Evaluation Tasks that are added to the framework.
After the workings of the framework are introduced, the implemented Evaluation Tasks are then presented in chapter 5. These are described individually to give an overview on how evaluation metrics can be added to the framework. For each task, the different measures and tools are introduced and the specifics of their implementation are explained.
In chapter 6, an evaluation of the implementations is given. The goals are evaluated using different models that are trained for various tasks. Furthermore, the chapter refers back to the goals from section 1.1 and highlights in what ways they are achieved.
Lastly, chapter 7 gives a summary of the results and an outlook on future modifications and additions.

[2] `https://www.mlflow.org/`

Chapter 2

Background

The basic types of models that the framework is applied to are explained in the following sections. After giving an overview of the basic principles behind SVM models, Decision Trees are introduced in a short section. As the created Evaluation Tasks are geared towards SVMs and the Decision Tree models are only a comparative choice, Decision Tree models are only described briefly.
After this, an introduction to confusion matrices as a statistical analysis is given. This section defines and describes mathematical background needed for later parts of this work.

2.1 SVM Models

The amount of different ML techniques is increasing at a rapid pace [73]. Due to this development, the evaluation of these differing models cannot be performed with the same algorithms. Methods like Artificial Neural Networks, Decision Trees or SVMs have to be evaluated using separate methods [99]. As such, the selection of a single ML method is needed to evaluate and develop the framework as a first start. The ML methods that are used as the exemplary cases in this work are SVMs.
The concept of separating examples using a hyperplane was used early on in mathematical research [38]. The use of SVMs as a machine learning implementation of this concept was then developed and popularized in the 1990s [14].
SVMs are chosen to be the starting point due to the mathematical and statistical theory they are built upon. The following explanation of the workings of basic maximum margin SVMs is based closely on the work from Boser, Guyon and Vapnik [14], which introduced non-linear classification using the kernel trick.
The training task for the maximum margin algorithm (for binary classification) is to find a function which classifies examples in the form of feature vectors x of dimension n into classes A and B. The training dataset therefore has the form:

$$(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$$

It consists of n examples x_i with labels y_i, where the following labelling is used:

$$y_i = \begin{cases} 1 & \text{if} \quad x_i \in A. \\ -1 & \text{if} \quad x_i \in B. \end{cases}$$

During the training, the parameters of a function $D(x)$ are then optimized to acquire predictions of the form:

$$x \in A \quad if\ D(x) > 0$$
$$x \in B \quad otherwise$$

The decision function with the to be adjusted parameters α_k, training patterns x_k and bias b then takes this form in dual space:

$$D(x) = \sum_{k=1}^{n} \alpha_k K(x_k, x) + b$$

The function K takes a special role, as it represents a predefined kernel. These kernels can take many different forms, like linear kernels [54], polynomial kernels [3], kernels based on Radial Basis Functions (RBF) [91] or kernels using sigmoid functions [60]. The choice of kernel significantly affects the behaviour of the SVM [77]. The kernel functions transform the data from the input space into a higher dimensional feature space. This enables a linear separation in the feature space using a hyperplane, even if a linear separation is not possible in the input space. The "kernel trick" was first introduced in 1964 [2] and avoids the problem of learning a nonlinear function by the ML method. As proposed by Aronszajn [5], valid kernels can be written as:

$$K(x, x') = \sum_{i} g_i(x) g_i(x')$$

In this formula, based on Mercer's theorem [68], g is any function in the Hilbert space. In the basic case, the construction of a maximum margin hyperplane is the next step towards computing the decision function. This hyperplane is defined solely by its support vectors, which are the examples on the margin boundaries. In case an SVM is constructed that separates the classes without misclassifications, the hyperplane is called hard margin.
In contrast to this, a soft margin SVM introduces slack variables ξ_i which take a value greater than 1 if the training example x_i is on the wrong side of the analysed hyperplane. This concept is outlined following the definitions of Joachims [53]. The optimization problem for linear soft margin SVMs then takes the following form in primal space:

2.1.1 Optimization Problem (Linear soft margin SVM (primal)).

$$\begin{aligned} minimize: \quad & V(\vec{w}, b, \vec{\xi}) = \tfrac{1}{2}\vec{w} \cdot \vec{w} + C \sum_{i=1}^{n} \xi_i \\ subject\ to: \quad & \forall_{i=1}^{n}: \ y_i[\vec{w} \cdot \vec{x_i} + b] \geq 1 - \xi_i \\ & \forall_{i=1}^{n}: \ \xi_i > 0 \end{aligned}$$

Here the vector $\vec{w}$ is the weight vector and C is a parameter used to influence the trade off between training errors and model complexity. A low value for C puts a higher priority on a low model complexity whereas a high value lowers the amount of training errors.
In dual space the optimization problem is:

2.1.2 Optimization Problem (Linear soft margin SVM (dual)).

$$\begin{aligned} minimize: \quad & W(\vec{\alpha}) = -\sum_{i=1}^{n} \alpha_i \ + \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i y_i \alpha_j y_j (\vec{x_i} \cdot \vec{x_j}) \\ subject\ to: \quad & \sum_{i=1}^{n} \alpha_i y_i = 0 \\ & \forall i \in [i..n]: \ 0 \leq \alpha_i \leq C \end{aligned}$$

Apart from the variable C bounding the values of α_i this optimization problem is identical to that of the hard margin SVM in dual space. Now the meaning of the α_i Lagrange multipliers becomes obvious; the set of support vectors consists of the training examples x_i for which the corresponding $\alpha_i > 0$.
The development of soft margin SVMs was motivated by the desire to accept a number of misclassifications in order to achieve a larger margin size [31]. A new regularization parameter C is introduced to control the trade-off between the number of misclassifications and the size of the margin. The resulting optimization problem for computing the support vectors can be solved as a quadratic programming process. Other methods for calculating the SVM classifier have been explored, like coordinated descent [47], sub-gradient descent [87] or cutting-plane [54] techniques. As the exact inner workings of these algorithms are not relevant for this work, they will not be discussed in more detail.
Additionally to the described usage of support vectors for classification, the idea can also be applied to other learning tasks. Support Vector Clustering is an unsupervised learning algorithm where similar elements are grouped into classes that are not predetermined [9]. Another development is the Support Vector Regression (SVR), where the numeric label value is estimated using the features. This method has advantages on problems with high dimensional inputs [34].

2.2 Decision Trees

Decision Tree models can take different forms. In this thesis, the models known as Classification And Regression Trees (CART), first introduced by Breiman et al. [16], are chosen as the base type. This results in binary splits for every inner node of the tree. Furthermore, only classification trees will be discussed, because they are the selected ML model for a comparison to SVM model evaluation outputs.
As with SVM based classification, the problem definition stays the same. A training sample with n examples consisting of the feature vectors $x_1, ..., x_n$ and class variables $y_1, ..., y_n$

Figure 2.1: Example of a Decision Tree based on the Iris dataset

with values $1, ..., k$ is used to find a prediction model. The goal is to find a model that predicts values of y for new x values, thereby partitioning the X space into k disjoint sets $A_1, ..., A_k$ such that $y = z$ if the example x is in class A_z with $z \in \{1, ..., k\}$ [62].
The starting set and subsequently generated subsets are partitioned recursively using conditions on the values of the features of X. For singular trees with a concise length and comprehensible split decisions, Decision Trees can be human readable. An example for a simple Decision Tree based on the Iris dataset [38] is depicted in figure 2.1.
During training, an optimal interval for features is computed which minimizes the sum of the impurities of the two child nodes. The separating value is chosen from the values of this feature. In the case of nominal features, CART normally only allows for the selection of a single value for the variable, however some implementations support splits based on a subset of the set of values the feature can have. This process is recursively repeated on the child nodes. The learning algorithm stops when a stopping criterion is reached (e.g. a maximum tree depth). All elements present in a leaf node are then classified as the most common class in this node during training.
An important difference between Decision Tree implementations is the choice of the impurity function. While other implementations use splitting criteria like the information gain (C4.5 algorithm [81]) or elaborate processes based on significance tests and χ^2 tests (GUIDE algorithm [61]), the CART algorithm used in this work is based on the Gini index for impurity.
The Gini impurity for a node q in the case of equal misclassification costs for all classes is defined as:

$$Gini(q) = 1 - \sum_{j=0}^{k} \left(\frac{n(j \mid q)}{n(q)} \right)^2$$

In this formula k is the number of distinct classes, $n(j \mid q)$ represents the number of examples of class j in q and $n(q)$ describes the total number of examples in q. To analyse the impurity after a specific split, the weighted average can be calculated:

$$Gini(q)_{split} = \frac{n(q_L)}{n(q)} Gini(q_L) + \frac{n(q_R)}{n(q)} Gini(q_R)$$

The choice of feature for the split is then straightforward. All possible splits are evaluated and the split with the lowest $Gini(q)_{split}$ impurity is selected [16].
After the creation of a Decision Tree T using the criteria above, the complexity is reduced. To limit the size and complexity (usually measured in the number of leaf nodes $|\tilde{T}|$), the cost-complexity $R_\beta(T)$ measure is introduced. Given a complexity parameter $\beta \geq 0$ denoting the penalty for each additional leaf node, the cost-complexity is defined as:

$$R_\beta(T) = R(T) + \beta|\tilde{T}|$$

A necessary part of the calculation is the cost of a Decision Tree T, denoted as $R(T)$. The cost of a Decision Tree is in turn calculated using the sum of the costs of the leaf nodes:

$$R(T) = \sum_{q \in \tilde{T}} R(q)$$

Using the set of priors π_j and the number of examples with class j in the training set n_j, the cost of a node can be calculated:

$$\begin{aligned} p(q) = &\sum_{j=1}^{k} \pi_j \frac{n(j \mid q)}{n_j} \\ r(q) = &\ 1 - \max_j \frac{\pi_j \frac{n(j \mid q)}{n_j}}{p(q)} \\ R(q) = &\ r(q)p(q) \end{aligned}$$

The measure $p(q)$ is also called the resubstitution estimate of the probability that any case falls into the node q. The resubstitution estimate of the probability of misclassification, given that a case falls into node q is given as $r(q)$.
The priors can be manually adjusted but are often chosen from the class prevalence in the training data as $\pi_j = \frac{n_j}{n}$. If chosen like this, the terms $[\pi_j \frac{n(j \mid q)}{n_j}]$ in the above formulas can be simplified to $[\frac{n(j \mid q)}{n}]$.
Using the cost-complexity $R_{beta}(T)$, subtrees are iteratively removed and replaced by a leaf node. This produces a series of Decision Trees, of which the optimal tree is chosen using cross-validation. The specific pruning and selection process exceeds the scope of this thesis, as Decision Trees are only a type of model used for comparison. The detailed pruning process can be explored in the original work [16].
Decision Trees can be used to construct more complex models. A common technique are Random Forests, which are based on the training of a set of uncorrelated Decision Trees relying on bootstrap aggregation and a randomized way of optimizing nodes [15].

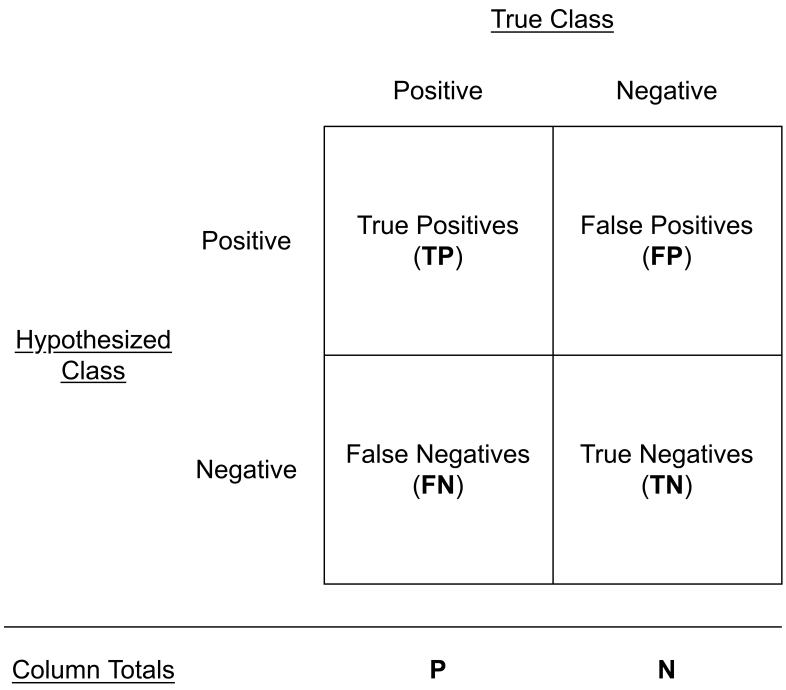

Figure 2.2: Structure of a confusion matrix for the binary case

2.3 Confusion Matrix

A confusion matrix shows the relation between the real class memberships and the classifications of the model [36]. While the matrix itself can be used to analyse the model, many further metrics are also based on the six measures that are evaluated for each class C_i:

1. Real Positive (P_i): The total number of cases in the dataset that are members of C_i
2. Real Negative (N_i): The total number of cases in the dataset that are members of a class other than C_i
3. True Positive (TP_i): The amount of cases that are correctly identified as members of C_i
4. False Positive (FP_i): The amount of cases that are incorrectly identified as members of C_i
5. True Negative (TN_i): The number of cases that are correctly identified as not being members of class C_i
6. False Negative (FN_i): The number of cases that are incorrectly identified as not being members of class C_i

In the case of binary classification, the index can be omitted, as a 2×2 matrix can express all the information about both classes (only needing a prior assignment of the two classes

as positive and negative). The layout of a confusion matrix for the binary classification case is shown in figure 2.2.
For a multi class model, the matrix is made up by one column and one row per class. The definitions of the base measures are however consistent irrespective of the number of classes.
As the total values may sometimes be misleading, the evaluated measures are often converted into ratios like the True Positive Ratio $TPR_i = \frac{TP_i}{P_i}$ or further metrics like the Accuracy $Acc = \frac{TP_i+TN_i}{P_i+N_i}$. A list of measures derived from the base values of a confusion matrix is given below (indices omitted for better clarity):

- Accuracy: $Acc = \frac{TP+TN}{TP+TN+FP+FN}$
- Sensitivity/Recall/True Positive Rate/Hit Rate: $TPR = \frac{TP}{P}$
- Specificity/True Negative Rate: $TNR = \frac{TN}{N}$
- Fallout/Alarm Rate/False Positive Rate: $FPR = 1 - TNR$
- Miss Rate/False Negative Rate: $FNR = 1 - TPR$
- Precision/Positive Predictive Value: $PPV = \frac{TP}{FP+TP}$
- Inverse Precision/Negative Predictive Value: $NPV = \frac{TN}{FN+TN}$
- False Discovery Rate: $FDR = 1 - PPV$
- False Omission Rate: $FOR = 1 - NPV$
- Positive Likelihood: $LR+ = \frac{TPR}{1-TNR}$
- Negative Likelihood: $LR- = \frac{1-TPR}{TNR}$
- Diagnostic Odds Ratio: $DOR = \frac{LR+}{LR-}$
- Youden's Index/Bookmaker Informedness: $BM = TPR + TNR - 1$
- Matthew's Correlation Coefficient: $MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$
- F_1-score: $F_1 = \frac{2 \times PPV \times TPR}{PPV+TPR}$
- Markedness: $MK = PPV + NPV - 1$
- Balanced Classification Rate/Balanced Accuracy: $BA = \frac{TPR+TNR}{2}$
- Jaccard/Thread Score: $Jaccard = \frac{TP}{TP+FP+FN}$

Many of the metrics have been proven to be vulnerable to class imbalances, for example PPV and NPV [90]. Therefore, an evaluation of the class prevalence $\frac{P}{P+N}$ is used to complement the interpretation of those metrics. Other measures are invariant to such imbalances, for example TPR and FPR [39]. Whether a metric is sensitive to class imbalance depends mostly on the columns of the confusion matrix used to calculate it [36].

Chapter 3

Conceptual Framework for Test Generation

In this chapter, a concept of a framework that allows an automated generation of tests for ML models is outlined. The basic workflow which a functional framework for the evaluation of ML models must provide is explained in the next section. Afterwards, the concept of Evaluation Tasks is introduced. As the storage of data and information makes up a big part of this work, further background is given on database technologies and their benefits for this application. Lastly, a draft of the evaluation assembly and result management is given.

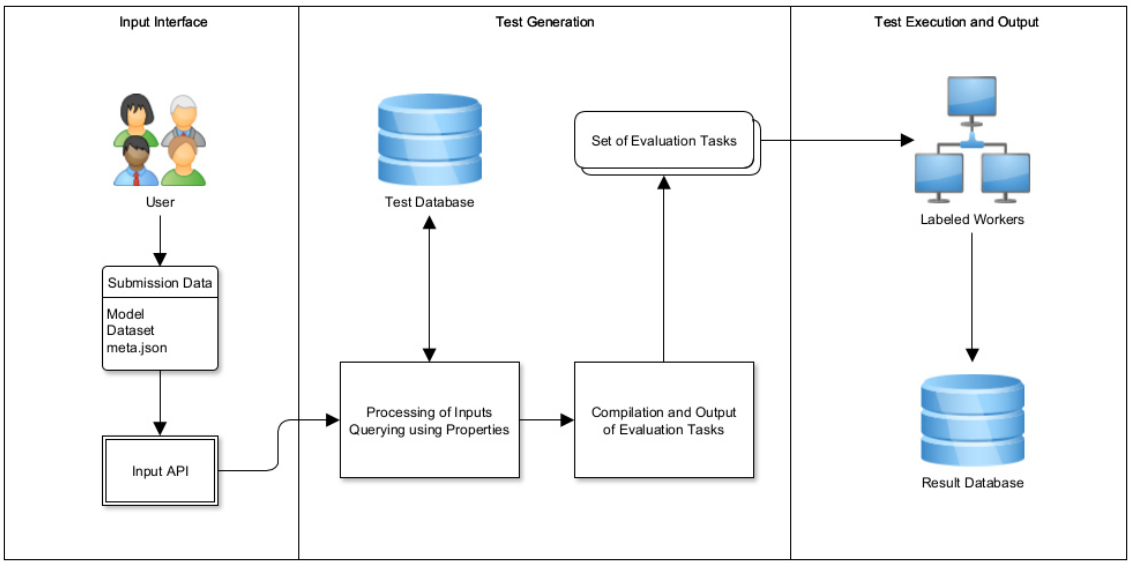

Figure 3.1: Full workflow of using the framework to evaluate a model

3.1 Workflow of the Framework

The workflow of the developed framework follows the visualization in figure 3.1. The focus of this thesis is on the process of test generation shown in the center of the flow chart.
In order to acquire executable evaluation software packets, the implemented methods of evaluation contained in the database (Evaluation Tasks) have to be applicable to different models. As not every evaluation technique can be applied to every model, the Evaluation Tasks have to be matched to the specific inputs given by the user. These inputs are mainly information about the model and the used dataset. The combination of the available information results in a set of constraints which reduce the number of applicable Evaluation Tasks.
The constraints are used to query the database of Evaluation Tasks, which results in the set of available tasks that can be carried out on the given inputs. The Evaluation Tasks are software packets which can be executed individually to assess different aspects of the model. Each task comes with requirements that have to be satisfied to enable their execution.
The Evaluation Tasks fetched from the database are combined with the inputs from the user to result in a set of independent pieces, which can evaluate different metrics and aspects of the model and dataset.
Running these tasks results in the evaluation of the model and dataset according to different metrics. The executed tasks deposit their results at a central result storage using an Application Programming Interface (API).
The user can use the obtained values to gain knowledge about the model and dataset. All the acquired information can then be combined and interpreted to result in an overall assessment like a Care Label [72].
In addition to the framework itself, some evaluation techniques need to be implemented to show the insertion process into the framework and for an evaluation of the framework itself. Some of the tests must be based on existing software to prove that an integration of established and proven tools is possible.
All in all, a concise summary of the elements that are to be developed in this work is the enumeration below:

1. Development of a framework for Evaluation Tasks
    (a) Creation of a database for evaluation software
    (b) Conceptualization of standardized formats and processes
    (c) Assignment of stored evaluation software to given inputs
    (d) Generation of individually executable Evaluation Task packets dependent on the given collection of software
    (e) Provision of a central platform for uploading, downloading and viewing evaluated outputs

2. Creation of an initial set of Evaluation tasks

   (a) Exemplary implementation of standalone evaluation software

   (b) Integration of existing evaluation tools

3.2 Evaluation Tasks

The Evaluation Tasks are the programs for evaluation of ML models that are integrated into the framework. These tasks can be focused on a single aspect or multiple combined tests. The different Evaluation Tasks can evaluate completely unrelated metrics, for example evaluations focused on robustness [23] or energy efficiency [59].
Another possible output is a certification of a model. This certification can be focused on different facets. The compliance with theoretical bounds [53] could be checked, or existing certifications, for example focused on robustness [19, 100] could be evaluated.
The setup of the Evaluation Tasks should follow a standard structure to give an identical method for execution, input transfer and output storage. To this end, a uniform entrypoint must be specified, however differing programs and files may be included into tasks beyond this starting point for the execution.
The Evaluation Tasks must come with specified requirements for their applicability. These may contain specific model properties, data structures or attributes of the executing system. This data is needed because evaluation algorithms are always geared towards specific models, ML task or data. On top of this, some evaluations are based on the execution on a specific type of Central Processing Unit (CPU) [30]. The introduction of other hardware for ML processes must also be specified, as some specially developed hardware offers unique programming and optimization opportunities [65].
Additionally, the tasks need to contain information for the configuration of the runtime environment. This information could be stored in a file in text-like format or by specifying the construction of the execution environment, for example using an environment managing system or containerization [97].

3.3 Database

In order to store information about Evaluation Tasks and the corresponding executable files, a database is needed. This database must provide the option to filter the contained data according to queries.
The prevailing database technologies can be divided into relational and non-relational databases. Relational databases consist of tables where columns signify the different data categories and rows correspond to unique elements of data. The majority of relational databases today use Structured Query Language (SQL) for interaction with the data [67].

Non-relational databases are more diverse, offering options like key-value-databases (e.g. Redis[3]), graph databases (e.g. Titan[4]), wide-column databases (e.g. Hypertable[5]) and document databases (e.g. MongoDB[6]). These non-relational databases offer different advantages for various use-cases and are in general more flexible and open to distribution on local and remote servers [32].
Due to the high flexibility and the ability to store all related data in a single document without the need to do join-operations, a document database is chosen for this project. MongoDB in particular is chosen because the performance benefit in query duration compared to relational databases is already proven [37].
Furthermore MongoDB allows changes to the database schema at any time. This is a benefit for this work, as the goal is an easily expandable framework. Future changes to the data model are therefore facilitated by using MongoDB.
This database must provide the ability to filter the contained evaluation techniques according to a given set of constraints. To this end, requirements for the execution of the tests need to be stored alongside the actual tests.
To match the constraints to the contents in the database, the framework must be able to construct queries in a modular way that allows different input values to be included. The database in this work has to process these queries to result in the desired set of outputs.
A solution to this problem is the storage of different values in sets. If for example the database contains an entry with the attribute $"task" : ["regression","classification"]$ and the constraints dictate that only entries that are compatible with classification tasks may be returned, then the examination whether the entry of the database fits this criterion becomes the simple check $[Is\ "classification" \in "task"?]$, with the answer deciding on whether this entry should be returned for the query.

3.4 Assembly of Evaluations

For the assembly of the complete and suitable set of evaluations, the inputs of the user need to be processed. The submitted model, dataset and metadata have to be collected and processed. This processing must extract the properties needed to construct a set of constraints. These constraints are then the information needed to acquire the applicable Evaluation Tasks.
Some type of software is needed to interact with the database to construct the executable Evaluation Tasks. For this capability, an interface to the database is needed. This interface must be able to query the database with the constraints of the model and download all returned evaluations.

[3] `https://redis.io`
[4] `http://titan.thinkaurelius.com`
[5] `https://hypertable.org`
[6] `https://mongodb.org`

The assembly software then has to construct the individual Evaluation Tasks to be ready to be executed. This entails the passing of inputs to the individual deployed tasks. The outputs of this part of the framework then is a set of individual Evaluation Tasks that are independently executable and completed with the needed inputs to run them.

3.5 Result Management

The execution of evaluations can yield different results. The tests can output sets of metrics, data files or documents depending on the focus of the evaluation. All of these outputs must be combined at a single point to acquire an overview of the results.
A central storage for evaluation results is not only beneficial to the handling of the tests for the user, but also provides the opportunity to allow an interaction with the results using other programs. As such, an API for easy access to the results is desired.
Users may want to store the results on a network server, especially if the execution of evaluations is distributed on multiple systems. The location of the result storage must therefore be customizable by the user. As a direct modification of the Evaluation Tasks by the user before execution is not intended, the information about the result storage must be input to the generation framework, which must pass the information to the individual Evaluation Tasks.
The optimal result management system not only provides an API for the access to the results for other software but also offers some type of interface for the user to view and compare results. This platform must be able to handle all types of result outputs, including documents for generated reports or certifications.

Chapter 4

Implemented Framework

The main software product of this work is the implemented framework. As per the goals formulated in chapter 1.1, the implemented framework must allow for an Expandable Generation of Evaluation Tasks, and is therefore referred to as ExGETa framework.
The first element of the created framework is the database where information about all implemented Evaluation Tasks and the software to execute these tasks is stored. This data storage is implemented using MongoDB.
The second part of the framework is the result storage. This element is implemented based on MLflow, so the section 4.2 describes the structure and use of MLflow.
The engine which matches the inputs to the tasks in the database and compiles independent packages for the execution is an integral part of the ExGETa framework. The implementation is shown in section 4.3.
Lastly, a structure for managing the evaluations and their outputs must be provided. These parts are explained in the following sections.

4.1 MongoDB

As explained in chapter 3.3, MongoDB is used as the database technology to store the code and information about available Evaluation Tasks. In order to allow an easy use of the framework on different machines, the database is set up as a cloud service hosted on the MongoDB Atlas[7] platform. This facilitates the setup of the database, offers an easier modification of infrastructure resources and eases the management of access control [84].
The framework only needs a single database with two collections. The collection `fs.chunks` is needed for storing the software of the Evaluation Tasks in a compressed ZIP container. The other collection (`fs.files`) stores metainformation about the tasks and which constraints need to be satisfied by the model and dataset for execution. This schema follows the specification of GridFS [80].

[7] `https://atlas.mongodb.com`

MongoDB works with JSON objects on the surface, which are transformed into Binary JSON (BSON) documents in the background by a binary-encoded serialization to store pairs of key (of type String) and value (any type including Arrays or documents).
As the size of BSON objects is limited to 4 MB in MongoDB applications, the specification GridFS is used for storing files exceeding this limit. GridFS specifies the use of a files collection to store metadata including the corresponding IDs of the chunk collection. The chunk collection stores the real data, where one or more chunks amount to the actual data to be stored [42]. The fields of both collections are explained in the following overview:

fs.chunks

- `chunks._id`
  - 12-byte BSON
  - Unique ObjectID of the chunk
- `chunks.files_id`
  - 12-byte BSON
  - Unique ObjectID of parent document in the `fs.files` collection
- `chunks.n`
  - 32-bit Integer
  - Sequence number of the chunk
  - Automatically generated starting at 0
- `chunks.data`
  - BSON document (binary)
  - The encoded payload of this chunk
  - Combined payloads of all connected chunks produce the stored file

fs.files

- `files._id`
  - 12-byte BSON
  - Unique ObjectID of the document
- `files.length`
  - 64-bit Integer
  - Size of the complete document in bytes

- `files.chunkSize`
  - 32-bit Integer
  - Size of each individual chunk in bytes (except for the last chunk)
- `files.uploadDate`
  - Date (internally 64-bit Integer)
  - Date of the first storage of the document
- `files.metadata`
  - Any datatype possible
  - Optional field
  - In our framework used to store constraints of the Evaluation Task for modeltype, method of execution and dataset limitations

The most important modification of the GridFS specification is the use of the `files.metadata` field to store the constraints of the Evaluation Tasks to enable filtering the documents by this property. The schema for these constraints is introduced now.

- "version": String
  - Version of the Evaluation Task in a yet unspecified format
  - Example: "0.1.1"
- "method": Array
  - ML algorithms that can be evaluated
  - Example: ["SVM", "DecisionTree"]
- "task": Array
  - ML tasks that can be evaluated with this technique
  - Example: ["classification", "regression"]
- "classes": Array
  - Amount of classes for a classification problem
  - Only exists for Evaluation Tasks with capabilities of testing classification models
  - Example: ["2", "multi"]

- "module": Array
  - The list of toolkits the models can be built from to be applicable
  - Example: ["sklearn.SVC", "sklearn.NuSVC"]
- "loader_module": Array
  - Supported storage formats for models
  - Example: ["sklearn.pickle", "sklearn.joblib"]
- "os": Array
  - Operating systems this Evaluation Task supports
  - Example: ["windows"]
- "hardware": Array
  - Hardware this Evaluation Task is optimized for
  - Example: ["cpu"]
- "data_format": Array
  - Possible data storage formats this Evaluation Task supports
  - Example: ["CSV"]

This list of properties for Evaluation Tasks in the database is not exhaustive, as the database schema may be extended for future additions. MongoDB allows a quick expansion of the database with examples using a different schema.
This metadata shows a baseline that enables a first filtering of evaluation procedures depending on their constraints. The type of query used to acquire a set of applicable Evaluation Tasks for given inputs is shown in chapter 4.3.2.

4.2 MLflow

A powerful tool to facilitate the management of the machine learning lifecycle is called MLflow[8]. This tool provides different components for the implementation of ML methods [98]. As MLflow is used as a baseline for the interoperability of this work, the individual elements are discussed in the following subsection. Afterwards, the uses of the components for this work are outlined.

[8] `https://www.mlflow.org/`

4.2.1 Elements of MLflow

MLflow can be used as a complete package for managing machine learning procedures but an individual usage of the separate parts is also possible. Therefore, the different components are explained individually in the following segments.

MLflow Projects

The first component is MLflow Projects. This is a set of conventions for unifying the execution of ML code. A MLflow Project has a name, an entry point (optionally even multiple entry points) and the execution environment for the entry points. This enables the ability to execute projects from the command line, the Python API and to chain multiple projects together to create a workflow.
The API also allows starting multiple projects to execute them in parallel. This property combined with the possible connection of projects to distributed storage systems (for input and output) is helpful for working with ML on data with high volume, velocity and variety (*Big Data* [66]).

MLflow Models

In order to achieve a workflow that is compatible with arbitrary ML models and techniques, packaging ML models in a standardized way is an important step. Therefore, MLflow Models can define *flavours*, which provide information about the correct interpretation of the model using each specified deployment tool. Additionally, a conda environment file provides information about the used packages or python version. Lastly the requirements file contains the pip dependencies for the model.

MLflow Registry

The MLflow Registry is a centralized place to store and work on a model. This is useful because it gives developers the opportunity to participate and work together on models. APIs and a user interface are provided in order to integrate MLflow into other software or use it as a standalone framework.

MLflow Tracking

MLflow Tracking introduces the concept of *runs*, which allow recording of information about the execution of code. Each run provides an output, which can include metrics in the case of evaluation code [28]. The runs can be provided using APIs for different use cases (Python, R, Java, REST). The saved information regarding the conducted runs can be queried using these APIs or the tracking user interface.

This structure allows the inclusion of execution information from cloud infrastructure and other network devices.

4.2.2 Application in this Work

The Evaluation Tasks that are generated by this framework need to be executable individually and the execution may be distributed on multiple systems. The tasks are therefore designed as MLflow Projects. This enables the execution from the command line or the API and makes parallelization possible.
The supported model formats have to be expandable as complete support for all possible formats is not feasible in a single work. One obvious choice as an initial supported format are MLflow Models. These can be created using various tools and contain a flavour specific to the used tool. Furthermore, if a model was created using MLflow Tracking, the tracking ID will be passed to the model, allowing access to data about the model creation at a later stage in the evaluation process.
The model registry provided by MLflow also offers a possible starting point for the evaluation of models. Due to this, the storage for models is integrated into the framework as well.
MLflow Tracking enables the recording of metrics during execution of code and is therefore used to act as the central element for storing evaluation results. This storage is realized as an SQL database running the collection for result storage.
The results can be viewed using the MLflow interface or the API. This makes it possible to view and compare results directly or use the framework as part of a complete workflow for ML models. In this context, the ExGETa framework could also be integrated into the training process of models by using the evaluation outputs taken from the API as metrics for optimization.

4.3 Task Generation

The generation of executable tasks is a three step process. First the inputs by the user have to be read and analysed. Then the framework has to match the gained information about the model and dataset with the available tasks in the database. If there are existing tasks in the database, these have to be assembled into packets that can be executed independently. The implementation of each of those steps is detailed in the following sections.

4.3.1 Input Processing

The framework expects three types of user input. As each type of input has to be well defined for the framework to be user friendly and expandable in the future, the different inputs model, dataset and metadata are explained in separate sections.

All of the inputs are processed by the ExGETa framework to collect information used to query the database of Evaluation Tasks. Later on the inputs are passed to each task so they may be accessed and used if needed.

Model

The mandatory input is the model which is to be evaluated. The model file can have different formats depending on the methods that were used to create the model. The framework can work with any model in theory, however it is limited by the available tests in the database. If no tests have been configured for the modeltype, no evaluation is possible.
In this work, the framework is configured for SVM models. Future additions are intended for other models (see chapter 7.2). One supported library is scikit-learn[9] [78] if the models have been encoded using pickle or joblib, which are python modules used for storing and loading data [89]. Scikit-learn is chosen as a priority because it is an increasingly popular package for ML applications and it offers capabilities of many ML methods [45].
Scikit-learn provides the classes `SVC`, `NuSVC` and `LinearSVC` for classification and `SVR`, `NuSVR` and `LinearSVR` for regression tasks. Additionally a method for unsupervised outlier detection is implemented in the class `OneClassSVM`. The classes `LinearSVC` and `LinearSVR` use liblinear [35] for efficient computation of SVMs with linear kernels. The other classes use libsvm [26], an open source library based on the works on Sequential Minimal Optimization (SMO) [55, 79] and the algorithms of the SVM^{light} [50] implementation.
As the properties of the model are of interest for both the assignment and the execution of tasks, the attributes and parameters for the different types are listed below:

- SVC and NuSVC
  - Parameters: C (only SVC), nu (only NuSVC), kernel, degree, gamma, coef0, shrinking, probability, tol, cache_size, class_weight, verbose, max_iter, decision_function_shape, break_ties, random_state
  - Attributes: class_weight_, classes_, coef_, dual_coef_, fit_status_, intercept_, n_features_in_, feature_names_in_, n_iter_, support_, support_vectors_, n_support_, fit_status_ (only NuSVC), probA_, probB_, shape_fit_
- SVR, NuSVR and OneClassSVM
  - Parameters: C (only SVR and NuSVR), nu (only NuSVR and OneClassSVM), kernel, degree, gamma, coef0, shrinking, tol, cache_size, verbose, max_iter, epsilon (only SVR)
  - Attributes: coef_, dual_coef_, fit_status_, intercept_, n_features_in_, feature_names_in_, n_iter_, support_, support_vectors_, n_support_, shape_fit_, offset_ (only OneClassSVM)

[9] `https://scikit-learn.org/`

- LinearSVC and LinearSVR
  - Parameters: penalty (only LinearSVC), epsilon (only LinearSVR), loss, dual, C, tol, class_weight (only LinearSVC), multi_class (only LinearSVC), fit_intercept, intercept_scaling, verbose, max_iter, random_state
  - Attributes: classes_ (only LinearSVC), coef_, intercept_, n_features_in_, feature_names_in_, n_iter_

The scikit-learn models can be saved to a file using pickle and joblib. These libraries offer encoding and decoding of model files using multiple protocols. The functions for saving a model are `pickle.dump()` and `joblib.dump()` while the reading a stored model is enabled by the functions `pickle.load()` and `joblib.load()`.
The framework is also set up to process MLflow models (see chapter 4.2.1) of different built-in flavors. MLflow models can be created and processed with many common ML toolkits like Keras[10], PyTorch[11], Spark MLlib[12], TensorFlow[13], ONNX[14] or XGBoost[15] (using the corresponding flavor) [41]. The wide range of toolkits that are supported by MLflow offers the opportunity to quickly enable evaluation methods for them. Saving an MLflow model results in a directory with the files of the model with some additional metadata. This metadata is saved in the form of `requirements.txt`, `conda.yaml` and `python_env.yaml` files listing all required software including the version in different formats and an `MLmodel` YAML[16] file. As this file contains important information about the model, the contents are explained using an example:

```
flavors:
    python_function:
        env: conda.yaml
        loader_module: mlflow.sklearn
        model_path: model.pkl
        python_version: 3.9.7
    sklearn:
        code: null
        pickled_model: model.pkl
        serialization_format: cloudpickle
        sklearn_version: 1.0.2
mlflow_version: 1.26.1
```

[10] https://keras.io
[11] https://pytorch.org
[12] https://spark.apache.org/mllib
[13] https://www.tensorflow.org
[14] https://onnx.ai
[15] https://xgboost.ai
[16] https://yaml.org/

```
model_uuid: be08d503576d40eebaaa911ad20c20e2
utc_time_created: '2022-07-04 15:08:49.593207'
```

The model in the above example can be used with all tools that provide support for one of the flavors `python_function` or `sklearn`. Additional information is provided for each flavor, in this case for example the path of the model. Additionally, the MLflow version used for saving the model, a universally unique identifier of the model and the time of creation are logged. In case the MLflow tracking API is used to record the run, a `run_id` is added to the file to allow any software accessing this model to retrieve artifacts from the linked run [28].
The MLmodel file format allows a simple compatibility with models of different origins without the necessity to implement evaluations multiple times. Another example for an open format is ONNX, which is focused on exchanging models of different ML tools.

Dataset

The dataset input entails challenges due to the diverse range of types of data. As the framework has to be able to (theoretically) offer support for all data inputs, the interface for passing datasets has to be highly open.
Datasets can for example take the form of in-memory storable files, files with a volume exceeding the memory size or a data stream. The data itself can also take different structures, like time-series data or static tabular data. The ExGETa framework needs to be compatible with a wide variety of data. Fortunately the main challenge in this area is to correctly implement the Evaluation Tasks and store the requirements that the dataset has to meet for every task. The framework itself can function agnostic of the data.
In order to specify the type of data supplied for the generation of tasks, the user can set the corresponding parameters in the metainformation input (see next section). Furthermore the user can determine the path to the data that the framework should use.
The processing of the data itself is left to the Evaluation Tasks, so these need to include a step for the detection of the data structure. Depending on the metadata that is given by the user, different methods and formats may be supported by Evaluation Tasks. The usage of online data sources is also possible by passing the datasource to the metadata file. This way of interacting with given datasets makes the framework highly compatible however it also brings drawbacks for the users. Users need to specify exactly how the given data can be interpreted using the extra metadata. Furthermore changes to the dataset may lead to necessary alterations of the metadata file, making the usage of the framework more complex.
This trade-off can be tackled by using data formats with a high amount of compatibility and built-in information about the interpretation of the data.

Metadata

The metadata is passed to the framework using a JSON file. This file has to be created by the user and is needed to correctly process the dataset and model. During the development of the ExGETa framework, the necessity of detailed and correct metadata of the user has become apparent.
Different model formats and model types need to be processed and evaluated using varied techniques. Even models trained using the same toolkits can have different sets of stored properties. An example for this are the scikit-learn algorithms `sklearn.SVC` and `sklearn.LinearSVC`. While `sklearn.SVC` models store the indices of the support vectors, the number of support vectors and the exact support vectors as attributes, none of these exist in `sklearn.LinearSVC` models. Here the support vectors have to be computed from the data by applying the decision function to the training dataset:

```
# given a sklearn.LinearSVC classification model linsvcmodel
    based on the data X:
decision_outs = linsvcmodel.decision_function(X)
sv_i = np.where(np.abs(decision_outs) <= 1 + 1e-15)[0]
support_vectors = X[sv_i]
```

Listing 4.1: Computation of the support vectors for a sklearn.LinearSVC model

The support vectors are calculated using the examples from the dataset that are within the bounds of the margin. This process is needed because the `sklearn.LinearSVC` classifier implementation is based on the liblinear library [35], while the `sklearn.SVC` implementation is based on the libsvm library [26].
Circumstances like this make extensive metadata regarding model and dataset necessary for the framework to properly function. To limit the time and effort needed by the user, the metadata is kept as simple as possible. This however leads to the situation that the metadata format may have to be expanded in the future to make the processing of models or datasets not considered in this work possible.
In the current form, the metadata JSON that is required to run the generation of Evaluation Tasks is defined like this:

```
{
  "name": String,
  "modelmeta": Object = {
    "task": String,
    "method": String,
    "module": String,
    "model_path": String,
```

```
    "loader_module": String
  },
  "datasetmeta": Object = {
    "data_path": String,
    "data_format": String,
    "labelloc": String,
    "traintest_cutoff": Integer,
    "n_labels": Integer,
    "headerlines": Integer,
    "data_separator": String,
    "scaling": String,
    "n_classes": Integer
  },
  "mlflow_uri": String,
  "mlflowexpid": String,
  "os": String,
  "hardware": String,
  "outpath": String
}
```

Listing 4.2: Structure of the metadata input JSON

The part of the JSON dedicated to the model contains all the information needed to recognize, interpret and apply the given model. The specifics of the model have an enormous impact on the applicability of Evaluation Tasks, so this Object in the JSON file has a high priority. The values of this object are detailed below.

- "task": String
  - ML task that is solved by the model
  - Examples: "classification", "regression", "clustering"
- "method": String
  - Method or algorithm used to solve the ML task
  - Examples: "SVM", "DecisionTree"
- "module": String
  - Toolkit that was used to create the model
  - Examples: "sklearn.SVC", "sklearn.NuSVR", "cv.ml.DTrees"

- "model_path": String
  - Path to the model
  - Passing the location of the model like this avoids the need to pass it as an argument to the framework
  - Example: "C://inputs/joblib_model.pkl"
- "loader_module": String
  - Storage and loading format of the model
  - Examples: "sklearn.joblib", "sklearn.pickle"

In order to correctly interpret the dataset given by the user, the `datasetmeta` object in the JSON includes information about the data structure. The exact assembly of the information is dependent on the specific type of dataset used. A short description of the elements of this object is given for a classification task in the following listing:

- "data_path": String
  - Path to the dataset
  - Passing the location of the dataset like this avoids the need to pass it as an argument to the framework
  - Example: "C://inputs/dataset.csv"
- "data_format": String
  - Data storage format of the dataset
  - Example: "CSV"
- "labelloc": String
  - Specifies the position of the label among the features of each example
  - Examples: "f", "l"
- "traintest_cutoff": Integer
  - Number of examples in the training dataset
  - The test data is taken from the data after this positional value
  - Examples: 200, 10000

- "n_labels": Integer
  - Number of labels that can be assigned to a single example
  - Examples: 1, 8, 0 (for unlimited)
- "headerlines": Integer
  - Number of lines in the file used for header data
  - Examples: 0, 1, 5
- "data_separator": String
  - Character or string used to separate the features
  - Examples: ",", ";", "_"
- "scaling": String
  - Scaling methods to be used on the data
  - Examples: "NONE", "01normalization"
- "n_classes": Integer
  - Number of distinct classes in the dataset
  - Examples: 2, 5, 280

The two objects `modelmeta` and `datasetmeta` cover most of the input data needed in this file. Some further general data is needed to specify information about the used system and the connection to the MLflow result tracking platform.
The last important information needed from the user is the path where the resulting Evaluation Tasks should be saved. The path may also lead to a place on a network connected server. All of this additional data is defined as follows:

- "name": String
  - Name for the evaluation run
  - Examples: "MNISTsvm", "CIFARdt"
- "mlflow_uri": String
  - The uniform resource identifier to the MLflow server
  - Examples: "`http://localhost:5000`", "`http://128.105.39.11:23457`"

- "mlflowexpid": String
  - MLflow experiment ID if the outputs should be linked to a prior execution
  - Examples: "17", "412"
- "os": String
  - Operating system of the processing node
  - Examples: "windows", "macos"
- "hardware": String
  - Hardware provided for the execution of evaluations
  - Example: "cpu"
- "outpath": String
  - Path to the location where the Evaluation Tasks should be stored
  - Examples: `"C://inputs/"`, `"\\Server2\Share\EvaluationTasks\"`

4.3.2 Assignment of Possible Tasks

In the second step of the framework, the information of the given and processed inputs is used to query the database for available tasks. As the GridFS database is split into a collection for metadata and a collection for the chunks of the evaluation software (see chapter 4.1), the first step is to discover which tasks are applicable using the metadata collection.
This collection (`fs.files`) is queried with the information gathered in the first step of the framework. This information is comprised of all data included in the input files, mostly the metadata JSON. An example query may look like this:

```
files.find("$and":
     ["metadata.task":  "classification",
     "metadata.method":  "SVM",
     "metadata.type":  "sklearn.SVC",
     "metadata.loader_module":  "sklearn.joblib",
     "metadata.classes":  "2",
     "metadata.os":  "windows",
     "metadata.data_format":  "CSV",
     "metadata.hardware":  "cpu"]
)
```

Only those task IDs are returned by the database where the constraints of the task match the properties of the inputs. This ensures the compatibility and the executability of the evaluation method. In the above example, only Evaluation Tasks are returned that support the assessment of binary (2 classes) `classification` models based on `SVM` algorithms. The required compatibility with `sklearn.SVC` models saved using `sklearn.joblib` is ensured. Furthermore the returned tasks need to be able to be executed on `windows` based systems mainly using the `cpu`. Lastly the tasks need to be able to process tabular datasets in `CSV` files.
The output of the query could be a set of Evaluation Tasks like this:

```
'_id': ObjectId('62e1fc926bda26c12b461630')
'_id': ObjectId('62e1feebfbcd65ff61a15bcb')
'_id': ObjectId('62e26c522569c2e88253ef77')
```

Every element of the produced set of entries represents a single Evaluation Task. The chunks in the database contain the `ZIP` files of the tasks. The `ZIP` files of these tasks are then retrieved using the IDs for the `fs.chunk` entries corresponding to the obtained elements.
This procedure reduces the problem of identifying and assigning the correct set of tasks for given inputs to a simple database query. This makes the procedure easy to understand and enables possibilities for optimization in the future.
The data from this collection is then used in the next step to produce runnable task packets.

4.3.3 Compilation of Task Packets

The task packets need to be modular and independently executable. To facilitate a straightforward development and addition of new Evaluation Tasks for the framework, some redundancy has to be accepted.
All task packets are therefore compiled using the decompressed Evaluation Task software from the previous step, the given user inputs (mainly the model and dataset) and metadata generated in this step. The information stored in the metadata mainly consists of the format of the model and dataset, the requirements needed for the execution and the locations of the important files.
Most of this information is taken directly from the metadata input supplied by the user. To this end, the contents of the input JSON are adopted for the metadata file provided to all Evaluation Tasks.
Additional information can be extracted from the model and dataset during processing. This information is then appended to the input JSON to facilitate the usage of the files for the Evaluation Tasks.
The processing of models is outsourced to a separate file with methods for specific tools. An example for such an extraction of information is given below.

```
def process_LinearSVC(modelfile):
    config["skparams"] = modelfile.get_params()
    if hasattr(modelfile, 'coef_'):
        config["coef_"] = modelfile.coef_.tolist()
    if hasattr(modelfile, 'intercept_'):
        config["intercept_"] = modelfile.intercept_.tolist()
    if hasattr(modelfile, 'classes_'):
        config["classes_"] = modelfile.classes_.tolist()
    if hasattr(modelfile, 'n_features_in_'):
        config["n_features_in_"] = modelfile.n_features_in_
    if hasattr(modelfile, 'feature_names_in_'):
        config["feature_names_in_"] = modelfile.
    feature_names_in_.tolist()
    if hasattr(modelfile, 'n_iter_'):
        config["n_iter_"] = modelfile.n_iter_.item()
```

Listing 4.3: Extraction of information from a sklearn.LinearSVC model

This step makes all information that can be gathered using the built-in methods available to following steps. The set of parameters and all attributes are extracted if they are available.

The resulting JSON file is then copied to the directory of each Evaluation Task inside a newly created `/inputs/` subdirectory. Along with the metadata, the model and dataset file are also passed to the same location.

The Evaluation Tasks are decompressed from their `ZIP` files and assembled in individual folders in the specified directory from the metadata input. This process alongside the insertion of input data for the Evaluation Tasks is executed for each element that is returned from the task database:

```
for t in ts:
    tpath = metafile["outpath"] + "/eval_tasks/task" + str(tn)
    fzip = dbc.getTaskFileByID(t["_id"])
    with zipfile.ZipFile(fzip, 'r') as toextract:
        toextract.extractall(path=(tpath))
        os.mkdir(tpath + "/inputs/")
        shutil.copy(modelfile, tpath + "/inputs/" + os.path.
    basename(modelfile))
        shutil.copy('options.json', tpath + "/inputs/options.
    json")
        shutil.copy(datafile, tpath + "/inputs/" + os.path.
    basename(datafile))
```

```
10      tn = tn + 1
11  print(str(tn) + " tasks have been generated!")
```

Listing 4.4: Assembly of executable Evaluation Tasks in the specified directory

4.4 Evaluation Tasks

As the goal of the framework is the evaluation of ML models, the usefulness of the framework depends heavily on the included Evaluation Tasks. A uniform structure is needed to ensure an equal behaviour of the different Evaluation Tasks. This structure is defined in the next section. Afterwards a suggested pattern for the implementation of Evaluation Tasks is presented. Lastly the implemented evaluation techniques that form an exemplary baseline for the framework are introduced.

4.4.1 File Structure

The Evaluation Tasks need to be executable on different systems without the need to install the dependencies and requirements of all tasks. Only those packages are installed that are needed for the execution of the specific tasks that are generated. This feature is achieved through the use of the package manager conda[17]. This software allows the creation and management of execution environments to properly manage requirements of different software packets.
A conda environment is stored along each Evaluation Task. This environment is specified to contain all dependencies needed to run the task. Before execution of each task, the conda environment is recreated and activated. After this step, the Evaluation Task must be executable on any system (provided conda is installed on the system).
The other required file of every task is the `evaluation_task.py` file. This may contain all of the code or only be a wrapper script for the evaluation software. In any case this file must be the entryfile for starting the Evaluation Task. This loose restriction theoretically allows the development and integration of Evaluation Tasks of any language or form, and therefore makes the framework versatile for all types of additions.
Developers of Evaluation Tasks may additionally use a theoretically unlimited number of other files if the execution of the task is dependent on it. As the Evaluation Tasks are compressed into a ZIP container, the structure of the file of the Evaluation Tasks does not have an impact on their storage in the database. The only limitation is that the usage of a subdirectory named `/inputs/` is prohibited because this is reserved for passing data to the task during compilation (see chapter 4.3.3).
The structure of the `evaluation_task.py` file itself is not strongly defined, however a suggested concept is presented in the following section.

[17] `https://conda.io`

4.4.2 Program Structure

To ensure compatibility with automated execution, the `__main__` function must not have any parameters. This function calls an `input_processing` function designed to extract the relevant information from the inputs in the `/inputs/` subdirectory. If the processing of multiple inputs, like model and dataset, is needed, another division into (in this case) `model_processing` and `dataset_processing` functions may be used. An important design decision at this point is the separation of input processing and the evaluation of the specific metrics.

As MLflow is used for the management of the evaluated metrics, each task has to pass the output measures to the MLflow tracking server. The tracking Unique Resource Identifier (URI) is passed to each task using the metadata file. This URI must be used to establish the connection, as a goal of the framework is to provide a central point where all evaluated metrics can be fetched and viewed.

Each task creates a data structure (in python the data type is a dictionary) and gathers all important metrics into this `loggingdata` variable. The metrics have to be saved as a key-value pair with a numeric value. When the software has completed all evaluations, the `loggingdata` is passed to the MLflow API as a collection of metrics for a specific MLflow run.

Chapter 5

Implemented Tasks

In order to test the framework in regards to compatibility to different evaluations and openness to additions, some Evaluation Tasks are developed and integrated into the framework. These are described in more detail in the following sections, each detailed with a description of the underlying metric and the practical implementation.

5.1 Confusion Matrix Evaluation

A standard procedure towards obtaining different metrics for classification tasks is the analysis of the confusion matrix [93]. A general introduction of the mathematical background is given in chapter 2.3.
The metrics that are to be evaluated are all of the base values of the confusion matrix and all of the derived measures introduced in chapter 2.3. This Evaluation Task therefore represents a case of a statistical analysis of the model performance.
This chapter presents the implementation of the measures and the integration of results into the result storage. To this end, the workflow of the Evaluation Task is described in detail.
The evaluation of the metrics derived from the confusion matrix is implemented for MLflow and scikit-learn models. The code structure follows the schema described in chapter 4.4.2 using a single `evaluation_task.py` file.
First the inputs are processed, starting with the metadata to access information that may be used in the processing of the model or dataset. Then the model and dataset are processed, where the specific type of model is detected. After this, the evaluation of metrics begins, depending on the detected model.
For scikit-learn models, the built-in tools are used to facilitate the evaluation. The function `sklearn.metrics.confusion_matrix(labels y, predictions p)` quickly generates a confusion matrix for binary and multiclass classification [18]. Afterwards, the base metrics are computed for each class:

```
conf_matrix = np.array(sklearn.metrics.confusion_matrix(y_test
    , predictions))

loggingdata = dict()
for i in range(len(conf_matrix)):
  all = np.sum(conf_matrix)
  posclass = np.sum(conf_matrix[:, i])
  posreal = np.sum(conf_matrix[i, :])
  negclass = all - posclass
  negreal = all - posreal
  tp = conf_matrix[i, i]
  fp = posclass - tp
  tn = all + tp - posclass - posreal
  fn = negclass - tn
```

Listing 5.1: Computation of the base metrics of the confusion matrix

Based on these base metrics, all of the derived metrics are computed and added to the set of metrics with a key indicating the class they correspond to:

```
loggingdata = dict()
for i in range(len(conf_matrix)):
  ...
  tpr = tp/posreal
  tnr = tn/negreal
  ppv = tp/posclass
  npv = tn/negclass
  fowlkes_markov_index = sqrt(ppv*tpr)
  ...
  if tpr is not None: loggingdata['true_positive_rate_class_'
   + str(i)] = tpr
  if tnr is not None: loggingdata['true_negative_rate_class_'
   + str(i)] = tnr
  if ppv is not None: loggingdata['
   positive_predictive_value_class_' + str(i)] = ppv
  if npv is not None: loggingdata['
   negative_predictive_value_class_' + str(i)] = npv
  if fowlkes_markov_index is not None:
    loggingdata['fowlkes_mallows_index_class_' + str(i)] =
   sqrt(ppv*tpr)
```

Listing 5.2: Excerpt of the evaluation of metrics derived from the confusion matrix values

After all these metrics are collected for each class, the built-in metrics of scikit-learn are added to the set of metrics:

```
...
loggingdata['accuracy_score'] = sklearn.metrics.accuracy_score
    (y, predictions)
loggingdata['hamming_loss'] = sklearn.metrics.hamming_loss(y,
    predictions)
loggingdata['hinge_loss'] = sklearn.metrics.hinge_loss(y,
    predictions)
loggingdata['log_loss'] = sklearn.metrics.log_loss(y,
    predictions)

return loggingdata
```

Listing 5.3: Excerpt of the base metrics provided by scikit-learn

This collection of metrics then makes up the output of this task, which is passed to the function for saving the metrics to the MLflow server. To demonstrate the Evaluation Task, at some of the experiments using models that are trained on selected datasets to acquire exemplary metrics, this task is included (see chapter 6.2).

5.2 SVM Robustness Evaluation

The accuracy of ML models has been shown to be negatively affected by perturbations of the input data, like blurring in the case of image data [92] or typing errors in the case of text translation systems [7]. The defence against targeted construction of adversarial examples has been proven to be an important research topic for safety relevant or privacy sensitive systems, like voice recognition systems [22] or perception in autonomous driving [21].
To evaluate the vulnerability of a model to changes of the input data, the robustness can be evaluated. In the case of SVM models, the robustness is measured mostly by modifying the inputs in small ways and analysing whether the model output changes. If such small perturbations do not often result in a different model prediction, the model can be viewed as robust.
An existing tool for the evaluation of robustness of SVM models is described in the next section, followed by the specifications of how this tool is used to create an Evaluation Task based on it.

5.2.1 Description of the Tool

The developed framework can be applied to integrate existing evaluation methods. One of these methods is the SVM Abstract Verifier (SAVer)[18] used to evaluate the robustness of SVM models.
This tool, developed by Ranzato and Zanella [82], approximates the robustness using the set of examples X and a perturbation function P. The region $P'(x)$ is an overapproximation of the region $P(x)$, which represents the perturbations of x. An abstract version of the SVM model to be evaluated is then used to classify points in $P'(x)$. These points are not necessarily contained in $P(X)$, but if all evaluated points of $P'(x)$ are classified as elements of the same class, this property also holds true for the subset $P(X)$. Therefore the model is declared robust on point x. If the model outputs more than one class for the points, the model is not declared robust on point x. Note however, that this only computes the provable robustness, as the real amount of points that are robust for all points in their respective regions $P(x)$ might be greater than the number given by SAVer due to the use of the overapproximation $P'(x)$.
The tool can be tuned by modifying the parameters for the abstraction and the perturbation. At this point in time, the tool supports binary classification SVMs and One-vs-One multiclass SVMs in a specified file format. This format is compatible with models trained using scikit-learn and therefore no complicated conversion is needed to integrate the tool into the framework.
SAVer outputs a detailed report on the given model, providing both example-specific information (correct label, predicted label and set of predicted labels in the perturbation set) and overall statistics. The metrics provided by the overall statistic are:

1. Correctness: The number of examples that are initially classified correctly by the model is given as metric for the correctness of the model.

2. Robustness: The robustness (also called stability) of the model measures the total amount of examples in the dataset evaluated as robust.

3. Conditional Robustness: This metric is sometimes only referred to as robustness. The conditional robustness is the number of examples that are classified correctly and proven as robust.

The mathematical and theoretical basis of the tool is beyond the scope of this work, however the original paper [82] describes the specifics in detail along with an analysis of the performance of SAVer. The outputs are shown for a dataset with size n, a perturbation value of ϵ and labels $y(i)$ for example i:

[18] `https://github.com/abstract-machine-learning/saver`

```
1 MODEL_PATH  DATASET_PATH  0      ϵ     y(0)      y_pred(0)        B(0)
2 ...
3 MODEL_PATH  DATASET_PATH  n − 1  ϵ     y(n − 1)  y_pred(n − 1)  B(n − 1)
4 [SUMMARY] Size Epsilon Avg. Time (ms) Correct  Robust Cond. robust
5 [SUMMARY] n    ϵ       t                n_corr       n_rob      n_condrob
```

Listing 5.4: Output metrics produced by SAVer

The robustness evaluation tool computes the predictions of the model $y_{pred}(i)$ and the superset of labels in the adversarial region $B(i)$. Based on these calculations, the number of correct classifications n_{corr}, robust classifications n_{rob} and conditional robust classifications $n_{condrob}$ according to the definitions (see chapter 5.2.1) are derived.
The upside of this representation is an intuitive overview, however the focus on human readability leads to a difficult integration into an ML pipeline. The tool does not provide an API or data format for usage of the metrics at a later stage of the model evaluation or during hyperparameter tuning.

5.2.2 Implemented Software

In contrast to the evaluation in chapter 5.1, creating an Evaluation Task based on the SAVer tool entails the integration of external software. This software is packaged into the file structure of the Evaluation Task and stored in the task database.
For the evaluation of robustness, SAVer requires the model to be in a special format [82]. This format follows the structure[19]:

```
ovo
<feature space size>
<number of classes>
<kernel type and parameters>
<class 1><number of support vectors for class 1>
<class 2><number of support vectors for class 2>
...
<class N><number of support vectors for class N>
<alpha coefficients>
<support vector»
<biases>
```

In order to offer compatibility with common model formats, the model mapper[20] implemented by the creators of SAVer is used on the input model. The converted model is then saved to allow access.
The L-infinity perturbation with a mutation factor of 1% is applied and the robustness

[19] taken directly from the documentation: `https://github.com/abstract-machine-learning/saver`
[20] `https://github.com/abstract-machine-learning/data-collection`

evaluated. To achieve this, the SAVer tool is executed with the converted model, dataset and the chosen mutation parameters.
Executing the SAVer tool with the converted model and dataset yields an output which contains a result analysis for each sample and an overall evaluation (see chapter 5.2.1). This output is then parsed to extract the different metrics from the last line:

```
result = program.communicate()[0]

lastrow = (str(result, 'utf-8').splitlines())[len(str(result,
    'utf-8').splitlines()) - 1]

siz = float(lastrow.split()[1])
loggingdata['epsilon'] = float(lastrow.split()[2])
loggingdata['avgtime'] = float(lastrow.split()[3])
ncorrect = float(lastrow.split()[4])
nrobust = float(lastrow.split()[5])
ncondrobust = float(lastrow.split()[6])
```

Listing 5.5: Parsing of the SAVer outputs

The values for the ϵ parameter and the average analysis time are immediately stored in the loggingdata as they are not needed for further calculations. The other metrics are combined with the size of the dataset to acquire ratios for correctness, robustness and conditional robustness:

```
loggingdata['ratiocorrect'] = ncorrect/siz
loggingdata['ratiorobust'] = nrobust/siz
loggingdata['ratiocondrobust'] = ncondrobust/siz
loggingdata['size'] = siz
loggingdata['ncorrect'] = ncorrect
loggingdata['nrobust'] = nrobust
loggingdata['nconditionalrobust'] = ncondrobust
```

Listing 5.6: Calculation of robustness ratios

All of the data that is computed and saved in the `loggingdata[]` variable, is then stored in the result database. This is implemented by logging the metrics to the MLflow tracking server.

5.3 Theoretical Bounds for SVMs

Some ML models can be evaluated in regards to theoretical guarantees. SVMs in particular are based on sound mathematical principles (see also chapter 2.1). As such, different

properties of SVMs can be used to compute estimations and bounds for certain metrics [49, 95]. The work of Joachims [53] on classification using SVM models is used as a basis for possible evaluations.
In the following section, the mathematical theory is described. After this, the implemented software evaluating the theoretical properties is presented. Lastly the process of combining the outputs of this software with an interpretation is detailed as an example of generating a different output than a set of metrics.

5.3.1 Theoretic Background

The bounds explored in this section are all taken from the works of Thorsten Joachims. The main sources of this Evaluation Task are the works on generalization performance [51] and text classification [53] in the context of SVMs. The set of estimators of performance measures based on the $\vec{\xi}$ and $\vec{\alpha}$ ($\xi\alpha$ estimators) are described in more detail in the following section.
The values α_i of vector $\vec{\alpha}$ describing the coefficients of the training examples are zero for exactly those examples that are not support vectors. The values ξ_i of $\vec{\xi}$ describe the loss of each example $\vec{x_i}$ in the training set. Considering a training set S with n elements and the hypothesis $h_{\mathcal{L}}$, which learner $\mathcal{L}$ returns, the $\xi\alpha$ estimator for the error rate $Err^n_{\xi\alpha}(h_{\mathcal{L}})$ of a stable soft-margin SVMs is:

$$Err^n_{\xi\alpha}(h_{\mathcal{L}}) = \frac{d}{n}$$

$$with\ d = |\{i : (\rho\alpha_i R^2_{\triangle} + \xi_i) \geq 1)\}|$$

In this formula, ρ is a parameter and $R^2_{\triangle}$ is an upper bound on $c \leq K(\vec{X}, \vec{x'}) \leq c + R^2_{\triangle}$ for all $\vec{x}$, $\vec{x'}$, some constant c and a Mercer Kernel [68] K. The value $\rho = 1$ has been suggested by Joachims [51] as a good choice for the parameter. This estimator of the error rate has been shown to be output higher values than the true error rate on average [53].
Using the same inputs as above, the following values can be computed:

$$\begin{aligned} d_{-+} &= |\{i : y_i = 1 \wedge (\rho\alpha_i R^2_{\triangle} + \xi_i) \geq 1)\}| \\ d_{+-} &= |\{i : y_i = -1 \wedge (\rho\alpha_i R^2_{\triangle} + \xi_i) \geq 1)\}| \\ n_{+} &= |\{i : y_i = 1\}| \\ n_{-} &= |\{i : y_i = -1\}| \end{aligned}$$

These values are helpful for estimating the Precision, Recall and F1, because they represent the amount of positive examples where $(\rho\alpha_i R^2_{\triangle} + \xi_i) \geq 1$ is true (d_{-+}), the amount of negative examples where this condition holds (d_{+-}) and the total amount of positive (n_+)

and negative (n_-) examples in the training set S. The $\xi\alpha$ estimators of these metrics are then calculated analogously to the corresponding metrics (see chapter 2.3):

$$\begin{aligned} Prec^n_{\xi\alpha}(h_{\mathcal{L}}) &= \frac{n_+ - d_{-+}}{n_+ - d_{-+} + d_{+-}} \\ Rec^n_{\xi\alpha}(h_{\mathcal{L}}) &= 1 - \frac{d_{-+}}{n_+} \\ F1^n_{\xi\alpha}(h_{\mathcal{L}}) &= \frac{2n_+ - 2d_{-+}}{2n_+ - d_{-+} + d_{+-}} \end{aligned}$$

Just like the estimator of the error rate, the $\xi\alpha$ estimators if Precision, Recall and F1 have been shown to produce lower values than the true measures on average [51].
The derivations, proofs and further information about these bounds exceed the scope of this work, as the focus is on the development of an evaluation framework. The background for these estimators has been provided by Joachims [53] and can be retraced in his work.
An addition to the estimators above is a bound on the expected test error. This bound was given by Vapnik [96] and is based on the Vapnik-Chervonenkis dimension (VC dimension). The VC dimension describes the maximum number of points that can be shattered by a set of functions [20]. For a classifier (representing the set of functions) to shatter a point configuration, the classifier must be able to find a division of the points that separate all positive from all negative examples perfectly for every possible assignment of positive and negative values.
In order to form the bound on the expected test error $R(\alpha)$, some definitions have to be introduced. Given a classifier $f(x, \alpha)$ tasked with learning a mapping $x_i \mapsto y_i$ of the vectors $x_i \in \mathbb{R}^n, i = 1, ..., l$ to the labels y_i using the parameters α, let the loss function take the form $\frac{1}{2}|y_i - f(x_i, \alpha)|$. The empirical mean error rate (on the training set) is then defined by:

$$R_{emp}(\alpha) = \frac{1}{2l} \sum_{i=1}^{l} |y_i - f(x_i, \alpha)|$$

Then for V as the VC dimension and a chosen η with $0 \leq \eta \leq 1$ the following bound holds with probability $1 - \eta$ [20]:

$$R(\alpha) \leq R_{emp}(\alpha) + \sqrt{\frac{V(ln(\frac{2l}{V}) + 1) - ln(\frac{\eta}{4})}{l}}$$

Other bounds have been explored specifically for Bag-of-Words data for text classification using TCat-Concepts [52], however these have not been implemented as an Evaluation Task yet, as the model and data have to satisfy some specific requirements in order for the bounds to be applicable. The implementation of an evaluation of these bounds does however constitute a useful addition to the framework in the future.

5.3.2 Implementation of the Calculations

In order to evaluate the measures described in chapter 5.3.1, a python application is implemented. The general guidelines for the realization of Evaluation Tasks (see chapter 4.4) are followed. The application does not rely on additional files or assets and consists only of the python executable.
First the input data is split into training dataset and test dataset according to the information given by the user. The training data is used to compute the misclassifications on the training data by applying the model. This measure when divided by the number of training examples yields the empirical error rate on the training data. Next the values for calculating the $\xi\alpha$ estimators are calculated by iterating through all training examples and their slack values:

```
for s in slack:
    if(count < len(alphas)):
        if(X_train[i] in clf.support_vectors_):
            if(p * alphas[count] * R + s >= 1):
                if(y_train[i] == 1):
                    dmp = dmp + 1
                if(y_train[i] == 0):
                    dpm = dpm + 1
                d = d + 1
            if(2 * alphas[count] * R + s >= 1):
                x = x + 1
            count = count + 1
        else:
            if(s >= 1):
                x = x + 1
                if(y_train[i] == 1):
                    dmp = dmp + 1
                if(y_train[i] == 0):
                    dpm = dpm + 1
                d = d + 1
    else:
        if(s >= 1):
            x = x + 1
            if(y_train[i] == 1):
                dmp = dmp + 1
            if(y_train[i] == 0):
                dpm = dpm + 1
```

```
28              d = d + 1
29      if(y_train[i] == 1):
30          nplus = nplus + 1
31      if(y_train[i] == -1):
32          nminus = nminus + 1
33      i = i + 1
```

Listing 5.7: Calculation of required values for the computation of $\xi\alpha$ estimators

Of special interest during this computation is the distinction between examples that are support vectors of the model and those that are not. This is important, because the formats of common SVM model implementations like the ones in scikit-learn only save α_i values for support vectors, as $\alpha_i = 0$ if i is not a support vector. For the computation of the measures needed for $\xi\alpha$ estimators all examples need to be included.
After the values for d_{-+}, d_{+-}, n_+ and n_- are calculated, the $\xi\alpha$ estimators can be explored. The program for the computation simply applies the formulas of chapter 5.3.1:

```
estimator = d / n
estRec = 1 - (dmp/nplus)
estPrec = (nplus - dmp)/(nplus - dmp + dpm)
estFone = ((2*nplus) - (2*dmp))/((2*nplus) - dmp + dpm)
nmax = max(nplus, nminus)
# value for always predicting most common class:
maxvalpredictoracc = nmax/n
```

Listing 5.8: Calculation of the $\xi\alpha$ estimators

Lastly the bound on the expected test error is calculated using the VC dimension. In the case of a linear SVM the VC dimension is simply the number of input features + 1 as was mathematically proven [4]. Following the formula from chapter 5.3.1, the calculation is:

```
rabound = remp + sqrt((vcdim*(log((2*l)/vcdim)+1) - log(eta/4)
    )/l)
```

Listing 5.9: Calculation of the bound on the expected test error

All of the evaluated bounds, the VC dimension and the $\xi\alpha$ estimators are then stored alongside the evaluation metrics from the test data. This output data is logged to the MLflow tracking server as metrics. Additionally a report is generated, which is detailed in the following section.

5.3.3 Generation of a Report

Providing users with a simple and understandable report on the evaluated metrics and bounds has great advantages in regards to the usability of the evaluation. To this end, a

Portable Document Format (PDF) file is generated which contains the vital information and a basic visualization.
The generation of a PDF file is enabled through the use of the PyFPDF[21] library. The construction of the file is simple:

```
pdf = FPDF()

# add a page
pdf.add_page()

# set style and size of headline font
pdf.set_font("Arial", size = 20, style="B")
pdf.cell(200, 30, txt = "Bounds and Measures Report", ln = 1,
    align = 'C')

# set style and size of text font
pdf.set_font("Arial", size = 15, style="")
pdf.cell(200, 10, txt = "Vapnik Chervonenkis dimension: " +
    str(vcdim), ln = 1)
...
```

Listing 5.10: Creation of a file using PyFPDF

In this way, all of the metrics calculated as detailed in chapter 5.3.2 are included in the report and explained using a short preceding text. This is already an improvement to the storage of raw values for users.
Additionally the significance of some values is illustrated by short texts. These are available regarding the value of the test accuracy and error. A simple check, if the accuracy is higher than one of a classifier which always predicts the most common class is implemented. If this check fails, the model must be reviewed as it is most likely that the training was not successful.
Furthermore, the bound on the expected test error (see chapter 5.3.1) is interpreted for a value of $\eta = 0.05$. This means the bound is holding with a probability of 95%, so it is highly advised to verify the model and dataset if the bound is invalid. The meaning of these interpretations is highlighted by changing the text color depending on the status of the evaluations:

```
if((1-score) > rabound):
    pdf.set_text_color(194,8,8) # set textcolor to red
```

[21] https://github.com/reingart/pyfpdf

```
3     pdf.multi_cell(200, 10, txt = "Test Error exceeds bounds!
      Probability for the bound holding is 95 percent.\nPlease
      check the model and dataset!", align = 'C')
4 else:
5     pdf.set_text_color(8,194,8) # set textcolor to green
6     pdf.multi_cell(200, 10, txt = "Test Error is within the
      bounds.\nProbability for the bound holding is 95 percent.",
      align = 'C')
```

Listing 5.11: Addition of interpretations to the report file

For users interested in the background of these interpretations and the metrics evaluated for the model, the references are supplied at the bottom of the report. These are the works of Burges [20] and Joachims [53] on SVMs and their properties. The additions and output of the model are straightforward:

```
1 # insert references for the Bounds
2 pdf.multi_cell(200, 10, "References: ")
3 pdf.set_font("Arial", size = 8, style="")
4 pdf.multi_cell(200, 5, "Burges, Christopher J. C.: A Tutorial
     on Support Vector Machines for Pattern Recognition. Data
     Mining and Knowledge Discovery, 2(2):121-167, jun 1998.\n "
     )
5 pdf.multi_cell(200, 5, "Joachims, Thorsten: Learning to
     Classify Text Using Support Vector Machines: Methods,
     Theory and Algorithms. Kluwer Academic Publishers, Norwell,
      Massachusetts, USA, 2002.\n ")
6 # save the pdf with name Boundsreport.pdf
7 pdf.output("Boundsreport.pdf")
```

Listing 5.12: References in the bounds report for the calculations

As the location and name of the report are defined by the program, the created PDF file containing the report is then added to the MLflow tracking server as an artifact. This allows users to view or download the PDF file and have instant access to a prepared interpretation of the metrics.

5.4 Energy Consumption Measurement

In recent years the measurement and limitation of the energy consumption of ML methods has become a focus of research [40]. Due to this development, a technique for measuring energy consumption is chosen as one of the Evaluation Tasks initially implemented for the

ExGETa framework.
In the next section the software is presented upon which the Evaluation Task is built. Afterwards the implementation of the Evaluation Task is detailed with the focus on how the existing tool is integrated. Lastly, the interpretation of the metrics achieved from the task are described and assessed.

5.4.1 Description of the Tool

A framework is chosen that allows sophisticated visualization of results to showcase the capabilities of ExGETa. The CodeCarbon emissions tracker[22] based on prior work from Lacoste et al. [56] and Lottick et al. [63] offers this capability.
CodeCarbon is an advanced tool for analysing energy consumption as it takes several factors of the execution into account. The computing infrastructure, usage of components and running time are combined to acquire measures for the power consumption. These measurements are then combined with location data (if available) to evaluate an estimate of the CO_2 equivalent (CO_2eq) emissions caused by the computation [86].
The CodeCarbon tool supports various hardware resources. Data regarding the Graphics Processing Unit (GPU) is gathered through the python bindings to the Nvidia management library (pynvml). CPU data is collected using different techniques depending on the operating system. For Linux systems, the Intel Running Average Power Limit (RAPL) energy sensors are a straightforward method of estimating power consumption for individual processes [43]. The CodeCarbon tool uses the RAPL files that are generated automatically to track the energy consumption.
For Windows and macOS systems with a CPU manufactured by Intel, the Intel Power Gadget[23] allows the tracking of power consumption. If this software based tracking is not successful, CodeCarbon uses a dataset containing over 2000 processors stored in a `cpu_power.csv` file as a fallback option.
To compute estimations of CO_2eq the location of execution is taken into account. Different cloud service providers and their locations are assigned individual factors. The execution on private hardware also results in different CO_2eq values depending on the location, as for example the generation of one kWh of energy in France led to 58g CO_2 emissions in 2021. In the same year, the generation of one kWh of energy in Poland led to 736g CO_2 emissions [71].
The CodeCarbon emission tracker outputs a `.csv` file with the following information about the experiment:

- timestamp: Date and time of execution
- project_name: Name of the current project

[22] `https://codecarbon.io/`
[23] `https://www.intel.com/content/www/us/en/developer/articles/tool/power-gadget.html`

- run_id: Generated unique identifier
- duration: Duration of execution in ms
- emissions: Evaluated CO_2eq emissions in kg
- energy_consumed: The energy consumed by the executing hardware in kWh
- country_name: Country of execution
- country_iso_code: 3-letter code of the country of execution according to ISO 3166-1 alpha-3 [24]
- region: State or province of execution
- cloud_provider: Name of the cloud provider (supported: Amazon Web Services, Microsoft Azure, Google Cloud Platform)
- cloud_region: Hosting region of the hardware of the cloud provider
- os: Operating system of the executing hardware
- python_version: Version of Python used for execution
- cpu_count: Number of CPU cores
- cpu_model: Model of main CPU
- gpu_count: Number of GPU processors
- gpu_model: Model of main GPU
- longitude: Geographical coordinate of longitude
- latitude: Geographical coordinate of latitude
- on_cloud: Boolean (Y/N) whether the execution is based on a cloud platform

5.4.2 Integration as a Task

The integration of the CodeCarbon tool into the ExGETa framework is straightforward. As CodeCarbon can be installed as a python package, the insertion of all requirements to the conda file is enough to allow an access to the desired methods. The mandatory python file `evaluation_task.py` can contain all of the code needed for evaluating a model with CodeCarbon.
First the inputs are processed analogously to the procedure in the other implemented Evaluation Tasks. The evaluation of energy consumption during training of the model can then be achieved with the following code:

```
...
tracker = EmissionsTracker(output_dir="./")
tracker.start()
predictions = loaded.fit(X, y)
emissions = tracker.stop()
...
```

Listing 5.13: Computation of the power consumption and emissions using CodeCarbon

The `emissions.csv` file containing all results is then generated in the path of the Evaluation Task. The metrics contained in the file described in section 5.4.1 are then read and stored as float values. The following snippet shows the storage of the outputs to the MLflow server:

```
...
with mlflow.start_run(run_name='Codecarbon Efficiency
    Evaluation'):
     mlflow.log_metrics(tolog)
     mlflow.log_artifact("emissions.csv")
```

Listing 5.14: Logging process of the CodeCarbon outputs

In contrast to tasks that only log the evaluated numeric metrics to MLflow (see section 5.1), this task shows the capability to log files of any format to MLflow. Figure 5.1 shows the output of a run of the energy consumption Evaluation Task as stored in MLflow. The reasoning on why this may be useful for this task is given in the next section.
The integration of this tool highlighted the challenge of this framework of being dependent on the correct implementation of underlying tools. During the integration of the CodeCarbon tool, a bug in the code led to problems in the execution of evaluations.
In the `core/cpu.py` file, a hardcoded path is given for the Intel Power Gadget software. If this software is not reachable at the exact path, the execution of the Intel software for measuring energy consumption fails. The hardcoded path is `C:/Program Files/Intel/Power Gadget 3.5/PowerLog3.0.exe`. As the version 3.5 is not the most recent release of the software, the path for the version 3.6 has changed when using the standard installation.
The issue of using this hardcoded path has been known to the developers since at least 2021[24], however no permanent and future proof solution to this issue is implemented yet. While the problem is easily fixed for the moment by changing the installation path of the Intel software or the hardcoded path in the code of CodeCarbon to the correct one, the problem highlights a challenge of the framework, as these types of modification should not be required by the user.

[24] `https://github.com/mlco2/codecarbon/issues/109`

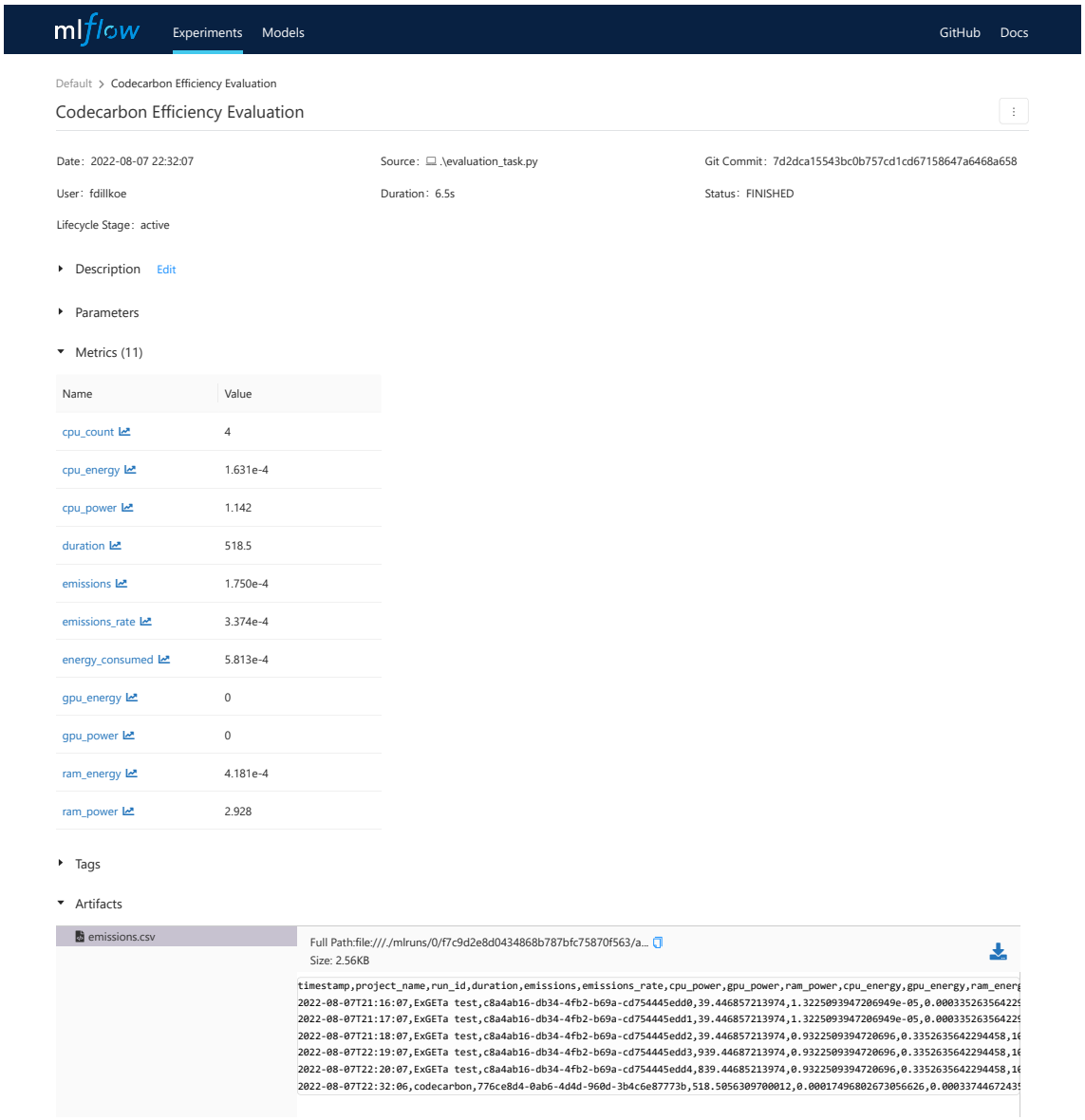

Figure 5.1: MLflow visualization of the outputs of the energy consumption Evaluation Task

5.4.3 Outputs and Interpretation

As described in section 5.4.1, this task produces multiple metrics. All of the metrics are stored as such to the MLflow tracking database. The MLflow API or the user interface can then be used to access the results. Additionally to the storage as metrics, this Evaluation Task also logs the resulting `emissions.csv` file as an artifact, to provide a possible use for visualization.
The CodeCarbon package contains an implementation of a visualization method for emissions tests. This visualization is implemented in the form of a webapplication using Dash[25]. This application called CarbonBoard allows the visualization of metrics of specific projects or the overall energy consumption (if available). Based on the works of Lacoste et al. [56] and Lottick et al. [63] the evaluated energy consumption and emissions are shown in relation to equivalent measures like miles driven in a car, weekly household emissions or television hours. Additionally global benchmarks about the equivalent emissions in different countries and information about the power mix in other regions are given.
An example of such an application generated using data from a CodeCarbon Evaluation Task is given in figures 5.2 and 5.3. This visualization technique offers an added value for users that are not familiar with the interpretation of energy values and helps to put the outputs into context.

[25] `https://plotly.com/dash/`

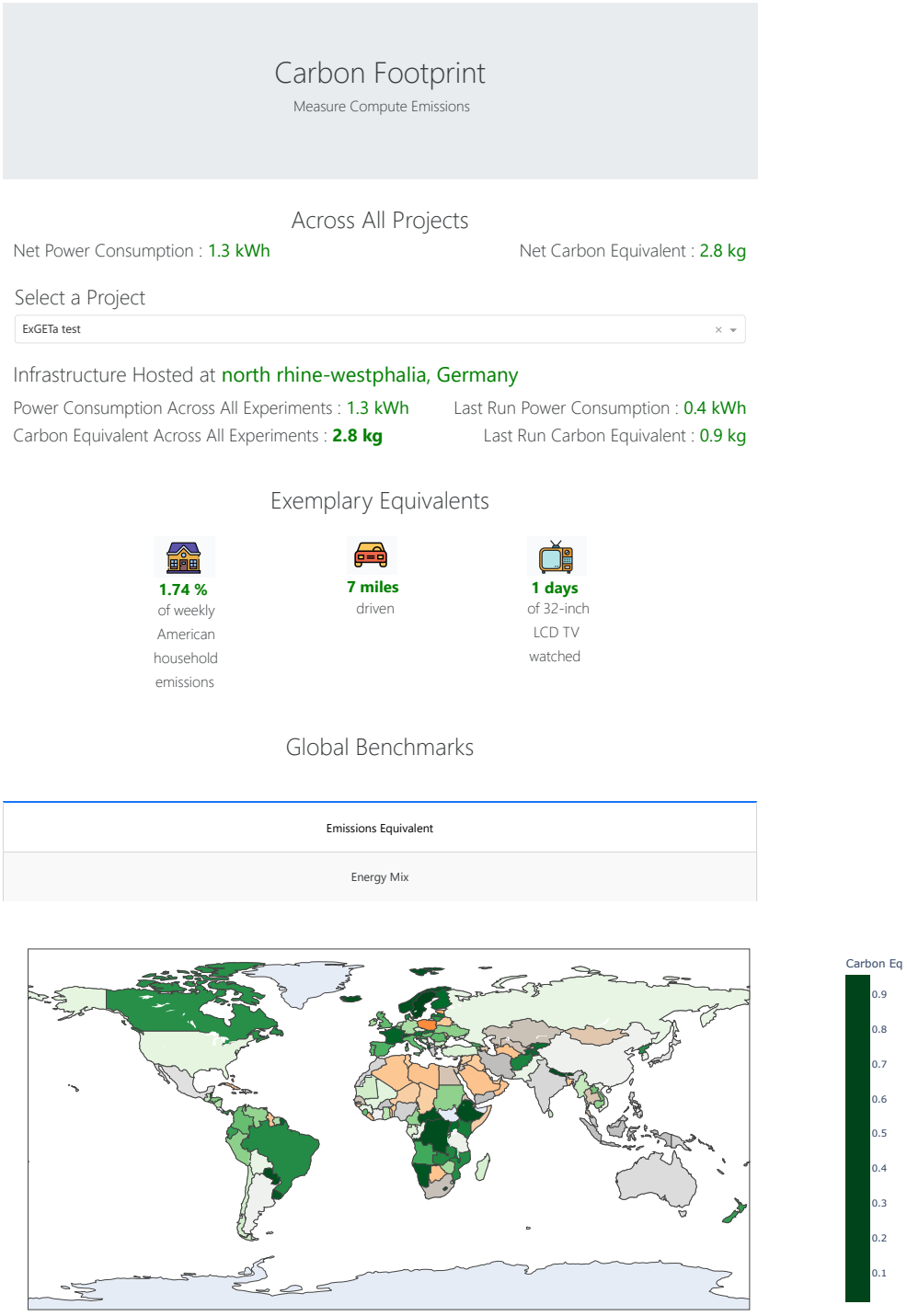

Figure 5.2: Generated Dash application with outputs of the CodeCarbon energy consumption Evaluation Task (part 1)

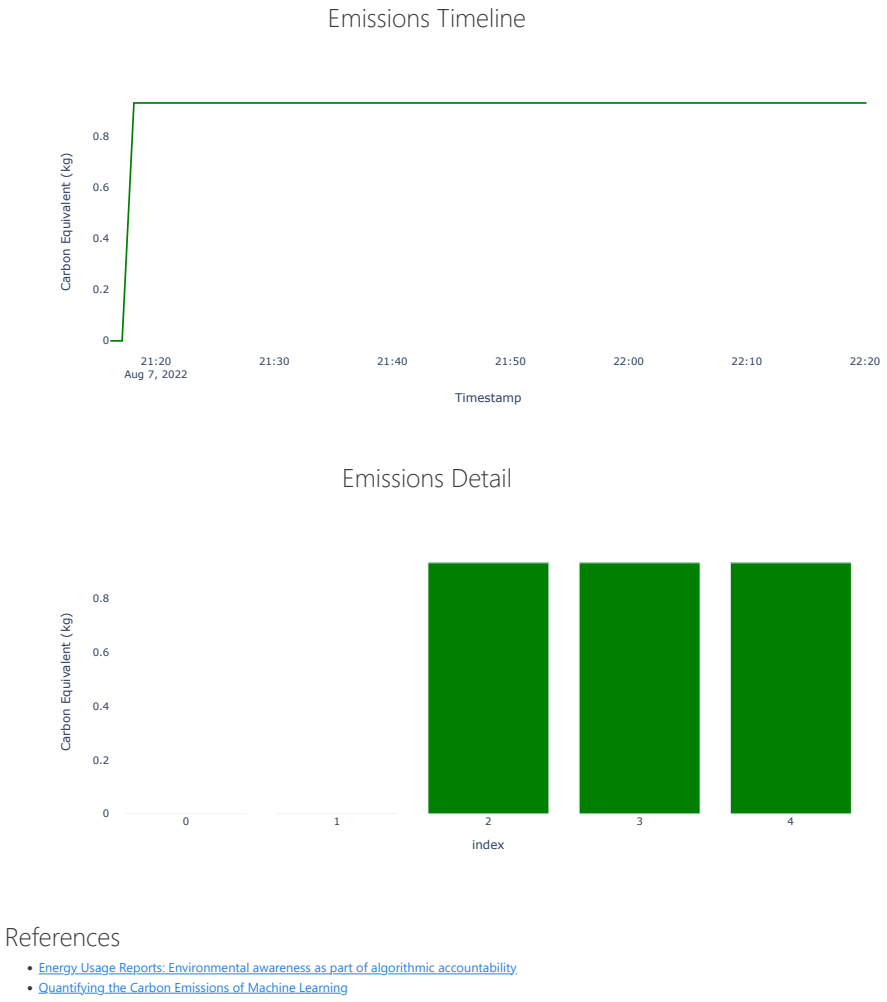

Figure 5.3: Generated Dash application with outputs of the CodeCarbon energy consumption Evaluation Task (part 2)

Chapter 6

Evaluation

To evaluate the developed framework, different techniques are used. The criteria for the evaluation of the framework are laid out in the next section, followed by a plan of experiments aimed at specific properties. Afterwards the developed software is analysed on a higher level of abstraction regarding achieved goals and overall quality.

6.1 Test Criteria

The framework is evaluated with a focus on specific aspects. In order to achieve a meaningful assessment of the framework, an obvious aspect is which central elements of the framework (see also chapter 3.1) are developed.
The status of these goals is analysed using empirical executions with selected models and datasets (see chapter 6.2). The inputs are chosen to showcase the outputs of the framework with different types of models and datasets.
Each of the experiments and their results are described in detail. Both the applicability of Evaluation Tasks and the correct execution of the tests is looked at.
The developed software is then to be tested on the goals that are achieved and the overall software quality. The empirical experiments can be used to evaluate the achieved goals.
To acquire a more theoretical evaluation of the developed software, the ISO/IEC 25010 [17] standard for evaluating the quality of software is used. This method of evaluation consists of eight main measures containing subcharacteristics (see figure 6.1). Each of the measures is evaluated based on the implementation and experimental results.
This standard process is chosen to provide a neutral basis for evaluation. Each of the characteristics describes a different property. While the importance of the individual aspects varies depending on the software product, an evaluation of every aspect guarantees a meaningful overview according to the formulated goals of the standard [17]. A report loosely based on the standard is given in chapter 6.7.2.

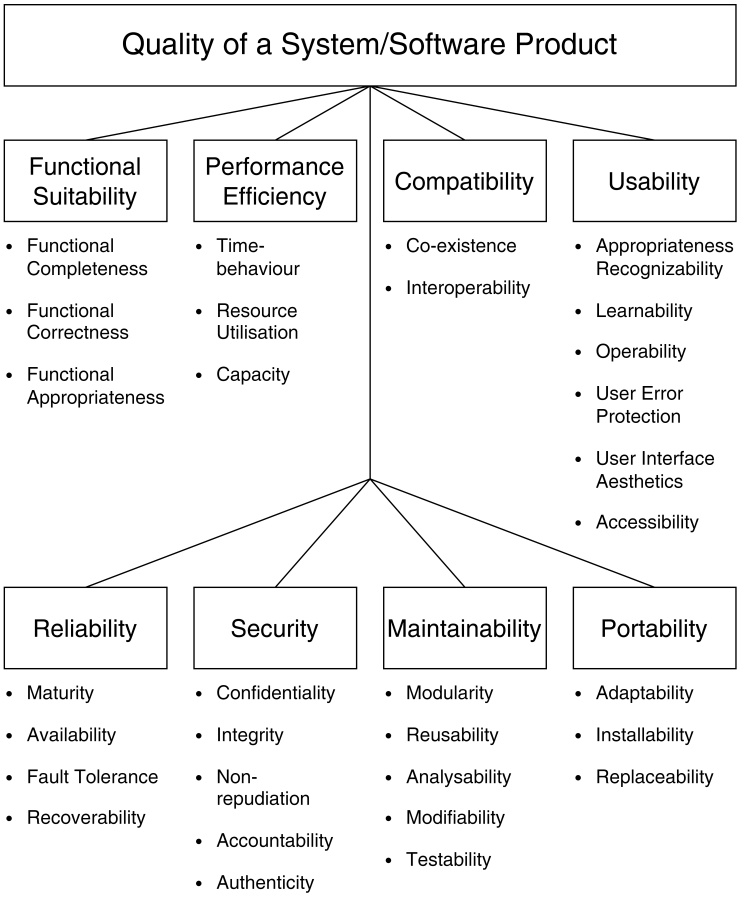

Figure 6.1: Software Quality Model according to ISO/IEC 25010:2011

Name	Task	n	c	|x|	Reference
MNIST	classification	784	10	70 000	LeCun et al. [57]
California Housing	regression	8	-	20 640	Pace and Barry [74]
Ionosphere	classification	34	2	351	Sigillito et al. [88]
Iris	classification	4	3	150	Fisher [38]

Table 6.1: Datasets chosen for the evaluation of the ExGETa framework

6.2 Experiment Composition

To examine the behaviour of the framework with a variety of inputs, a number of different models and datasets are chosen. An overview of the datasets and their basic properties model task, number of features (n), number of target classes (c) and number of samples (|x|) is given in table 6.1.

As a simple baseline, an extremely common dataset in ML publications is chosen. This dataset is the Modified National Institute of Standards and Technology (MNIST) dataset first used by LeCun et al. [57]. The dataset consists of 70 000 greyscale images of handwritten digits. Each image has a format of 28 pixels by 28 pixels and is assigned a digit as label. The classification of the MNIST data has been studied extensively [6] with models achieving error rates as low as 0.24% without any preprocessing or data augmentation [13]. Due to the popularity and widespread use as a benchmark for image processing systems, the MNIST database is chosen as basic benchmark for the framework. A standard SVC model is trained using scikit-learn and evaluated using the ExGETa framework (see section 6.3).

SVMs can be used for tasks other than classification, such as regression or clustering. To evaluate the compatibility of the framework with different learning tasks, a regression model trained using scikit-learn is chosen.

The dataset first chosen for training and testing of the regression model is the Boston Housing dataset. Originally published in 1978 [46], the dataset is rather small with only 506 examples and 14 variables. Similarly to the MNIST dataset for classification, this dataset has also been applied extensively to different regression methods and has become a standard case among the available datasets to the extent that some ML frameworks include it as examples (like Tensorflow Keras [70] or the R package mlbench [58]).

As this dataset has recently been highlighted as including ethically questionable beliefs, it has been replaced in frameworks like scikit-learn[26], SHAP[27] or CasualNex[28]. The reason for this is mainly the feature B, which encapsulates the belief of the authors that racial self-segregation has a positive effect on the values of houses.

[26] `https://github.com/scikit-learn/scikit-learn/pull/18594`
[27] `https://github.com/slundberg/shap/pull/2501`
[28] `https://github.com/quantumblacklabs/causalnex/issues/91`

Model	Dataset	Conf. Matrix	Robustness	Energy Cons.	Bounds Rep.
SVC	MNIST	✓	✓	✓	
SVR	California H.			✓	
Linear SVC	Ionosphere	✓	✓	✓	✓
Dec. Tree	Iris	✓		✓	

Table 6.2: Applicability of the Evaluation Tasks to the trained models

A dataset with a similar premise is the California Housing dataset, introduced in 1997 by Pace and Barry [74]. This dataset contains 20 640 examples with 8 numeric describing features and a median house value as target. This dataset is used as a replacement for the planned Boston Housing dataset. The training of the SVR model and the resulting evaluations based on the California Housing dataset are detailed in section 6.4.
Binary classification allows some specific evaluation techniques, which is the reason why an additional SVM is trained using the SVC class of the scikit-learn package. The kernel of this model is configured to be linear.
To evaluate the model, a dataset with a binary classification task is chosen. The Ionosphere dataset was originally analysed in 1989 as an application for Neural Networks [88]. The dataset consists of 351 examples with 34 features from a radar system in Goose Bay, Canada. Additionally, each example is labeled with a binary class for either *good return* or *bad return*.
The learning task is therefore a binary classification of this label. Details about the training of the model can be found in section 6.5.
Lastly a Decision Tree model is chosen as a comparison and to assess the capability of the ExGETa framework to be applied to other model types. The implementation of Decision Trees in scikit-learn is chosen as it is based on the CART algorithm described in chapter 2.2.
The Iris dataset is chosen for the Decision Tree model. This dataset was first published in 1936 [38]. It has become a popular basis for research, with multiple different versions of the dataset being in use today due to data corruption [11]. The dataset is very small with only 150 samples and 4 describing features (petal width/length and sepal width/length). The examples are each labelled with one of the three species of the Iris genus: Iris setosa, Iris virginica and Iris versicolor.
The simplicity of the dataset makes human understandable Decision Trees possible. The training and experiments of this model are explained in section 6.6.
As not all implemented Evaluation Tasks are applicable to the trained models, the framework correctly only generates the executable ones. An overview of the applicability of the Evaluation Tasks is given in table 6.2. The reasons for the constraints are detailed in the sections for each model.

6.3 Experiment 1: SVC on MNIST

A simple SVC model using a Radial Basis Function (RBF) kernel is trained using the common MNIST training dataset [57]. This training dataset consists of 60 000 of the total 70 000 greyscale images of handwritten digits.
As each of the 28 by 28 pixels represents a single feature, each datapoint has 784 describing features and a label between 0 and 9. Before training, the data is normalized as is common procedure.
The regularization parameter is chosen as $C = 0.5$ and the training of the model is straightforward using the standard scikit-learn method. After the training, the model is saved locally using pickle:

```
clf = SVC(C=0.5, kernel='rbf').fit(X_train,y_train)
# save model
with open('pickle_mnist.pkl#, 'wb') as pickle_file:
    pickle.dump(clf, pickle_file)
```

Listing 6.1: Training of an RBF SVM model for classifying the MNIST dataset

This model is applicable to three of the implemented Evaluation Tasks. Only the report on the theoretical bounds cannot be generated, as it is only applicable to binary classification tasks at the current time.

6.3.1 Confusion Matrix Evaluation

As the model solves a classification task using an SVM trained using scikit-learn, the Evaluation Task for the confusion matrix evaluation can be applied. The ExGETa framework therefore generates this task correctly.
Due to the high number of classes, the confusion matrix can be used to evaluate measures for each individual class. This results in 255 total metrics that are evaluated for this model and dataset.
Although the full set of evaluations is too extensive to be analysed in detail at this point, it should be noted that the metrics allow a comparison of the model in regards to the different classes. As an example, the accuracy on the model for examples of class 1 is 99.7%, while examples of class 7 only have an accuracy of 98.6%. The full results of this evaluation are given in the appendix.

6.3.2 Robustness Evaluation

The SAVer robustness evaluation should be applicable to the model, as scikit-learn SVM models are supported by the tool. The generation of this evaluation for the given inputs of the ExGETa framework is therefore correct.
Nevertheless, the execution of this Evaluation Task fails with an error. The problem is

found to be a line in the `classifier_mapper.py` file of the developers of SAVer, used to convert the model into the correct format. While creating a converted model file, properties of the scikit-learn model are extracted, including the value of γ:

```
elif classifier.kernel == 'rbf':
  csv_writer.writerow(['rbf', classifier.gamma])
```

Listing 6.2: Extraction of the gamma value from the model during conversion

The γ value is an important part of the rbf kernel formula:

$$K(x, x') = e^{-\gamma||x-x'||^2}$$

The value of γ is a parameter factor for controlling the influence of examples [27]. The code in the `classifier_mapper.py` does not correctly insert the needed γ value, but instead inserts the type of γ used during training. Instead of a numerical value, the strings 'scale' or 'auto' are inserted if these options are used in scikit-learn.
A further calculation would be needed to compute the value for γ using these formulas:

$$\gamma_{scale} = \frac{1}{n_{features} * X.var()} \gamma_{auto} = \frac{1}{n_{features}}$$

The issue is submitted to the repository of the authors[29] along with a suggestion for a possible workaround. The author has announced a fix of this bug for the future, but at the time of this writing, the robustness tool is not applicable to the model.

6.3.3 Energy Consumption Evaluation

The evaluation of the energy consumption is generated and executed successfully. As the size of the dataset is comparatively small and the computational complexity of training an SVM is limited, the resulting values are also rather modest.
A total power consumption of 7.89 Wh is recorded. This is evaluated to be equivalent to about 2.37 g of carbon dioxide emissions. The accuracy of this evaluation is not proven, as it is dependent on the measuring capabilities of the used system. The full results are given in the appendix.

6.4 Experiment 2: SVR on California Housing

To show the possibilities of evaluating other tasks, a scikit-learn SVR model is trained to solve a regression task on the California Housing dataset [74]. This dataset contains some challenges that make some preprocessing necessary.

[29] `https://github.com/abstract-machine-learning/data-collection/issues/1`

The column `ocean_proximity` does not contain numerical values but forms a categorical feature instead. Additionally, the column `total_bedrooms` contains missing values. Different methods for imputation of missing values exist [94].
As this work is not focused on the development of optimized models, and the number of affected examples is comparatively small with 207 out of 20 640 total examples, the rows with a missing value are omitted. With the same background, the column `ocean_proximity` is omitted, even though methods for converting the values exist [25].
The SVR model is trained with an RBF kernel on 80% of the normalized data. This model is stored using joblib.
As most of the developed Evaluation Tasks are focused on classification models, only the energy consumption evaluation based on CodeCarbon can be applied to the model. Consequently only this task is generated by the framework when the trained model, dataset and describing metadata are used as inputs.

6.4.1 Energy Consumption Evaluation

The training of this model consumes only a small amount of energy (0.30 Wh), equivalent to 0.09 g of carbon emissions. This is mainly due to the limited complexity of training an SVR model on a dataset of this size.
The emissions would be higher if the training process was executed repeatedly. Resampling techniques like crossvalidation would result in a multiplication of the energy consumption and emissions due to the repetition of training procedures. The full results of this evaluation are given in the appendix.

6.5 Experiment 3: Linear SVC on Ionosphere

The training of the SVC model is achieved similarly to the process in chapter 6.3. The main difference in the training parameters is the selection of a linear kernel.
The Ionosphere dataset [88] is not normalized, so scaling of the features is common practice. This is achieved using the built-in scikit-learn functions of the *sklearn.preprocessing* collection:

```
label_encoder = LabelEncoder()
Y = label_encoder.fit_transform(Y)
X = StandardScaler().fit_transform(X)

# Train-Test split
X_train = X[:traintestcutoff, :]
X_test = X[traintestcutoff:, :]
y_train = Y[:traintestcutoff]
```

```
y_test = Y[traintestcutoff:]

clf = SVC(C=0.5, kernel='linear').fit(X_train,y_train)
# save model
joblib.dump(clf, "joblib_iono.pkl")
```

Listing 6.3: Training of a linear SVM model for classifying the Ionosphere dataset

The different subsets of training data and test data are chosen with a size of 200 elements and 151 elements respectively. The unusually large ratio of test data is selected to get more insights into the evaluation algorithms used on the model.
The produced model is applicable to all implemented Evaluation Tasks, and therefore various aspects can be inspected. The ExGETa framework generates four Evaluation Tasks when applied to the saved model. The outputs of the tasks are presented in the next sections:

6.5.1 Confusion Matrix Evaluation

This Evaluation Task calculates all measures related to the confusion matrix, as presented in chapter 5.1. The script for the computations is implemented in a way that analyses all classes individually. For the binary classification case, this leads to some redundancy in the measures, as for example the true positive rate for one class represents the true negative rate for the other.
This could be optimized but represents a property connected to the task being applicable to classifications with more than two classes. The Evaluation Task produces an output of 60 metrics. As the full output is not relevant for this evaluation, the output is included in the appendix for interested readers. To give an idea on the performance of the model, the overall accuracy score of 92.6% can be used.

6.5.2 Robustness Evaluation

The robustness evaluation is also applicable to the model. As described in chapter 5.2, this task outputs a static set of metrics. These are stored to the MLflow server. The full results are compiled in the appendix.
The most important measure is the conditional robustness, which represents the ratio of correct and robust classifications. The value achieved by the SVM on the Ionosphere dataset is 84.3%. Interesting to note is furthermore that only 12 of the 22 misclassifications are evaluated as robust, representing a ratio of 54.5%. This leads to the deduction that misclassifications are less robust than correct classifications, which might be a desired result.

Test Accuracy exceeds that of a naive most-frequent-class classifier and is higher than 75%.

Test Error is within the bounds.

Probability for the bound holding is 95 percent.

Figure 6.2: Interpretations of the bounds evaluated on the linear SVM trained on the Ionosphere dataset

6.5.3 Energy Consumption Evaluation

The energy consumption of this model is very low due to the small dataset size and comparatively low computational complexity of the linear SVM model. The values for consumed energy, carbon emissions et cetera are close to zero, so a further analysis is not useful.
Anyhow, the evaluation is completed successfully, so the results are part of the appendix. In practice, most models can be expected to have a higher energy output than this very simple example.

6.5.4 Bounds Report Evaluation

The bounds and properties discussed in chapter 5.3 can be evaluated, as the classification task is binary with an SVM using a linear kernel. The ExGETa therefore correctly generates the corresponding Evaluation Task.
Executing this task yields metrics and a report. This report contains metrics along with the interpretations of them shown in figure 6.2. The full generated report is attached in the appendix.
As the computed bounds are rather loose, this result can be taken as a rough indication of a correct implementation and usage of the model. In contrast, the scenario of a user mistakenly using the wrong class labels, and therefore testing the model on data completely antithetical to the training data would achieve a very low accuracy and the report output shown in figure 6.3.

6.6 Experiment 4: Decision Tree on Iris

The Iris dataset [38] is chosen to give an example for a Decision Tree model. The Decision Tree model is trained using the `sklearn.tree.DecisionTreeClassifier`. This implementation is based on the CART concept [16] introduced in chapter 2.2.
The dataset is shuffled to remove any pre-existing ordering. After this, a split is selected to divide the data into $\frac{2}{3}$ training data (100 examples) and $\frac{1}{3}$ test data (50 examples).

Test Accuracy is less or equal to that of a naive most-frequent-class classifier!

This represents a failed classification model!

Test Error exceeds bounds! Probability for the bound holding is 95 percent.

Please check the model and dataset!

Figure 6.3: Interpretations of the bounds evaluated on the linear SVM trained on the Ionosphere dataset using opposing testdata

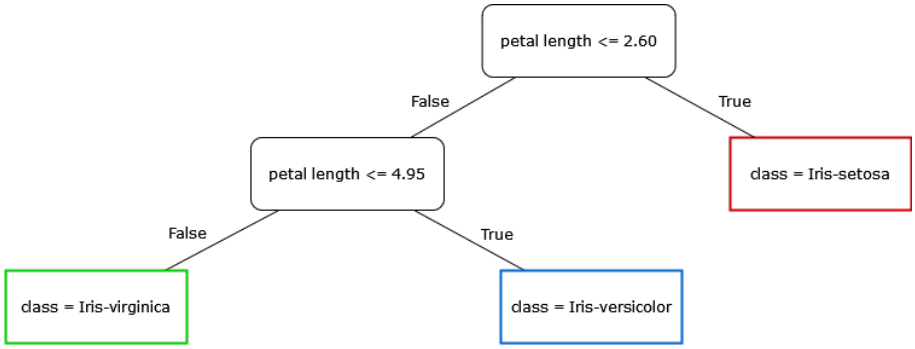

Figure 6.4: Trained Decision Tree based on the Iris dataset limited to a depth of two

The maximum depth of the Decision Tree is set to a very low value of two to acquire a simple and compact model that can be analysed easily. This artificial limitation greatly lowers the performance of the model, but is chosen to enable meaningful evaluations.
The resulting tree can be seen in figure 6.4. Interesting to note in this case is that only one feature is used for the classification. This leads to the observation that the petal length seems to be highly useful for discriminating between the three plant species.
The ExGETa framework can generate two of the implemented tasks. The robustness evaluation is not available because the underlying SAVer tool only supports the evaluation of SVM models [82]. The bounds discussed in chapter 5.3 are also only implemented for SVM models. The correct generation and execution of the other evaluations however shows the capability for evaluation methods for other model types to be integrated into the framework.

6.6.1 Confusion Matrix Evaluation

For the Decision Tree model trained on the Iris dataset the confusion matrix evaluation produces 78 metrics as output. These are all stored correctly in the MLflow tracking database.
An analysis of the results yields the finding that all examples of class Iris-setosa in the test data are correctly identified. This can be seen in the accuracy for this class, which is 100%

and the values for the true positive rate and true negative rate, which are also both 100%. In contrast to this, there are some misclassifications between class Iris-versicolor and Iris-virginica. Four examples of the test data are incorrectly classified as Iris-versicolor and one example is incorrectly classified as Iris-virginica. This results in a true positive rate of class Iris-versicolor of 94.7% and a true positive rate of 77.8% for class Iris-virginica. As the misclassifications only happen between these classes, the accuracy of both classes is 90%.
When interpreting these results and combining them with the knowledge of the tree, one could come to the conclusion that the leafs on the left side could be optimized with a possible further split.

6.6.2 Energy Consumption Evaluation

Similarly to the linear SVC model evaluated in chapter 6.5, the Decision Tree model of this experiment is very simple and the dataset only a low number of examples (150 total). The evaluation of the energy consumption therefore also yields no useful measures.
It could however be noted, that the difference between this experiment and a more complex task like the classification of MNIST data (see chapter 6.3) is immediately visible. While the energy consumption of the Decision Tree model can hardly be measured, the training of the SVM on the MNIST data (70 000 images) does consume an observable amount of energy.

6.7 Assessment of the Implementations

The assessment of the implemented software consists of two main elements. Firstly it is evaluated if and in what way the defined necessary elements are implemented. To reiterate, the main aspects that needed to be developed are:

1. Development of a framework for Evaluation Tasks
    (a) Creation of a database for evaluation software
    (b) Conceptualization of standardized formats and processes
    (c) Assignment of stored evaluation software to given inputs
    (d) Generation of individually executable Evaluation Task packets dependent on the given collection of software
    (e) Provision of a central platform for uploading, downloading and viewing evaluated outputs

2. Creation of an initial set of Evaluation Tasks
   (a) Exemplary implementation of standalone evaluation software
   (b) Integration of existing evaluation tools

After these aspects are evaluated, the quality of the developed software is rated in a more standardized way. This ensures a more impartial rating of the implementations.

6.7.1 Achieved Goals

The assessment of the formulated goals is split into two parts. In the first part, the capabilities of the framework are analysed, and the usefulness is evaluated for ML developers. The second part is focused on the Evaluation Tasks that are implemented and what they represent for the framework.

ExGETa Framework

The database is created using MongoDB (see chapter 4.1). In order to store large software, the GridFS specification is used. The metadata for each task is stored in a separate collection from the large program chunks for a better usability and more efficient queries. This achieves the subgoal (1a), the creation of a database for evaluation software.
Standardized formats are specified for both the file structure of Evaluation Tasks and the structure of the programs (see chapter 4.4). Together with the fixed use of Anaconda and MLflow for an easier integration and management the subgoal (1b), conceptualization of standardized formats and processes, can be considered met.
Queries are used to filter contents of the MongoDB database resulting in the exact set of Evaluation Tasks suitable for the given inputs. This fulfils the goal (1c) of a system for the assignment of software to given inputs.
A python script unzips and combines the retrieved evaluation software into independent Evaluation Task packets. These packets can be executed individually and each provide one or more metrics for the evaluation of the inputs, therefore meeting the requirements of subgoal (1d).
The ML management tool MLflow is used to provide the platform for interaction with the output metrics. The MLflow tracking service gives users the ability to store and view the metrics that have been evaluated. The integration of this service therefore meets the requirements for subgoal (1e).

Evaluation Tasks

To prove the usability of the framework, several Evaluation Tasks are implemented for integration. These are described in detail in chapter 5. Some standalone programs are implemented and integrated, these are the basic confusion matrix evaluation (section 5.1)

and the evaluation of theoretical bounds and properties (section 5.3). These serve as examples for possible additional standalone evaluation techniques that can be implemented for the framework. The subgoal (2a) can therefore be considered fulfilled.
Existing tools for evaluation need to be able to be included into the framework. To analyse the possibilities, wrappers for the tools SAVer (section 5.2) and CodeCarbon (section 5.4) are added to the database of the framework. This shows, that the integration of existing tools is possible and easy for common software, achieving goal (2b).

6.7.2 Software Quality

The quality of the developed software is evaluated loosely following the criteria of the ISO/IEC 25010 [17] standard. Each analysed criteria is covered in an individual section based on the subcriteria.

Functional Suitability

The main function the ExGETa framework should fulfil is the automated generation of Evaluation Tasks for given input data. This function is implemented for exemplary models and evaluation techniques.
The aim of this work is not the development of a fully functional framework for the evaluation of all possible models but rather to form a baseline for future collaborative research. Therefore the function desired by the user (automated evaluation of a comprehensive set of models) is not completely implemented yet.
All implemented elements of the framework and the exemplary Evaluation Tasks are however correct in their function, as shown in the experiments. An exception to this is the robustness evaluation generated for the MNIST SVM (see chapter 6.3), which cannot be executed due to a bug in the underlying tool.
This shows a vulnerability of the framework, as it is only as good as the integrated Evaluation Tasks. If the tasks or underlying implementations contain mistakes, the evaluations may also be incorrect or not executable at all.
Overall the functional suitability of the framework is given, however it is so far constrained to the specified conditions and limits of this work. Further research and expansions are needed for a framework capable of fulfilling function of a universal evaluation tool.

Performance Efficiency

The performance efficiency of the framework itself is very good. The resources needed for the generation of tasks is minimal and the time needed for the full generation process is in the range of one to two seconds on a common consumer laptop.
The framework does however require a database for storage of the Evaluation Tasks (MongoDB cloud storage is used in this work). This database also does not require extensive

computational resources with the full size of the data not exceeding 20 MB with the set of implemented Evaluation Tasks.
The framework has the capacity for considerably more evaluations and procedures to be integrated. The database technology is chosen to be performant and expandable in the future. All software is created modular and capable for higher workloads than present at the current state of the framework. The capacity for future integration of more complex evaluations is therefore given.

Compatibility

The compatibility of the framework is defined mainly by the compatibility of the individual Evaluation Tasks. The framework itself can share a common environment with other software without problems as no low-level hardware commands are used.
The ExGETa framework can be integrated into a higher workflow structure. To this end, the tool is interoperable with it's inputs and execution command, the modular execution of generated Evaluation Tasks and the outputs of the framework. The outputs represent the biggest example for interoperability because the usage of the existing MLflow tracking software enables the simple use of the MLflow API for accessing and working with the output metrics and data.
The Evaluation Tasks themselves are compatible with the execution environment specified in the requirements. At this time, tasks cannot exchange information between each other, as the focus is on allowing a distributed execution of the Evaluation Tasks independently of each other.

Usability

The focus in this thesis is on the conception and implementation of the baseline framework. The usability of this tool can still be improved upon.
As the framework is newly developed, no tutorials or demonstrations other than what is discussed in this work exist at this time. To improve this area, further documentation and guide resources need to be realized.
The ExGETa framework does contain a basic user error protection, so if the tool is executed with wrong types of arguments or the characteristics of the inputs are not described correctly in the metadata file, the framework will exit with an error instead of generating wrong results. The execution of the generated Evaluation Tasks is simplified to the point of the standardized execution of a single script file, where not many errors can be made.
The user interface of the developed program is not very sophisticated. No graphical interfaces exist at this point other than the output interface based on MLflow. This does however provide a good platform for viewing and interacting with the evaluation results for the user.

Reliability

The framework is tested to be very reliable, working without errors for a diverse set of experiments without downtime. This is mainly due to the usage of established technologies like MongoDB that form the basis of the framework.
The ExGETa tool is fault tolerant in the sense that errors in the implementation of one Evaluation Task do not affect the execution or results of other Evaluation Tasks. This is shown in the experiment using the MNIST data, where the robustness could not be evaluated due to an error in the underlying SAVer robustness tool (see chapter 6.3).

Security

The security of the framework is dependant on the configuration of the MongoDB database and MLflow tracking server. Both of these storages have the capabilities for confidentiality, accountability and authenticity [28, 84]. The management of access control is therefore up to the person setting up these resources.
The degree of non-repudiation is also acceptable, as the results of all evaluations are logged to the tracking server with information about the system and time of execution.

Maintainability

As the goal of this work is the development of a baseline framework, a focus in all elements is on possibilities for future improvements, expansions and modifications. The Evaluation Tasks are therefore implemented completely modularly so that new tasks can be integrated easily and existing ones can be modified or improves without influencing others.
The framework itself is also ready for upcoming changes. All of the software is openly available and not obfuscated. The steps of the generation of Evaluation Tasks can be improved upon to introduce more sophisticated techniques for areas like execution on different hardware.

Portability

The framework itself is designed to be highly portable. The tool can be executed on any given machine without the need for an extensive setup. This preparation for execution of the program is even shorter if the user has already realized a database for the Evaluation Tasks and a running MLflow tracking server exists. Then the user only has to pass the path to these resources to the framework to integrate it into the existing systems.
The Evaluation Tasks are also portable, as they use conda files for managing requirements and dependencies. This allows for a high degree of efficiency in the installation and adaptation of the tasks for new hardware. No software has to be manually installed or adapted by the user to execute the evaluation procedures.

Chapter 7

Discussion

In this final chapter, an overview of the developed concepts and software is given. The findings of the evaluation are summarised and interpreted. Lastly an outlook on possible future modifications, improvements and additions is given, as the developed framework only represents a baseline for future collaborative research.

7.1 Results

In this work, a concept for a framework for automated generation of Evaluation Tasks for machine learning models was developed. This abstract concept forms the answer to the formulated research question: *What elements are needed for an automated generation of Evaluation Tasks for ML models?*
The identified elements (see chapter 3.1) have then been implemented in the ExGETa framework on a basic level. These are a database for evaluations and their applicability information, a sound concept for inputs with a software that can match these with the available evaluations, the implementation of a generation of individually executable Evaluation Tasks and the provision of a central result storage. The implementation of all relevant parts was confirmed in chapter 6.7.1. This evaluation of the framework has also confirmed the correctness and completeness of the identified elements.
Multiple Evaluation Tasks with the focus on different aspects of the model have been drafted and implemented. These Evaluation Tasks have proven the ability of the framework to integrate multiple evaluations and correctly aggregate the results.
A set of different models has been trained and applied to the framework. These experiments have shown that the developed software is capable of evaluating various aspects for different models.
The applicability of individual Evaluation Tasks is also being checked correctly by the developed software. To this end, a database technology was chosen and realized to efficiently handle queries regarding the stored evaluations.

The framework is in the current state focused on SVM models and lacks the support for many common models, for example Neural Networks. This limitation was accepted from the start, as a fully functional and complete framework would have exceeded the scope of one thesis. Future expansions can broaden the capabilities in this regard.
Within the bounds set at the start, a functional baseline has been created for future collaborative research to built upon. All parts of the ExGETa framework have been developed to be modular and open. Possible additions for the future are detailed in the next section.

7.2 Future Work

In order to allow future additions and modifications on the developed software, all of the files and code are made available in a public repository[30]. The repository is set up to allow for a simple quickstart with minimal effort. To this end, a readme file containing information about the usage of the framework is provided.
The developed framework is in a baseline state, so there are copious possibilities for future work. Some examples and suggestions are presented.
The first obvious area of expansion is the set of Evaluation Tasks. A fully functional framework for evaluating ML models is dependent on the availability of tests for any model that may be used as an input. As the scope of this work was focused around SVM models, evaluation methods and checks for theoretical properties for other models, like Neural Networks, could be added. To achieve this, the Evaluation tasks need to be created according to the specifications along with metadata about their applicability. All of this then needs to be stored in the database. It is possible that the framework software would need to be modified to allow other models.
The efficiency of the execution of Evaluation Tasks could also be optimized. The ExGETa framework is set up to generally enable a distributed execution of the evaluations, but no method for this has been implemented yet. The individual Evaluation Tasks could be executed on different nodes in a network which would require a method for assigning nodes to the Evaluation Tasks. The result storage is already implemented in a way that allows this because the path to the storage server is already part of the metadata the evaluations are provided with.
Another area for future work is the integration of specific hardware. The integration for GPU specific executions is needed due to the common usage of these processors as accelerators specifically for Deep Learning [69]. Beyond this, an integration of Field Programmable Gate Arrays (FPGAs) is conceivable. This type of hardware entails specific challenges for the implementations, but has also been used for accelerating Deep Learning [12]. Other types of algorithms have also been adapted for execution on FPGAs, for example the training of SVMs [1].

[30] `https://github.com/FDillk/ExGETa`

The evaluation of ML models is closely connected to the optimization of hyperparameters. This describes the process of modifying the configuration of the training parameters of a model to achieve optimal evaluation results [10]. The needed repeated evaluations could be executed using the ExGETa framework.
During the experiments and assessment of the framework, some aspects have been identified as being challenging. A bug in an underlying tool that was used to evaluate robustness has shown that the framework can always only be as reliable as the Evaluation Tasks in the database (see chapter 6.3.2). Mistakes in the theory or implementation of the evaluations will lead to an incomplete or incorrect set of results. The CodeCarbon robustness tool has shown a similar problem highlighting this issue (see chapter 5.4.2). Future work must take these problems into account during development of new features.
Lastly the developed framework could be integrated into the workflow of another software. Due to the simple API for passing inputs and the standardized location of result data, the framework can be included in a more extensive process. This process could include the training of models or a more sophisticated generation of a visualization of the model properties. Using the outputs of the evaluations from this framework, the generation of certifications like the ones from the Care Label concept [72] can be simplified.
Overall many areas for expansion and optimization exist. The modular and open development of the framework enable a future collaborative research effort in the area of automated evaluations with a focus on ML models.

List of Figures

2.1 Example of a Decision Tree based on the Iris dataset 8
2.2 Structure of a confusion matrix for the binary case 10

3.1 Full workflow of using the framework to evaluate a model 13

5.1 MLflow visualization of the outputs of the energy consumption Evaluation Task 52
5.2 Generated Dash application with outputs of the CodeCarbon energy consumption Evaluation Task (part 1) 54
5.3 Generated Dash application with outputs of the CodeCarbon energy consumption Evaluation Task (part 2) 55

6.1 Software Quality Model according to ISO/IEC 25010:2011 58
6.2 Interpretations of the bounds evaluated on the linear SVM trained on the Ionosphere dataset 65
6.3 Interpretations of the bounds evaluated on the linear SVM trained on the Ionosphere dataset using opposing testdata 66
6.4 Trained Decision Tree based on the Iris dataset limited to a depth of two . . 66

Bibliography

[1] Afifi, Shereen, Hamid GholamHosseini and Roopak Sinha: *FPGA Implementations of SVM Classifiers: A Review.* SN Computer Science, 1(3):133, April 2020.

[2] Aizerman, M. A., E. A. Braverman and L. Rozonoer: *Theoretical foundations of the potential function method in pattern recognition learning.* In *Automation and Remote Control*, number 25, pages 821–837, 1964.

[3] Ali, S. and K.A. Smith: *Automatic parameter selection for polynomial kernel.* In *Proceedings of the 5th IEEE Workshop on Mobile Computing Systems and Applications*, pages 243–249, 2003.

[4] Anthony, Martin and Norman Biggs: *PAC Learning and Neural Networks.* In Arbib, Michael A. (editor): *The Handbook of Brain Theory and Neural Networks*, page 840–843. MIT Press, Cambridge, MA, USA, second edition, November 2002.

[5] Aronszajn, Nachman: *Theory of Reproducing Kernels.* Transactions of the American Mathematical Society, 68(3):337–404, 1950.

[6] Baldominos, Alejandro, Yago Saez and Pedro Isasi: *A Survey of Handwritten Character Recognition with MNIST and EMNIST.* Applied Sciences, 9(15), August 2019.

[7] Belinkov, Yonatan and Yonatan Bisk: *Synthetic and Natural Noise Both Break Neural Machine Translation.* In *International Conference on Learning Representations*, 2018.

[8] Bellotti, Victoria and Keith Edwards: *Intelligibility and Accountability: Human Considerations in Context-Aware Systems.* Human-Computer Interaction, 16(2):193–212, December 2001.

[9] Ben-Hur, Asa, David Horn, Hava T. Siegelmann and Vladimir Vapnik: *Support Vector Clustering.* Journal of Machine Learning Research, 2:125–137, November 2001.

[10] Bergstra, James, Rémi Bardenet, Yoshua Bengio and Balázs Kégl: *Algorithms for Hyper-Parameter Optimization.* In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, page 2546–2554, Red Hook, NY, USA, 2011. Curran Associates Inc.

[11] Bezdek, J.C., J.M. Keller, R. Krishnapuram, L.I. Kuncheva and N.R. Pal: *Will the real iris data please stand up?* IEEE Transactions on Fuzzy Systems, 7(3):368–369, 1999.

[12] Blaiech, Ahmed Ghazi, Khaled Ben Khalifa, Carlos Valderrama, Marcelo A.C. Fernandes and Mohamed Hedi Bedoui: *A Survey and Taxonomy of FPGA-Based Deep Learning Accelerators.* Journal of Systems Architecture, 98(C):331–345, September 2019.

[13] Bochinski, Erik, Tobias Senst and Thomas Sikora: *Hyper-Parameter Optimization for Convolutional Neural Network Committees Based on Evolutionary Algorithms.* In *2017 IEEE International Conference on Image Processing (ICIP)*, page 3924–3928. IEEE Press, 2017.

[14] Boser, Bernhard E., Isabelle M. Guyon and Vladimir N. Vapnik: *A Training Algorithm for Optimal Margin Classifiers.* In *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, pages 144–152. ACM Press, 1992.

[15] Breiman, Leo: *Random Forests.* Machine Learning, 45(1):5–32, October 2001.

[16] Breiman, Leo, Jerome H. Friedman, Richard A. Olshen and Charles J. Stone: *Classification and Regression Trees.* Chapman & Hall, Boca Raton, FL, USA, 1984.

[17] BSI: *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models: Country code (BS ISO/IEC 25010:2011).* Standard, British Standards Institution, London, UK, March 2011.

[18] Buitinck, Lars, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt and Gaël Varoquaux: *API design for machine learning software: experiences from the scikit-learn project.* In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[19] Bunse, Mirko and Katharina Morik: *Certification of Model Robustness in Active Class Selection.* In *Proceedings of the European Conference on Machine Learning*

and Principles and Practice of Knowledge Discovery in Databases. ECML PKDD 2021, Part II, page 266–281, Berlin, Heidelberg, September 2021. Springer-Verlag.

[20] Burges, Christopher J. C.: *A Tutorial on Support Vector Machines for Pattern Recognition.* Data Mining and Knowledge Discovery, 2(2):121–167, June 1998.

[21] Cao, Yulong, Chaowei Xiao, Benjamin Cyr, Yimeng Zhou, Won Park, Sara Rampazzi, Qi Alfred Chen, Kevin Fu and Z. Morley Mao: *Adversarial Sensor Attack on LiDAR-Based Perception in Autonomous Driving.* In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2267–2281, New York, NY, USA, 2019. ACM.

[22] Carlini, Nicholas, Pratyush Mishra, Tavish Vaidya, Yuankai Zhang, Micah Sherr, Clay Shields, David Wagner and Wenchao Zhou: *Hidden Voice Commands.* In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 513–530, USA, 2016. USENIX Association.

[23] Carlini, Nicholas and David Wagner: *Towards Evaluating the Robustness of Neural Networks.* In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57, Los Alamitos, CA, USA, May 2017. IEEE.

[24] CEN: *Codes for the representation of names of countries and their subdivisions - Part 1: Country code (ISO 3166-1:2020).* Standard, European Committee for Standardization, Brussels, BE, September 2020.

[25] Cerda, Patricio, Gaël Varoquaux and Balázs Kégl: *Similarity encoding for learning with dirty categorical variables.* Machine Learning, 107(8):1477–1494, September 2018.

[26] Chang, Chih-Chung and Chih-Jen Lin: *LIBSVM: A Library for Support Vector Machines.* ACM Transactions on Intelligent Systems and Technology, 2(3), May 2011.

[27] Chang, Qun, Qingcai Chen and Xiaolong Wang: *Scaling Gaussian RBF kernel width to improve SVM classification.* In *2005 International Conference on Neural Networks and Brain*, volume 1, pages 19–22, 2005.

[28] Chen, Andrew, Andy Chow, Aaron Davidson, Arjun DCunha, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Clemens Mewald, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Avesh Singh, Fen Xie, Matei Zaharia, Richard Zang, Juntai Zheng and Corey Zumar: *Developments in MLflow: A System to Accelerate the Machine Learning Lifecycle.* In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*, DEEM'20, New York, NY, USA, 2020. ACM.

[29] Chen, Tianqi and Carlos Guestrin: *XGBoost: A Scalable Tree Boosting System.* In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[30] Contreras, Gilberto and Margaret Martonosi: *Power Prediction for Intel XScale® Processors Using Performance Monitoring Unit Events.* In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISLPED '05, page 221–226, New York, NY, USA, August 2005. Association for Computing Machinery.

[31] Cortes, Corinna and Vladimir Vapnik: *Support-Vector Networks.* Machine Learning, 20(3):273–297, September 1995.

[32] Davoudian, Ali, Liu Chen and Mengchi Liu: *A Survey on NoSQL Stores.* ACM Computing Surveys, 51(2), April 2018.

[33] Dignum, Virginia: *Responsible Artificial Intelligence: How to Develop and Use AI in a Responsible Way.* Springer Publishing Company, Inc., 1st edition, 2019.

[34] Drucker, Harris, Chris J. C. Burges, Linda Kaufman, Alex Smola and Vladimir N. Vapnik: *Support Vector Regression Machines.* In *Proceedings of the 9th International Conference on Neural Information Processing Systems*, NIPS'96, page 155–161, Cambridge, MA, USA, 1996. MIT Press.

[35] Fan, Rong-En, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang and Chih-Jen Lin: *LIBLINEAR: A Library for Large Linear Classification.* Journal of Machine Learning Research, 9:1871–1874, June 2008.

[36] Fawcett, Tom: *An Introduction to ROC Analysis.* Pattern Recognition Letters, 27(8):861–874, June 2006.

[37] Filip, Petr and Lukáš Čegan: *Comparison of MySQL and MongoDB with focus on performance.* In *2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, pages 184–187. IEEE, 2020.

[38] Fisher, Ronald Aylmer: *The Use of Multiple Measurements in Taxonomic Problems.* Annals of Eugenics, 7(2):179–188, 1936.

[39] García, V., Ramon A. Mollineda and J. Salvador Sánchez: *Theoretical Analysis of a Performance Measure for Imbalanced Data.* In *20th International Conference on Pattern Recognition*, pages 617–620, 2010.

[40] García-Martín, Eva, Crefeda Faviola Rodrigues, Graham Riley and Håkan Grahn: *Estimation of Energy Consumption in Machine Learning.* Journal of Parallel and Distributed Computing, 134(C):75–88, December 2019.

[41] Garg, Satvik, Pradyumn Pundir, Geetanjali Rathee, P.K. Gupta, Somya Garg and Saransh Ahlawat: *On Continuous Integration / Continuous Delivery for Automated Deployment of Machine Learning Models using MLOps.* In *2021 IEEE Fourth International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*, pages 25–28, 2021.

[42] Gu, Yunhua, Xing Wang, Shu Shen, Jin Wang and Jeong-Uk Kim: *Analysis of data storage mechanism in NoSQL database MongoDB.* In *2015 IEEE International Conference on Consumer Electronics - Taiwan*, pages 70–71, 2015.

[43] Hähnel, Marcus, Björn Döbel, Marcus Völp and Hermann Härtig: *Measuring Energy Consumption for Short Code Paths Using RAPL.* SIGMETRICS Perform. Eval. Rev., 40(3):13–17, January 2012.

[44] Hall, Mark, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann and Ian H. Witten: *The WEKA Data Mining Software: An Update.* SIGKDD Explorations Newsletter, 11(1):10–18, November 2009.

[45] Hao, Jiangang and Tin Kam Ho: *Machine Learning Made Easy: A Review of Scikit-learn Package in Python Programming Language.* Journal of Educational and Behavioral Statistics, 44(3):348–361, 2019.

[46] Harrison, David and Daniel L Rubinfeld: *Hedonic housing prices and the demand for clean air.* Journal of Environmental Economics and Management, 5(1):81–102, 1978.

[47] Hsieh, Cho-Jui, Kai-Wei Chang, Chih-Jen Lin, S. Sathiya Keerthi and S. Sundararajan: *A Dual Coordinate Descent Method for Large-Scale Linear SVM.* In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, page 408–415, New York, NY, USA, 2008. ACM.

[48] Ivory, Melody Y. and Marti A Hearst: *The State of the Art in Automating Usability Evaluation of User Interfaces.* ACM Computing Surveys, 33(4):470–516, December 2001.

[49] Jaakkola, Tommi S. and David Haussler: *Probabilistic Kernel Regression Models.* In *Proceedings of the 1999 Conference on AI and Statistics.* Morgan Kaufmann, 1999.

[50] Joachims, Thorsten: *Making Large-Scale Support Vector Machine Learning Practical.* In Schölkopf, B., C. Burges and A. Smola (editors): *Advances in Kernel Methods: Support Vector Learning*, chapter 11, page 169–184. MIT Press, Cambridge, MA, USA, 1999.

[51] Joachims, Thorsten: *Estimating the Generalization Performance of an SVM Efficiently.* In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, page 431–438, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[52] Joachims, Thorsten: *A Statistical Learning Learning Model of Text Classification for Support Vector Machines.* In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '01, page 128–136, New York, NY, USA, 2001. ACM.

[53] Joachims, Thorsten: *Learning to Classify Text Using Support Vector Machines: Methods, Theory and Algorithms.* Kluwer Academic Publishers, Norwell, Massachusetts, USA, 2002.

[54] Joachims, Thorsten: *Training Linear SVMs in Linear Time.* In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, page 217–226, New York, NY, USA, 2006. ACM.

[55] Keerthi, S. S., S. K. Shevade, C. Bhattacharyya and K. R. K. Murthy: *Improvements to Platt's SMO Algorithm for SVM Classifier Design.* Neural Computation, 13(3):637–649, 2001.

[56] Lacoste, Alexandre, Alexandra Luccioni, Victor Schmidt and Thomas Dandres: *Quantifying the Carbon Emissions of Machine Learning.* Workshop on Tackling Climate Change with Machine Learning at NeurIPS 2019, 2019.

[57] LeCun, Yann, Léon Bottou, Yoshua Bengio and Patrick Haffner: *Gradient-based learning applied to document recognition.* Proceedings of the IEEE, 86(11):2278–2324, November 1998.

[58] Leisch, Friedrich and Evgenia Dimitriadou: *mlbench: Machine Learning Benchmark Problems*, January 2021. R package version 2.1-3.

[59] Li, Da, Xinbo Chen, Michela Becchi and Ziliang Zong: *Evaluating the Energy Efficiency of Deep Convolutional Neural Networks on CPUs and GPUs.* In *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*, pages 477–484, Atlanta, GA, USA, October 2016. IEEE.

[60] Lin, Hsuan-Tien and Chih-Jen Lin: *A Study on Sigmoid Kernels for SVM and the Training of non-PSD Kernels by SMO-type Methods*. Neural Computation, June 2003.

[61] Loh, Wei-Yin: *Improving the precision of classification trees*. The Annals of Applied Statistics, 3(4):1710–1737, December 2009.

[62] Loh, Wei-Yin: *Classification and Regression Trees*. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 1:14 – 23, January 2011.

[63] Lottick, Kadan, Silvia Susai, Sorelle A. Friedler and Jonathan P. Wilson: *Energy Usage Reports: Environmental awareness as part of algorithmic accountability*. Workshop on Tackling Climate Change with Machine Learning at NeurIPS 2019, 2019.

[64] Lundberg, Scott M. and Su-In Lee: *A Unified Approach to Interpreting Model Predictions*. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 4768–4777, Red Hook, NY, USA, 2017. Curran Associates Inc.

[65] Markidis, Stefano, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng and Jeffrey S. Vetter: *NVIDIA Tensor Core Programmability, Performance and Precision*. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531, May 2018.

[66] McAfee, Andrew, Erik Brynjolfsson, Thomas H Davenport, DJ Patil and Dominic Barton: *Big data: the management revolution*. Harvard business review, 90(10):60–68, 2012.

[67] Meier, Andreas and Michael Kaufmann: *SQL & NoSQL Databases: Models, Languages, Consistency Options and Architectures for Big Data Management*, chapter Data Management, page 6. Springer Vieweg Wiesbaden, Wiesbaden, 2019.

[68] Mercer, James: *Functions of Positive and Negative Type, and their Connection with the Theory of Integral Equations*. Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character, 209:415–446, 1909.

[69] Mittal, Sparsh and Shraiysh Vaishay: *A Survey of Techniques for Optimizing Deep Learning on GPUs*. Journal of Systems Architecture, 99(C), October 2019.

[70] Moolayil, Jojo: *Learn Keras for Deep Neural Networks*, chapter 2, pages 17–52. Springer, Vancouver, CA, 2019.

[71] Moore, Charles, Sarah Brown, Phil MacDonald, Matt Ewen and Hannah Broadbent: *European Electricity Review 2022*. Technical Report, Ember, London, UK, February 2022.

[72] Morik, Katharina, Helena Kotthaus, Lukas Heppe, Danny Heinrich, Raphael Fischer, Andreas Pauly and Nico Piatkowski: *The Care Label Concept: A Certification Suite for Trustworthy and Resource-Aware Machine Learning*, 2021.

[73] Nascimento, Elizamary de Souza, Iftekhar Ahmed, Edson Oliveira, Márcio Piedade Palheta, Igor Steinmacher and Tayana Conte: *Understanding Development Process of Machine Learning Systems: Challenges and Solutions*. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, 2019.

[74] Pace, R. Kelley and Ronald Barry: *Sparse Spatial Autoregressions*. Statistics & Probability Letters, 33(3):291–297, 1997.

[75] Papernot, Nicolas, Fartash Faghri, Nicholas Carlini, Ian Goodfellow, Reuben Feinman, Alexey Kurakin, Cihang Xie, Yash Sharma, Tom Brown, Aurko Roy, Alexander Matyasko, Vahid Behzadan, Karen Hambardzumyan, Zhishuai Zhang, Yi-Lin Juang, Zhi Li, Ryan Sheatsley, Abhibhav Garg, Jonathan Uesato, Willi Gierke, Yinpeng Dong, David Berthelot, Paul Hendricks, Jonas Rauber, Rujun Long and Patrick McDaniel: *Technical Report on the CleverHans v2.1.0 Adversarial Examples Library*, 2018.

[76] Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai and Soumith Chintala: *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. In Wallach, H., H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox and R. Garnett (editors): *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[77] Patle, Arti and Deepak Singh Chouhan: *SVM kernel functions for classification*. In *2013 International Conference on Advances in Technology and Engineering (ICATE)*, pages 1–9, 2013.

[78] Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Van-

DERPLAS, A. PASSOS, D. COURNAPEAU, M. BRUCHER, M. PERROT and E. DUCHESNAY: *Scikit-learn: Machine Learning in Python.* Journal of Machine Learning Research, 12:2825–2830, 2011.

[79] PLATT, JOHN: *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines.* Advances in Kernel Methods-Support Vector Learning, 208, July 1998.

[80] PLUGGE, EELCO, PETER MEMBREY and TIM HAWKINS: *GridFS.* In *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*, pages 83–95. Apress, Berkeley, CA, 2010.

[81] QUINLAN, J. ROSS: *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, October 1993.

[82] RANZATO, FRANCESCO and MARCO ZANELLA: *Robustness Verification of Support Vector Machines.* In *Static Analysis: 26th International Symposium*, page 271–295, Berlin, Heidelberg, 2019. Springer-Verlag.

[83] RAUBER, JONAS, WIELAND BRENDEL and MATTHIAS BETHGE: *Foolbox: A Python toolbox to benchmark the robustness of machine learning models*, 2018.

[84] SAMANTA, ASHIS KUMAR and NABENDU CHAKI: *Performance Monitoring of MongoDB on Varied Cluster Configuration: An Experimental Approach.* In *2021 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT)*, pages 525–530. IEEE, 2021.

[85] SAMEK, WOJCIECH, GRÉGOIRE MONTAVON, ANDREA VEDALDI, LARS KAI HANSEN and KLAUS-ROBERT MÜLLER (editors): *Explainable AI: interpreting, explaining and visualizing deep learning*, volume 11700. Springer Nature, 2019.

[86] SCHMIDT, VICTOR, KAMAL GOYAL, ADITYA JOSHI, BORIS FELD, LIAM CONELL, NIKOLAS LASKARIS, DOUG BLANK, JONATHAN WILSON, SORELLE FRIEDLER and SASHA LUCCIONI: *CodeCarbon: Estimate and Track Carbon Emissions from Machine Learning Computing.* 2021.

[87] SHALEV-SHWARTZ, SHAI, YORAM SINGER and NATHAN SREBRO: *Pegasos: Primal Estimated Sub-GrAdient SOlver for SVM.* In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, page 807–814, New York, NY, USA, 2007. ACM.

[88] SIGILLITO, VINCENT G, SIMON P WING, LARRIE V HUTTON and KILE B BAKER: *Classification of radar returns from the ionosphere using neural networks.* Johns Hopkins APL Technical Digest, 10(3):262–266, 1989.

[89] Smith, Einar: *Python, the Fundamentals.* In *Introduction to the Tools of Scientific Computing*, pages 19–50. Springer International Publishing, Cham, Switzerland, 2020.

[90] Sokolova, Marina and Guy Lapalme: *A systematic analysis of performance measures for classification tasks.* Information Processing & Management, 45:427–437, July 2009.

[91] Steinwart, I., D. Hush and C. Scovel: *An Explicit Description of the Reproducing Kernel Hilbert Spaces of Gaussian RBF Kernels.* IEEE Transactions on Information Theory, 52(10):4635–4643, 2006.

[92] Szegedy, Christian, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow and Rob Fergus: *Intriguing properties of neural networks.* In *International Conference on Learning Representations*, 2014.

[93] Tharwat, Alaa: *Classification assessment methods.* Applied Computing and Informatics, 17(1):168–192, January 2021.

[94] van Buuren, Stef: *Flexible Imputation of Missing Data.* Chapman & Hall/CRC Interdisciplinary Statistics. CRC Press LLC, Second edition, 2018.

[95] Vapnik, Vladimir N. (editor): *Statistical Learning Theory*, chapter 10, page 401–442. Adaptive and Learning Systems for Signal Processing, Communications and Control. Wiley-Interscience, Chichester, United Kingdom, 1998.

[96] Vapnik, Vladimir N.: *The Nature of Statistical Learning Theory.* Information Science and Statistics. Springer, New York, NY, USA, Second edition, 2000.

[97] Watada, Junzo, Arunava Roy, Ruturaj Kadikar, Hoang Pham and Bing Xu: *Emerging Trends, Techniques and Open Issues of Containerization: A Review.* IEEE Access, 7:152443–152472, October 2019.

[98] Zaharia, Matei, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe et al.: *Accelerating the machine learning lifecycle with MLflow.* IEEE Data Engineering Bulletin, 41(4):39–45, 2018.

[99] Zhang, Jie M., Mark Harman, Lei Ma and Yang Liu: *Machine Learning Testing: Survey, Landscapes and Horizons.* IEEE Transactions on Software Engineering, 48(1):1–36, 2022.

[100] Zügner, Daniel and Stephan Günnemann: *Certifiable Robustness of Graph Convolutional Networks under Structure Perturbations.* In *Proceedings of the 26th*

ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '20, page 1656–1665, New York, NY, USA, August 2020. ACM.

Appendix A

Outputs of the Evaluations

A.1 RBF SVC on MNIST

accuracy_class_0	0.996
accuracy_class_1	0.997
accuracy_class_2	0.99
accuracy_class_3	0.992
accuracy_class_4	0.992
accuracy_class_5	0.992
accuracy_class_6	0.994
accuracy_class_7	0.986
accuracy_class_8	0.991
accuracy_class_9	0.989
accuracy_score	0.959
balanced_accuracy_class_0	0.99
balanced_accuracy_class_1	0.994
balanced_accuracy_class_2	0.976
balanced_accuracy_class_3	0.977
balanced_accuracy_class_4	0.977
balanced_accuracy_class_5	0.97
balanced_accuracy_class_6	0.982
balanced_accuracy_class_7	0.972
balanced_accuracy_class_8	0.97
balanced_accuracy_class_9	0.961
balanced_accuracy_score	0.958
diagnostic_odds_ratio_class_0	21233.2
diagnostic_odds_ratio_class_1	53182.6
diagnostic_odds_ratio_class_2	3331.8

diagnostic_odds_ratio_class_3	5288.1
diagnostic_odds_ratio_class_4	5913.8
diagnostic_odds_ratio_class_5	4534.8
diagnostic_odds_ratio_class_6	9293.7
diagnostic_odds_ratio_class_7	1890
diagnostic_odds_ratio_class_8	3753.7
diagnostic_odds_ratio_class_9	3280.7
f1_score_class_0	0.979
f1_score_class_1	0.988
f1_score_class_2	0.95
f1_score_class_3	0.96
f1_score_class_4	0.961
f1_score_class_5	0.952
f1_score_class_6	0.969
f1_score_class_7	0.931
f1_score_class_8	0.951
f1_score_class_9	0.945
fdr_class_0	0.024
fdr_class_1	0.015
fdr_class_2	0.06
fdr_class_3	0.04
fdr_class_4	0.035
fdr_class_5	0.038
fdr_class_6	0.03
fdr_class_7	0.091
fdr_class_8	0.042
fdr_class_9	0.035
fn_class_0	17
fn_class_1	11
fn_class_2	42
fn_class_3	41
fn_class_4	42
fn_class_5	51
fn_class_6	31
fn_class_7	47
fn_class_8	55
fn_class_9	75
fnr_class_0	0.017
fnr_class_1	0.01

fnr_class_2	0.041
fnr_class_3	0.041
fnr_class_4	0.043
fnr_class_5	0.057
fnr_class_6	0.032
fnr_class_7	0.046
fnr_class_8	0.056
fnr_class_9	0.074
for_class_0	0.002
for_class_1	0.001
for_class_2	0.005
for_class_3	0.005
for_class_4	0.005
for_class_5	0.006
for_class_6	0.003
for_class_7	0.005
for_class_8	0.006
for_class_9	0.008
fowlkes_mallows_index_class_0	0.979
fowlkes_mallows_index_class_1	0.988
fowlkes_mallows_index_class_2	0.95
fowlkes_mallows_index_class_3	0.96
fowlkes_mallows_index_class_4	0.961
fowlkes_mallows_index_class_5	0.952
fowlkes_mallows_index_class_6	0.969
fowlkes_mallows_index_class_7	0.931
fowlkes_mallows_index_class_8	0.951
fowlkes_mallows_index_class_9	0.945
fp_class_0	24
fp_class_1	17
fp_class_2	63
fp_class_3	40
fp_class_4	34
fp_class_5	33
fp_class_6	29
fp_class_7	98
fp_class_8	40
fp_class_9	34
fpr_class_0	0.003

fpr_class_1	0.002
fpr_class_2	0.007
fpr_class_3	0.004
fpr_class_4	0.004
fpr_class_5	0.004
fpr_class_6	0.003
fpr_class_7	0.011
fpr_class_8	0.004
fpr_class_9	0.004
hamming_loss	0.041
informedness_class_0	-1.015
informedness_class_1	-1.008
informedness_class_2	-1.034
informedness_class_3	-1.036
informedness_class_4	-1.039
informedness_class_5	-1.054
informedness_class_6	-1.029
informedness_class_7	-1.035
informedness_class_8	-1.052
informedness_class_9	-1.071
markedness_class_0	0.974
markedness_class_1	0.984
markedness_class_2	0.935
markedness_class_3	0.956
markedness_class_4	0.96
markedness_class_5	0.957
markedness_class_6	0.966
markedness_class_7	0.904
markedness_class_8	0.952
markedness_class_9	0.957
matthews_corrcoef	0.954
matthews_corrcoef_class_0	0.977
matthews_corrcoef_class_1	0.986
matthews_corrcoef_class_2	0.944
matthews_corrcoef_class_3	0.955
matthews_corrcoef_class_4	0.957
matthews_corrcoef_class_5	0.948
matthews_corrcoef_class_6	0.965
matthews_corrcoef_class_7	0.923

matthews_corrcoef_class_8	0.946
matthews_corrcoef_class_9	0.939
neg_likelihood_ratio_class_0	0.017
neg_likelihood_ratio_class_1	0.01
neg_likelihood_ratio_class_2	0.041
neg_likelihood_ratio_class_3	0.041
neg_likelihood_ratio_class_4	0.043
neg_likelihood_ratio_class_5	0.057
neg_likelihood_ratio_class_6	0.032
neg_likelihood_ratio_class_7	0.046
neg_likelihood_ratio_class_8	0.057
neg_likelihood_ratio_class_9	0.075
npv_class_0	0.998
npv_class_1	0.999
npv_class_2	0.995
npv_class_3	0.995
npv_class_4	0.995
npv_class_5	0.994
npv_class_6	0.997
npv_class_7	0.995
npv_class_8	0.994
npv_class_9	0.992
pos_likelihood_ratio_class_0	369.3
pos_likelihood_ratio_class_1	516.4
pos_likelihood_ratio_class_2	136.6
pos_likelihood_ratio_class_3	215.6
pos_likelihood_ratio_class_4	253.9
pos_likelihood_ratio_class_5	260.2
pos_likelihood_ratio_class_6	301.7
pos_likelihood_ratio_class_7	87.37
pos_likelihood_ratio_class_8	212.9
pos_likelihood_ratio_class_9	244.8
ppv_class_0	0.976
ppv_class_1	0.985
ppv_class_2	0.94
ppv_class_3	0.96
ppv_class_4	0.965
ppv_class_5	0.962
ppv_class_6	0.97

ppv_class_7	0.909
ppv_class_8	0.958
ppv_class_9	0.965
prevalence_class_0	9020
prevalence_class_1	8865
prevalence_class_2	8968
prevalence_class_3	8990
prevalence_class_4	9018
prevalence_class_5	9108
prevalence_class_6	9042
prevalence_class_7	8972
prevalence_class_8	9026
prevalence_class_9	8991
pt_class_0	0.049
pt_class_1	0.042
pt_class_2	0.079
pt_class_3	0.064
pt_class_4	0.059
pt_class_5	0.058
pt_class_6	0.054
pt_class_7	0.097
pt_class_8	0.064
pt_class_9	0.06
tn_class_0	8996
tn_class_1	8848
tn_class_2	8905
tn_class_3	8950
tn_class_4	8984
tn_class_5	9075
tn_class_6	9013
tn_class_7	8874
tn_class_8	8986
tn_class_9	8957
tnr_class_0	0.997
tnr_class_1	0.998
tnr_class_2	0.993
tnr_class_3	0.996
tnr_class_4	0.996
tnr_class_5	0.996

tnr_class_6	0.997
tnr_class_7	0.989
tnr_class_8	0.996
tnr_class_9	0.996
tp_class_0	963
tp_class_1	1124
tp_class_2	990
tp_class_3	969
tp_class_4	940
tp_class_5	841
tp_class_6	927
tp_class_7	981
tp_class_8	919
tp_class_9	934
tpr_class_0	0.983
tpr_class_1	0.99
tpr_class_2	0.959
tpr_class_3	0.959
tpr_class_4	0.957
tpr_class_5	0.943
tpr_class_6	0.968
tpr_class_7	0.954
tpr_class_8	0.944
tpr_class_9	0.926
ts_class_0	0.959
ts_class_1	0.976
ts_class_2	0.904
ts_class_3	0.923
ts_class_4	0.925
ts_class_5	0.909
ts_class_6	0.939
ts_class_7	0.871
ts_class_8	0.906
ts_class_9	0.895
zero_one_loss	0.041

Table A.1: Output of the Confusion Matrix Evaluation of the rbf SVC on MNIST

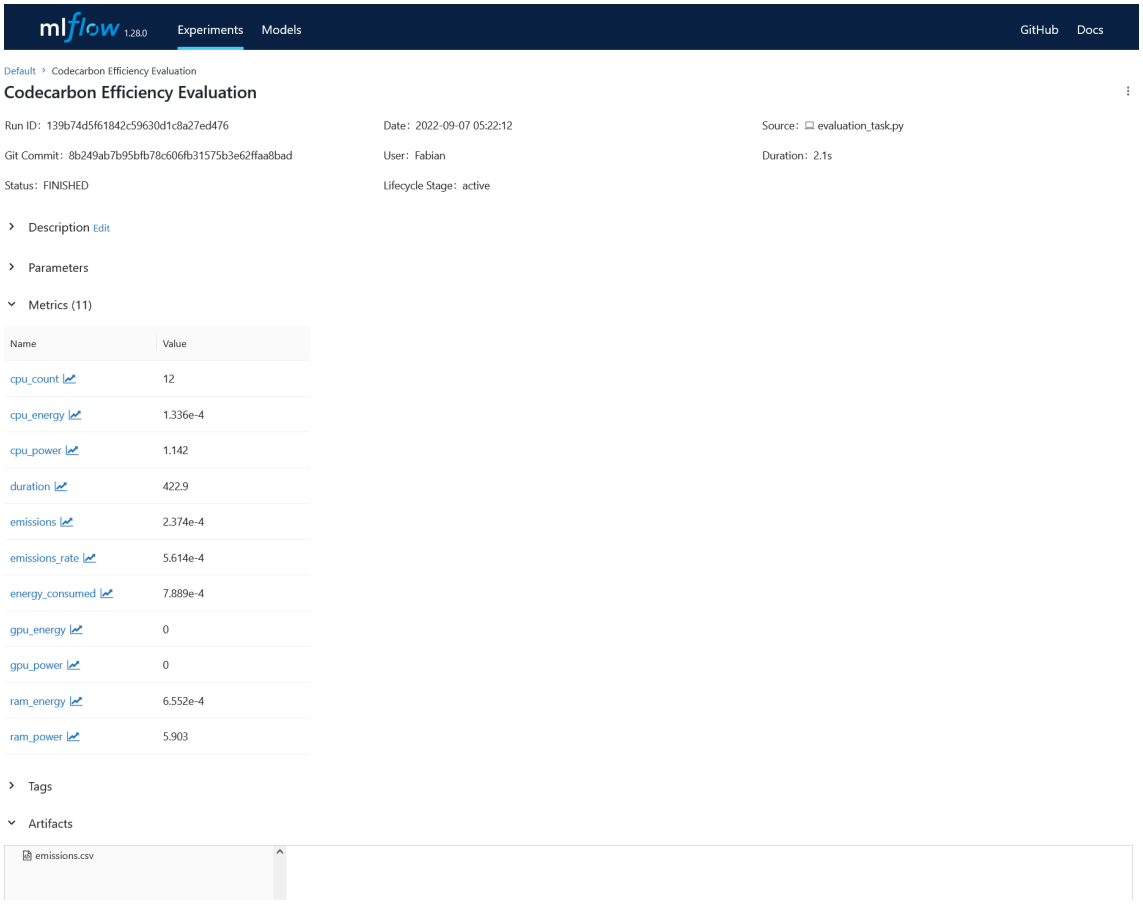

Figure A.1: Output of the Energy Consumption Evaluation of the rbf SVC on MNIST

A.2 RBF SVR on California Housing

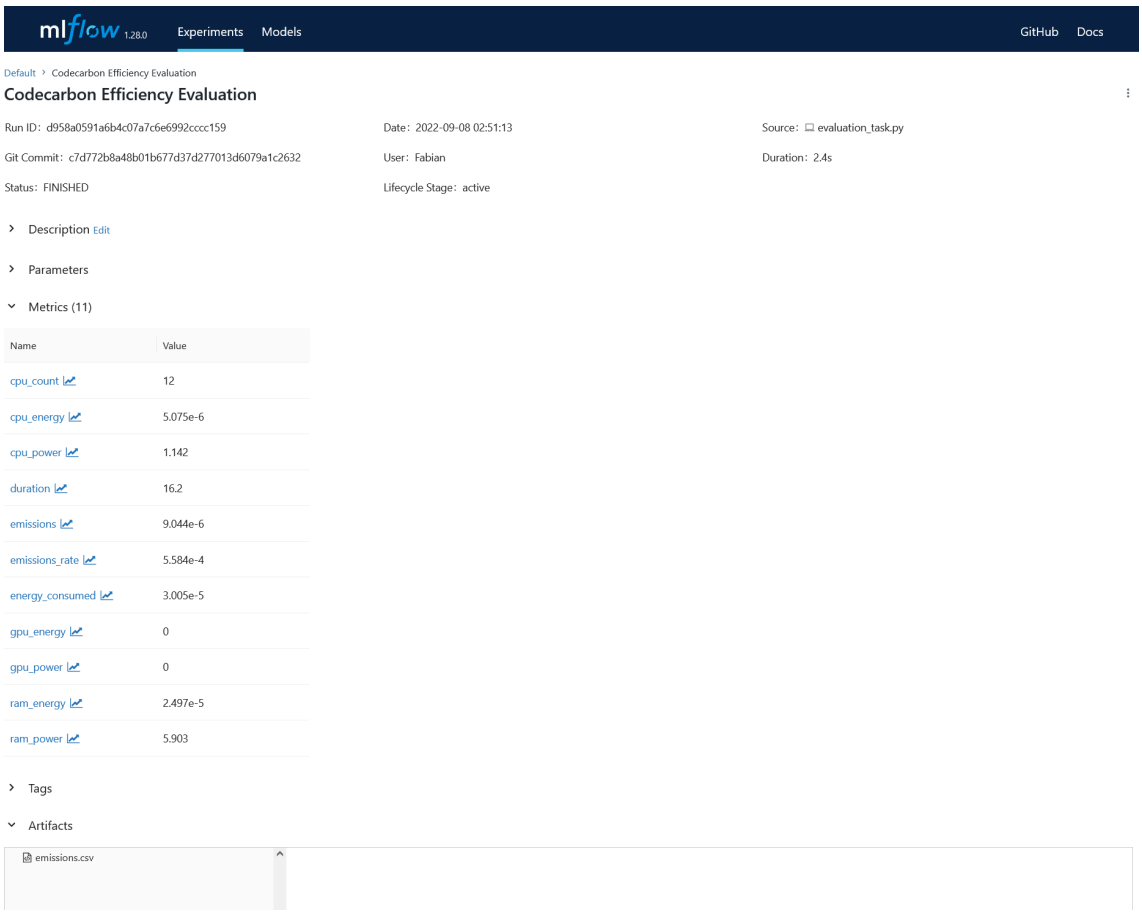

Figure A.2: Output of the Energy Consumption Evaluation of the rbf SVR on California Housing

A.3 Linear SVC on Ionosphere Dataset

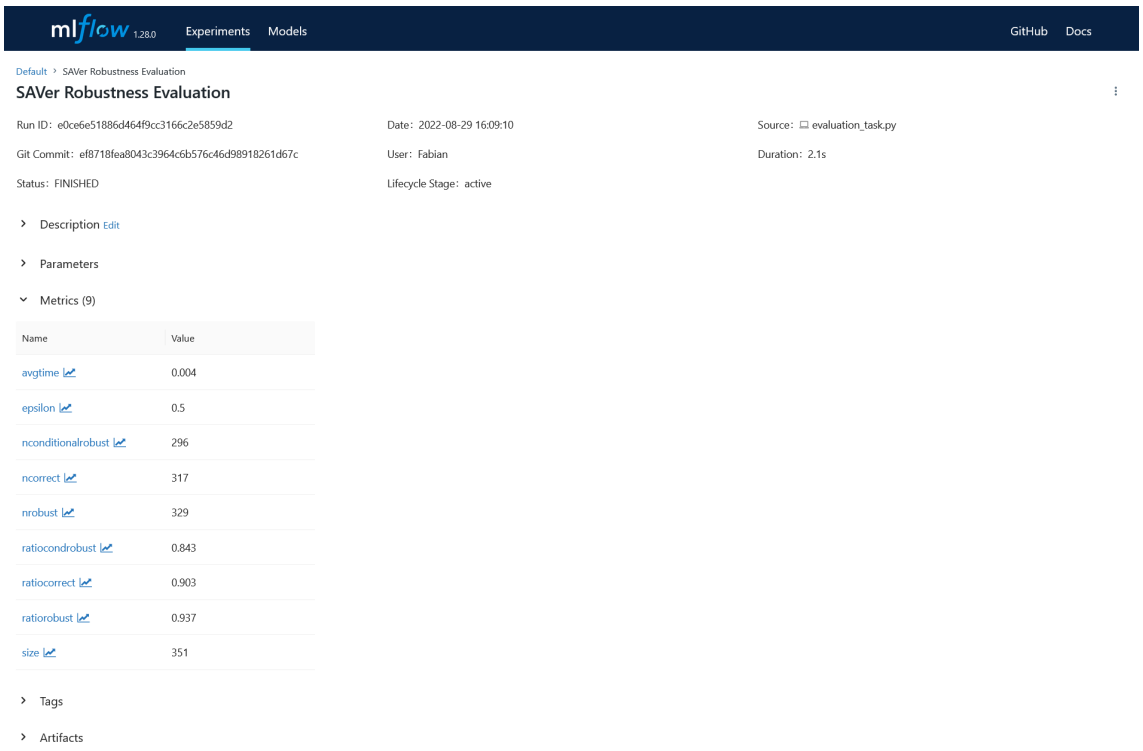

Figure A.3: Output of the Robustness Evaluation of the linear SVC on Ionosphere

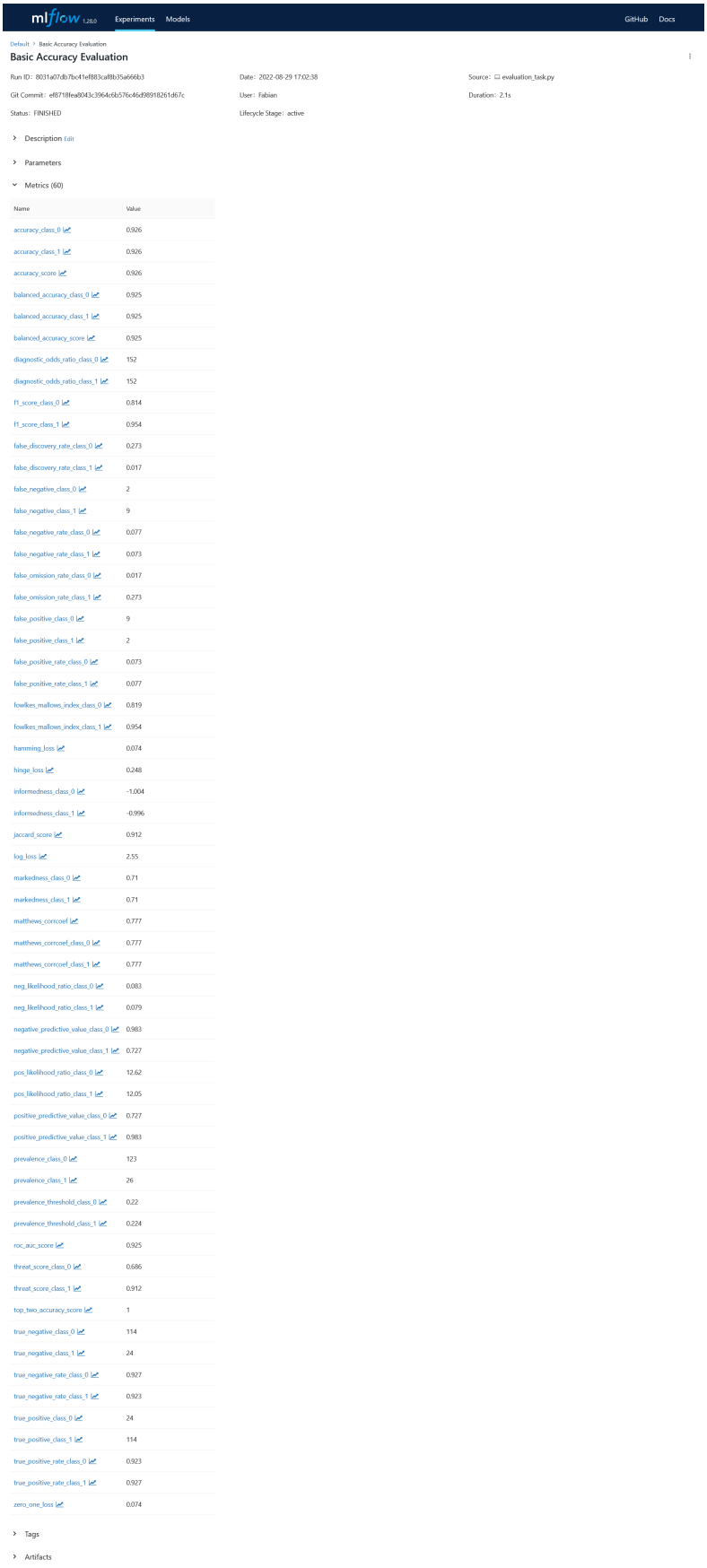

Figure A.4: Output of the Confusion Matrix Evaluation of the linear SVC on Ionosphere

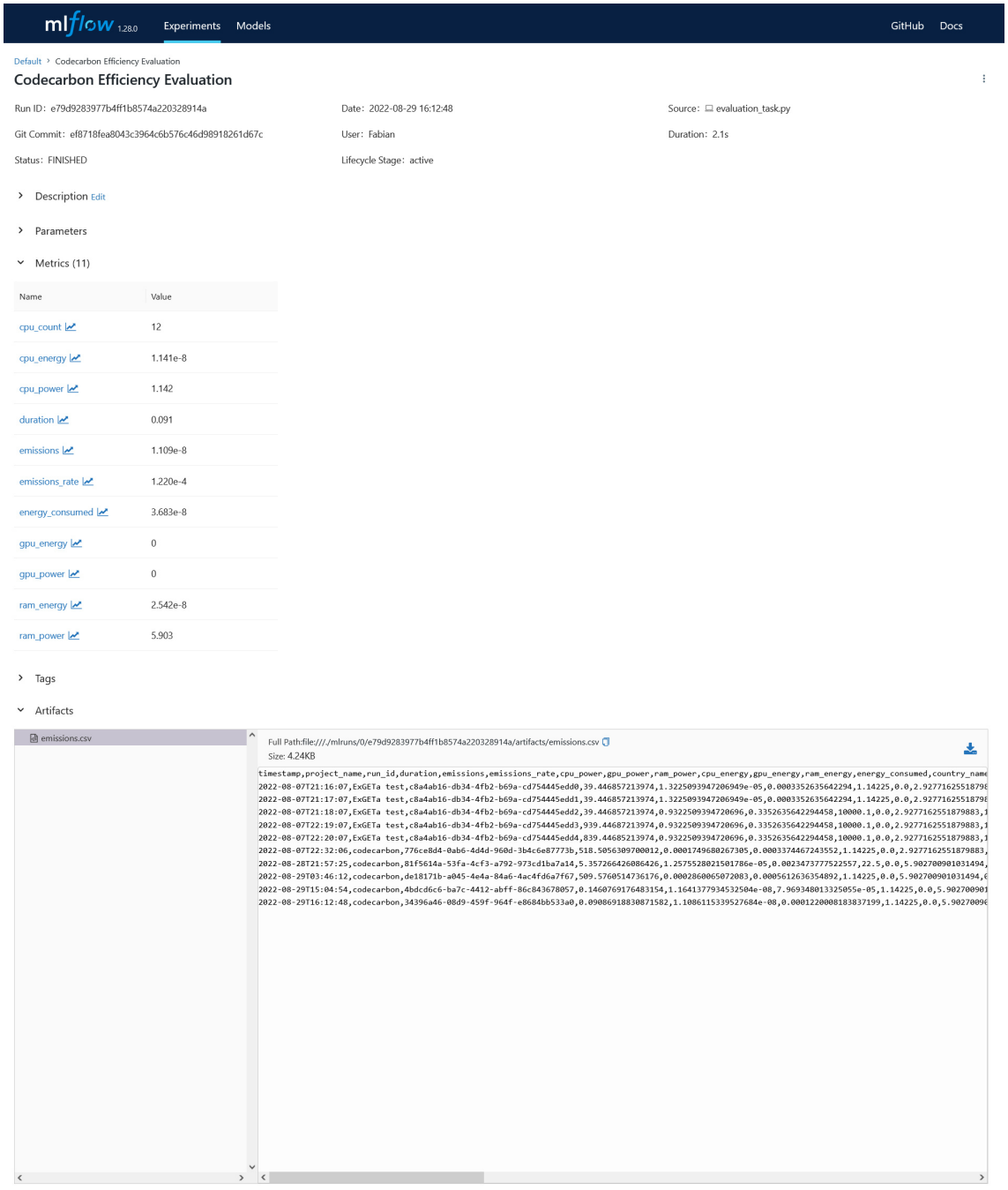

Figure A.5: Output of the Energy Consumption Evaluation of the linear SVC on Ionosphere

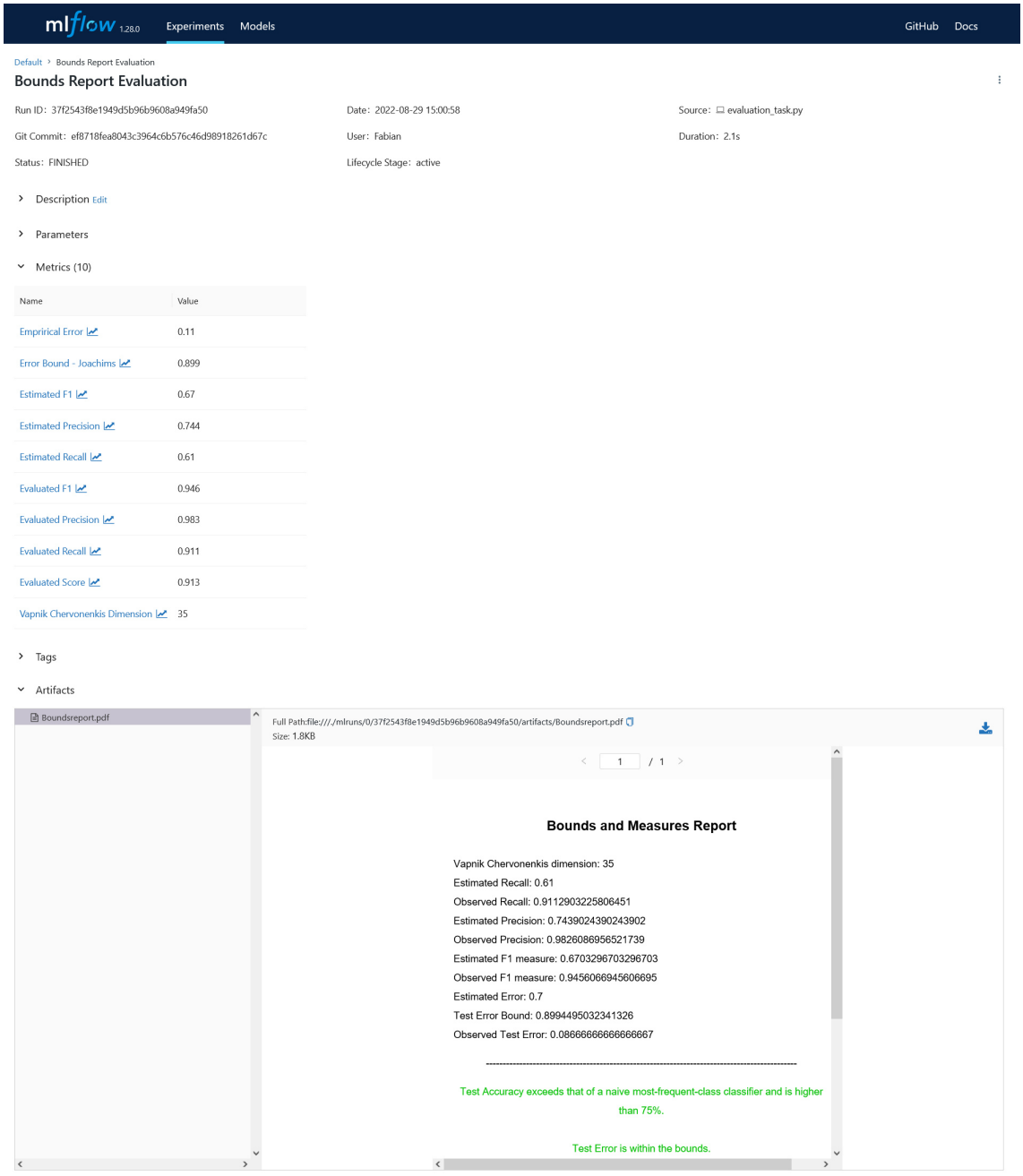

Figure A.6: Output metrics of the Bounds Report Evaluation of the linear SVC on Ionosphere

Bounds and Measures Report

Vapnik Chervonenkis dimension: 35

Estimated Recall: 0.61

Observed Recall: 0.9112903225806451

Estimated Precision: 0.7439024390243902

Observed Precision: 0.9826086956521739

Estimated F1 measure: 0.6703296703296703

Observed F1 measure: 0.9456066945606695

Estimated Error: 0.7

Test Error Bound: 0.8994495032341326

Observed Test Error: 0.08666666666666667

Test Accuracy exceeds that of a naive most-frequent-class classifier and is higher than 75%.

Test Error is within the bounds.

Probability for the bound holding is 95 percent.

References:

Burges, Christopher J. C.: A Tutorial on Support Vector Machines for Pattern Recognition. Data Mining and Knowledge Discovery, 2(2):121-167, jun 1998.

Joachims, Thorsten: Learning to Classify Text Using Support Vector Machines: Methods, Theory and Algorithms. Kluwer Academic Publishers, Norwell, Massachusetts, USA, 2002.

Figure A.7: Output report of the Bounds Report Evaluation of the linear SVC on Ionosphere

A.4 Decision Tree on Iris

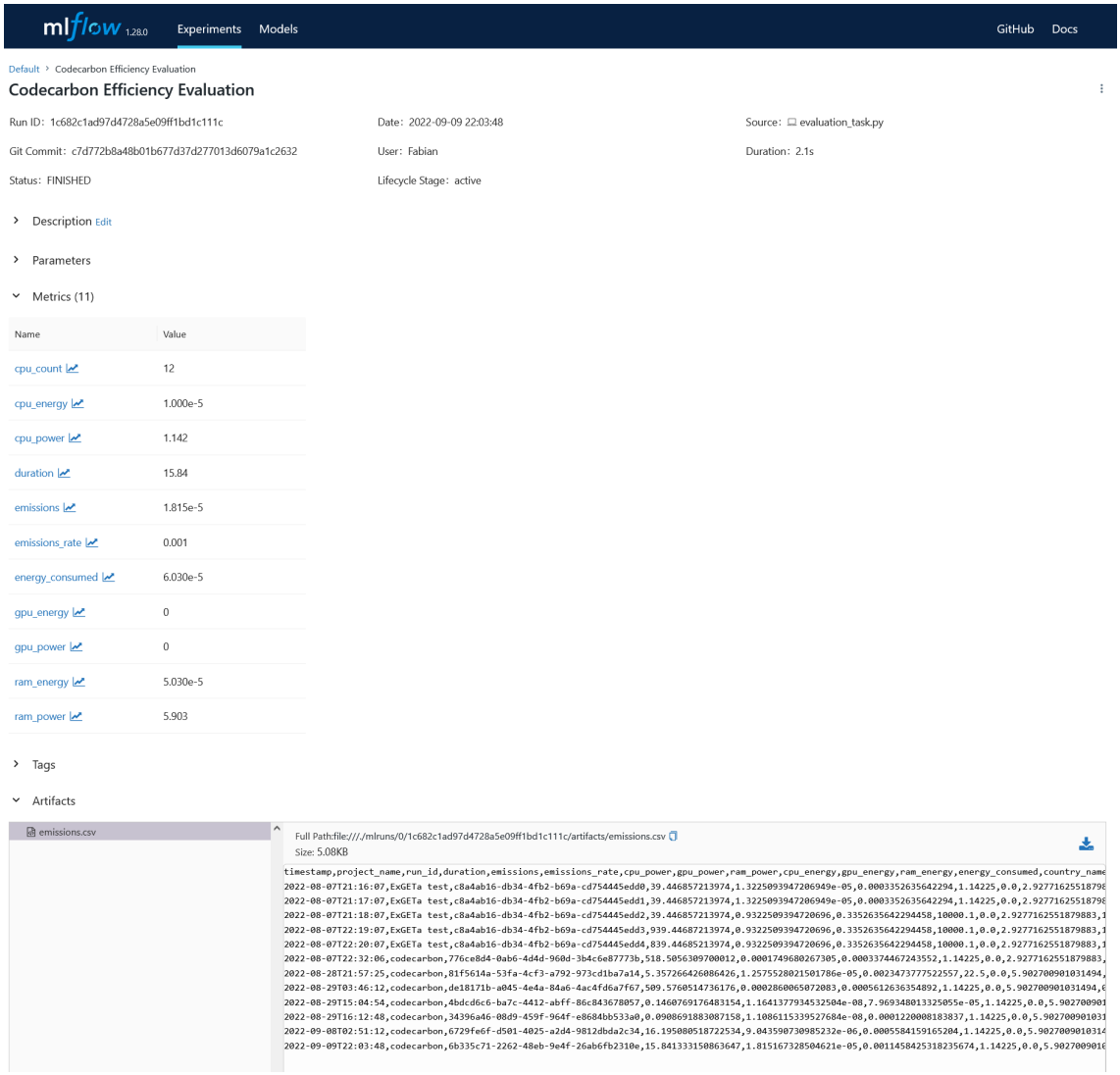

Figure A.8: Output of the Energy Consumption Evaluation of the Decision Tree on Iris

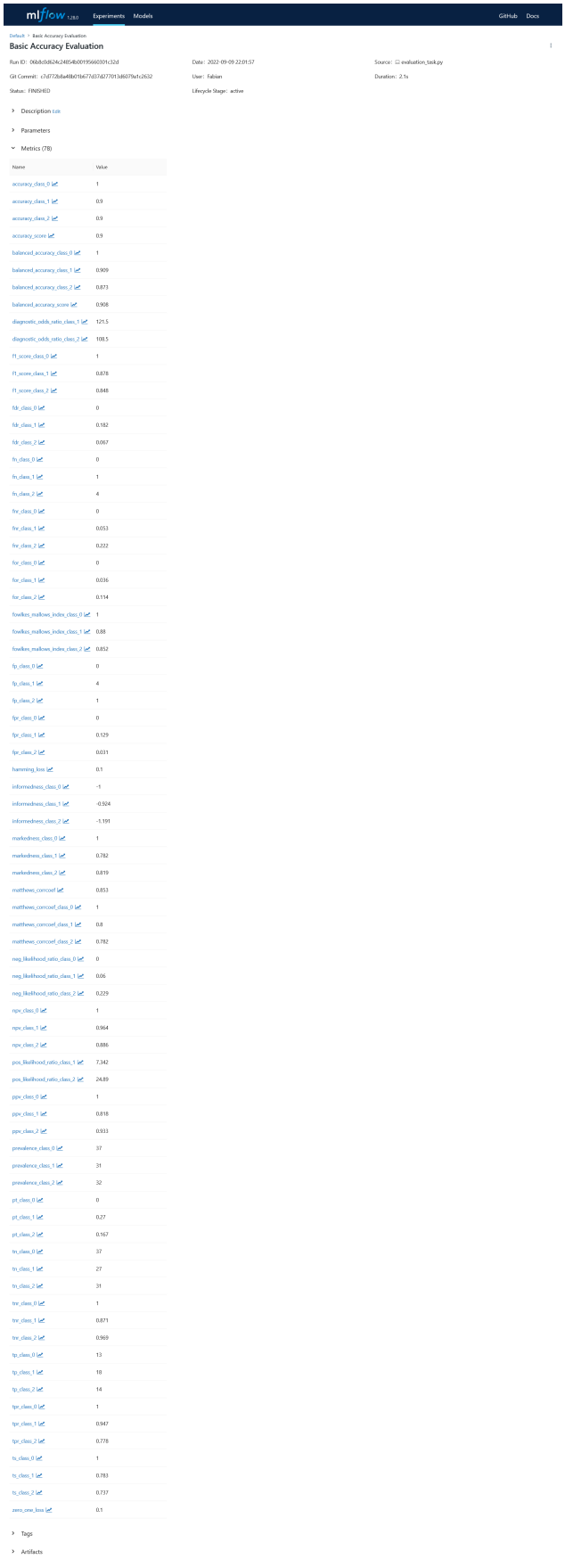

Figure A.9: Output of the Confusion Matrix Evaluation of the Decision Tree on Iris