# The Symbolic Machine Learning Prover

Franz Brauße    Zurab Khasidashvili    Konstantin Korovin

February 23, 2024

**Abstract**

Symbolic Machine Learning Prover (SMLP) is a collection of tools for reasoning about machine learning models. SMLP is based on SMT solver(s). In this document we describe functionality for computing safe and stable regions of neural network models satisfying optput specifications. This corresponds to solving $\epsilon$-guarded $\exists^*\forall^*$ formulas over NN representations [BKK20].

SMLP has been developed by Franz Brauße, Zurab Khasidashvili and Konstantin Korovin and is available under the terms of the Apache License v2.0.[1]

## Contents

[1] `https://www.apache.org/licenses/LICENSE-2.0`

# 1 Changes

- 07/07/2020: documentation of SMLP v0.1

## 2 Introduction

*Symbolic Machine Learning Prover* (SMLP) offers multiple capabilities for system's *design space exploration*. These capabilities include methods for selecting which parameters to use in modeling design for configuration optimization and verification; ensuring that the design is robust against environmental effects and manufacturing variations that are impossible to control, as well as ensuring robustness against malicious attacks from an adversary aiming at altering the intended configuration or mode of operation. Environmental affects like temperature fluctuation, electromagnetic interference, manufacturing variation, and product aging effects are especially more critical for correct and optimal operation of devices with analog components, which is currently the main focus area for applying SMLP.

To address these challenges, SMLP offers multiple modes of design space exploration; they will be discussed in detail in Section 11. The definition of these modes refers to the concept of *stability* of an assignment to system's parameters that satisfies all model constraints (which include the constraints defining the model itself and any constraint on model's interface). We will refer to such a value assignment as a *stable witness*, or *(stable) solution* satisfying the model constraints. Informally, stability of a solution means that any eligible assignment in the specified region around the solution also satisfies the required constraints. This notion is sometimes referred to as robustness. SMLP works with parameterized systems, where parameters (also called *knobs*) can be tuned to optimize the system's performance under all legitimate inputs.

Figure 1 depicts how SMLP views a system to analyze. Variables $x1, x2$ are the system's inputs, variables $p1, p2$ are the system's parameters, and variables $y1, y2$ are the system's outputs. The input, knob, and global constraints on the system's interface define the legal input space of the system as well as requirements that the system must meet after selecting the knob configuration. The model exploration task might consist of optimizing the system's knobs for a number of objectives, synthesizing the knob values to find a witness to a query (e.g., a desired condition), or verifying that a given configuration satisfies an assertion on the system's outputs.

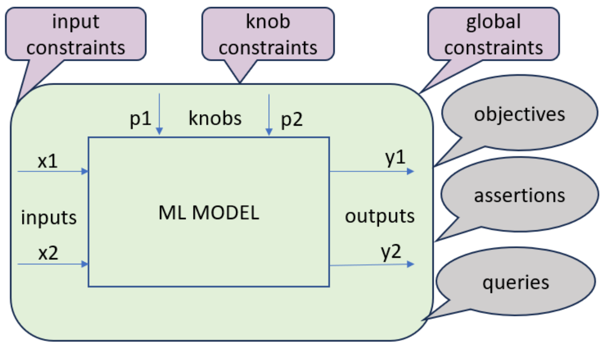

Figure 1: Parameterized system, with interface constraints

For example, in the circuit board design setting, topological layout of circuits, distances, wire thickness, properties of dielectric layers, etc. can be such parameters, and the exploration goal would be to optimize the system performance under the system's requirements [MSK21]. The difference between knobs and inputs is that knob values are selected during design phase, before the system goes into operation; on the other hand, inputs remain free and get values from the environment during the operation of the system. Knobs and inputs correspond to existentially quantified and universally quantified variables in the formal definition of model exploration tasks.

Thus in the usual meaning of verification, optimization and synthesis, respectively, all variables are inputs, all variables are knobs, and some of the variables are knobs and the rest are inputs.

Below by a *model* we refer to an ML model that models the system under exploration.

The *model exploration cube* in Figure 2 provides a high level and intuitive idea on how the model exploration modes supported in SMLP are related. The three dimensions in this cube represent synthesis (↘-axis), optimization (→-axis) and stability (↑-axis). On the bottom plane of the cube, the edges represent the synthesis and optimization problems in the following sense: synthesis with constraints configures the knob values in a way that guarantees that assertions are valid, but unlike optimization, does not guarantee optimally with respect to optimization objectives. On the other hand, optimization by itself is not aware of assertions on inputs of the system and only guarantees optimality with respect to knobs, and not the validity of assertions in the configured system. We refer to the process that combines synthesis with optimization and results in an optimal design that satisfies assertions as *optimized synthesis*. The upper plane of the cube represents introducing stability requirements into synthesis (and as a special case, into verification), optimization, and optimized synthesis. The formulas that make definition of stable verification, optimization, synthesis and optimized synthesis precise are discussed Section 11.

Could not used column sep = 8.5em, between origins option in the cube.

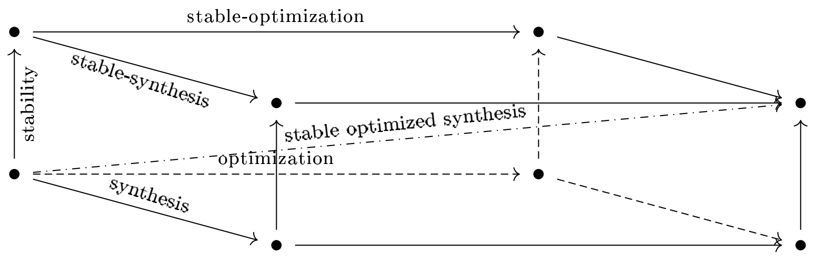

Figure 2: Exploration Cube

## 3 SMLP architecture

SMLP tool architecture is depicted in Figure 3. It consists of the following components: 1) Design of experiments (DOE), 2) System that can be sampled based on DOE, 3) ML model trained on the sampled data, 4) SMLP solver that handles different system exploration modes on a symbolic representation of the ML model, 5) Targeted model refinement loop.

SMLP supports multiple ways to generate training data known under the name of *Design Of Experiments (DOE)*. These methods include: full-factorial, fractional-factorial, Plackett-Burman, Box-Behnken, Box-Wilson, Sukharev-grid, Latin-hypercube, among other methods, which try to achieve a smart sampling of the entire input space with a relatively small number of data samples. In Figure 3, the leftmost box-shaped component called DOE represents SMLP capabilities to generate test vectors to feed into the system and generate training data; the latter two components are represented with boxes called SYSTEM and DATA, respectively.

The component called ML MODEL represents SMLP capabilities to train models; currently neural network, polynomial and tree-based regression models are supported. Modeling analog devices using polynomial models was proposed in the seminal work on *Response Surface Methodology (RSM)* [BW51], and since then has been widely adopted by the industry. Neural networks

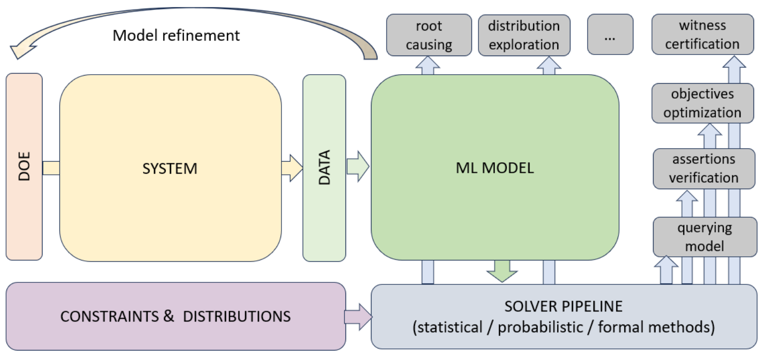

Figure 3: SMLP Tool Architecture

and tree-based models are used increasingly due to their wider adoption, and their exceptional accuracy and simplicity, respectively.

The component called SOLVER PIPELINE represents model exploration engines of SMLP (e.g., connection to SMT solvers), which besides a symbolic representation of the model takes as input several types of constraints and input sampling distributions specified on the model's interface; these are represented by the component called CONSTRAINTS & DISTRIBUTIONS located at the low-left corner of Figure 3, and will be discussed in more detail in Section 11. The remaining components represent the main model exploration capabilities of SMLP.

Last but not least, the arrow connecting the ML MODEL component back to the DOE component represents a *model refinement* loop which allows to reduce the gap between the model and system responses in the input regions where it matters for the task at hand (there is no need to achieve a perfect match between the model and the system everywhere in the input space). The targeted model refinement loop is discussed in Section 14.

## 4 How to run SMLP: a quick start

Command to run SMLP in *optimize* mode is given in Figure 4.

```
../src/run_smlp.py -data ../smlp_toy_basic -mode optimize -pareto t \
-spec ../smlp_toy_basic -resp y1,y2 -feat x1,x2,p1,p2 -model dt_sklearn \
-dt_sklearn_max_depth 15 -mrmr_pred 0 -epsilon 0.05 -delta 0.01 \
-save_model t -model_name toy_basic -save_model_config t \
-plots f -seed 10 -log_time f -out_dir ../out -pref try -pref try
```

Figure 4: Example of SMLP's command to build a decision tree model and perform an optimization task. Occurrences of ../ should be replaced with respective full or relative paths.

The option -data ../smlp_toy_basic defines the labeled datasat to use for model training and test. The dataset should be provided as smlp_toy_basic.csv file, and the two dots in ../smlp_toy_basic should be replaced with the actual full path to the file; the .csv suffix itself should be omitted. This dataset is displayed in Table 1, and it has six columns $x1, x2, p1, p2, y1, y2$.

The option -mode optimize defines the analysis mode to run, and option -pareto t instructs

|   | x1 | x2 | p1 | p2 | y1 | y2 |
|---|---|---|---|---|---|---|
| 0 | 2.9800 | -1 | 0.1 | 4 | 5.0233 | 8.0000 |
| 1 | 8.5530 | -1 | 3.9 | 3 | 0.6936 | 12.0200 |
| 2 | 0.5580 | 1 | 2.0 | 4 | 0.6882 | 8.1400 |
| 3 | 3.8670 | 0 | 1.1 | 3 | 0.2400 | 8.0000 |
| 4 | -0.8218 | 0 | 4.0 | 3 | 0.3240 | 8.0000 |
| 5 | 5.2520 | 0 | 4.0 | 5 | 6.0300 | 8.0000 |
| 6 | 0.2998 | 1 | 7.1 | 6 | 0.9100 | 10.1250 |
| 7 | 7.1750 | 1 | 7.0 | 7 | 0.9600 | 1.1200 |
| 8 | 9.5460 | 0 | 7.0 | 6 | 10.7007 | 9.5661 |
| 9 | -0.4540 | 1 | 10.0 | 7 | 8.7932 | 6.4015 |

Table 1: Toy dataset smlp_toy_basic.csv with two inputs $x_1, x_2$, two knobs $i_1, i_2$, and two outputs $y_1, y_2$.

SMLP that pareto optimization should be performed (as opposed to performing multiple single-objective optimizations when multiple objectives are specified).

The option -spec ../smlp_toy_basic defines the full path to the specification file that specifies the optimization problem to be solved. Figure 5 depicts the contents of this specification (spec) file. It defines legal ranges of variables $x1, x2, p1, p2, y1, y2$, where appropriate, which ones are inputs, which ones are knobs, which ones are the outputs, defines additional constraints on them, and defines the optimization objectives. Detailed description of the fields of the specification (which is loaded as a Python dictionary) is given in Section 7.

Options -resp y1,y2 -feat x1,x2,p1,p2 define the names of the responses and features to be used from the provided dataset. This information is available in the spec file as well, and therefore these options can be omitted in our example. In general, option values provided as part of the command line override values of these options specified in the spec file, and therefore command line options are convenient to quickly adapt an SMLP command without changing the spec file. Also, a spec file is needed mostly for the model exploration modes of SMLP, and command line options make invocation of SMLP in other modes simpler.

Option -model dt_sklearn instructs SMLP to train dt_sklearn model, which according to SMLP's naming convention for model training algorithms means to use the decision tree (dt) algorithm supported in sklearn package. And -dt_sklearn_max_depthl 15 instructs SMLP to use the sklearn's max_depth hyper-parameter value 15, based on a similar naming convention for hyper-parameters supported in model training packages used in SMLP.

Options -epsilon 0.05 -delta 0.01 define values for constants $\epsilon$ and $\delta$ required for approximating search for optima and guaranteeing that the search will terminate. These optimization algorithms and proofs that usage of constant $\delta > 0$ guarantees the termination can be found in [BKK20, BKK22]. Constant $\epsilon$ defines a termination criterion for search for optima, and is used to guarantee that the computed optima are not more than $\epsilon$ away (after scaling the objectives) from the real optima (of the function defined by the ML model). A formal description of usage of $\epsilon$ can be found in Section 11 as well as in [BKK20, BKK22]

Options -save_model t -save_model_config t instruct SMLP respectively to save the trained model and to save the option values used in current SMLP run into a SMLP invocation configuration file. Besides saving the model, SMLP saves all required information to enable rerun of the saved model on a new data. More details on saving a trained model and reusing it later on a new data is provided in Section 10.

Option -mrmr_predl 0 specifies that all features should be used for training a model, while option value greater than 0 defines how many features selected by the MRMR algorithm should

```
{
  "version": "1.2",
  "variables": [
    {"label":"y1", "interface":"output", "type":"real"},
    {"label":"y2", "interface":"output", "type":"real"},
    {"label":"x1", "interface":"input", "type":"real", "range":[0,10]},
    {"label":"x2", "interface":"input", "type":"int", "range":[-1,1]},
    {"label":"p1", "interface":"knob", "type":"real", "range":[0,10], "rad-rel":0.1, "grid":[2,4,7]},
    {"label":"p2", "interface":"knob", "type":"int", "range":[3,7], "rad-abs":0.2}
  ],
  "alpha": "p2<5 and x1==10 and x2<12",
  "beta": "y1>=4 and y2>=8",
  "eta": "p1==4 or (p1==8 and p2 > 3)",
  "assertions": {
    "assert1": "(y2**3+p2)/2>6",
    "assert2": "y1>=0",
    "assert3": "y2>0"
  },
  "objectives": {
    "objective1": "(y1+y2)/2",
    "objective2": "y1"
  }
}
```

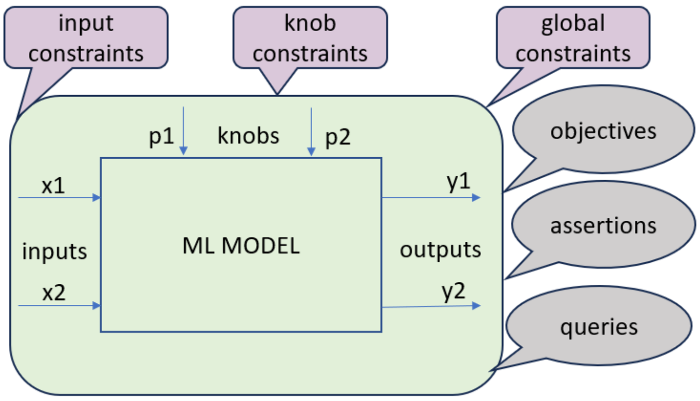

Figure 5: Specification smlp_toy_basic.spec used by SMLP command in Figure fig. 4.

be used for model training (see also Section 9.2.2).

After model training (or loading a pre-trained model) SMLP generates plots to visualize model predictions against the actual response values found in labeled data (training|test|new data). Option -plots f instructs SMLP to not open these plots interactively while SMLP is running; these plots are saved for offline inspection. See Section 10 for more information regarding prediction plots.

Option -seed 10 is required to ensure determinism in SMLP execution (running the same command should yield the same result). And option -log_time f instructs SMLP to not include time stamp in logged messages.

Option -out_dir ../out defines the output directory for all SMLP reports and collateral output files. See Section 8.2 for more information about SMLP output directory and reports.

Optimization progress is reported in file try_smlp_toy_basic_optimization_progress.csv, where try is the run ID specified using option -prefl try; smlp_toy_basic is the name of the data file, and optimization_progress.csv is the file name suffix for that report. This report contains details on input, knob, output and the objective's values demonstrating the proven upper and lower bounds of the objectives during search for a pareto optimum. It is available anytime after search for optimum has started and first approximations of the optima have been computed.

## 5 Concepts and Preliminaries

In this document, syntactical highlights are placed on `literal strings`, which are ASCII sequences used to communicate input to and output from programs; on `commands` to be executed; and on `API references` which correspond to either concrete symbols or abstract concepts defined in the corresponding API.

We assume familiarity with JSON, `make` and CSV files. Since the CSV format is not unambiguously defined, we give the concrete requirements in appendix A.1.

The glossary in this section defines concepts which are used throughout this document when describing in detail the problems solved by SMLP and is meant to be referred to on an per-instance basis instead of reading it as a whole.

**Center Threshold** A rational value in $[0, 1]$ larger or equal to Threshold.

**Codomain** A real vector space.

**Data set** A list of points in $D \times C$ where $D$ is the domain and $C$ is the codomain.

**Domain** A domain $D$ is the cartesian product $\bigtimes_{i=1}^{n} D_i$ where $D_i$ is a subset of either $\mathbb{Z}$, $\mathbb{R}$ or a discrete finite subset of $\mathbb{Q}$. In other places we use sets instead of finite subsets of Q?

**Feature** Any column of a data set $\mathcal{D}$ is called a feature.

**Instance** A tuple $(N, \mathcal{N}_D, o, \mathcal{N}_o)$ is called an instance if $N$ is an NN over domain $D$, $\mathcal{N}_D$ is a data normalization, $o$ is an objective function and $\mathcal{N}_O$ is an objective normalization. If – in addition to an instance $I$ – a data set $\mathcal{D}$ over $D$ is given, $(I, \mathcal{D})$ is called a data instance.

**NN** Neural network as understood by the Keras API of Tensorflow in HDF5 format. In particular, the Symbolic Machine Learning Prover only handles `Dense` layers with either `relu` or `linear` activation function. The input layer must be defined on domain $D$ and the output layer must map to the codomain $C$.

**Region** Given a domain $D = \bigtimes_{i=1}^{d} D_i$, a region (in $D$) is a product $\bigtimes_{i=1}^{d} C_i$ where each $C_i \subseteq D_i$ is a finite union of compact intervals for $i = 1, \dots, n$. For discrete $D_i$, e.g., a subset of $\mathbb{Z}$, $C_i$ is the finite union of point intervals. Otherwise, $D_i$ is a bounded subset of $\mathbb{R}$ and $C_i$ corresponds to just one interval $[c_i \pm r_i]$ with center $c_i \in D_i$ and rational radius $r_i > 0$.

**Safe** A region $R$ is considered safe (wrt. a given target function $f$) for a constant $t \in \mathbb{Q}$ if $f$ satisfies $f(x) \geq t$ for all $x \in R$.

**.spec** Specification file describing the domain, codomain and regions in the domain. Its format is a JSON list where the $i$-th entry corresponds to the $i$-th column in the CSV describing the data set.

Given a data set and a .spec file, the components $D_i$ of the domain are defined as $D_i = E_i(F_i)$ where $E_i$ is called the embedding of $F_i$ into $D_i$ where $F_i$ is the $i$-th input feature.

**Target function** The target function $f_I$ of an instance $I = (N, \mathcal{N}_D, o, \mathcal{N}_o)$ is defined as the composition $\mathcal{N}_o \circ o \circ \mathcal{N}_{D,\mathrm{o}} \circ N \circ \mathcal{N}_{D,\mathrm{i}}$. The target function $f_{I,\mathcal{D}}$ of a data instance $(I, \mathcal{D})$ is $x \mapsto \min(\{f_I(x)\} \cup \{\mathcal{N}_o(o(y)) : (x, y) \in \mathcal{D}\})$.

**Threshold** The threshold for a region $R$ and target function $f$ is defined as the maximal $t \in \{t_1, \dots, t_n\} \subset [0, 1] \cap \mathbb{Q}$ for which $R$ is safe wrt. $f$ for $t$ if it exists, and $-\infty$ otherwise.

The threshold for a domain $D$ and target function $f$ is defined as the maximal $t \in \{t_1, \dots, t_n\} \subset [0, 1] \cap \mathbb{Q}$ for which there is a region that is safe wrt. $f$ for $t$ if it exists, and $-\infty$ otherwise.

## 6 Prerequisites

This section describes software environments SMLP is known to work in. All packages corresponding to software listed in in Section 6.2 should be installed "system-wide", meaning that they are accessible to the corresponding interpreter or compiler listed in Section 6.1 without additional options.

Please note, that SMLP may run with versions of the packages not listed here. These lists are subject to change and are gradually extended as soon as more versions have been tested or other package requirements arise.

### 6.1 Compilers / Interpreters / Utils

These tools should be available in one of the paths mentioned in the `PATH` environment variable.

- `bash`: GNU Bourne Again Shell version 5.0_p17 `http://tiswww.case.edu/php/chet/bash/bashtop.html`
- `gmake`: GNU make version 4.1, 4.2 or 4.3 `https://www.gnu.org/software/make/make.html`
- `awk`: GNU awk version 5.0.1 or 5.1 `https://www.gnu.org/software/gawk/gawk.html`
- `sed`: GNU sed version 4.8 `http://sed.sourceforge.net/`
- GNU coreutils version 8.32 `https://www.gnu.org/software/coreutils/`
- GNU time version 1.7, 1.7.2 or 1.9 `https://www.gnu.org/directory/time.html`
- `cc`: either GNU C Compiler version 4.7.4, 5.4.0, 9.3.0 or 10.1.0 `https://gcc.gnu.org/` or Clang version 10.0.0 `https://clang.llvm.org/`
- `python3`: CPython version 3.6 or 3.7 `https://www.python.org/`

### 6.2 Libraries

The paths to the installed `python` directory should be set in the `PYTHONPATH` environment variable. See `python3 -h` for details on this variable.

- Tensorflow version 2.1 or 2.2 `https://www.tensorflow.org/`
- Z3 including its Python API version 4.8.6 or 4.8.8 `https://github.com/Z3Prover/z3`
- Pandas version 0.24.2 `https://pandas.pydata.org/`
- scikit-learn version 0.20.4 or 0.22.2_p1 `https://scikit-learn.org/`
- matplotlib version 2.2.4 or 3.1.2 `https://matplotlib.org/`
- seaborn version 0.9.x or 0.10.x `https://seaborn.pydata.org/`
- HDF5 for Python version 2.10.0 `https://www.h5py.org/`
- kjson version 0.1.3 (bundled in release) `https://github.com/fbrausse/kjson`

## 7 SMLP problem specification

The specification file defines the problem conditions in a JSON compatible format, whereas SMLP exploration modes can be specified via command line options. Figure 5 depicts a toy system with two inputs, two knobs, and two outputs and a matching specification file for model exploration modes in SMLP. This system and the spec file were used in Section 4 to give a quick introduction on how to run SMLP in the optimize mode.

"version" specifies the version of the spec file format. Versions are defined for backward compatibility.

"variables" defines properties of the system's interface variables. For each variable it specifies its

- "label" the name, e.g., $x1$
- "interface" function, which can be input, knob, or output
- "type" which can be real, int, or set (for categorical features)
- "range" for variables of real and int types, e.g., $[2, 4]$ (must be a closed interval). Implimentation allows plus/minus infinity – discorage its usage because of performance considerations? The ranges serve as assumptions in model exploration modes of SMLP.
- "grid" for knobs, which is a list of values that a knob variable is allowed to take on within the respective declared ranges, independently from other knobs. The $\eta$ constraints introduced below further restrict the multi-dimensional grid. Both real and int typed knobs can be restricted to grids (but do not need to). Grids serve as assumptions in model exploration modes of SMLP.

"alpha" defines extra constraints on inputs and knobs (on top of constraints inferred from input and knob ranges and knob grids). These constraints serve as assumptions in model exploration modes of SMLP.

"eta" defines extra constraints on knobs (on top of constraints inferred from knob ranges and grids).

"beta" defines global constraints on inputs, knobs and outputs that serve as requirements that need to be met by selected knob configurations.

"assertions" defines assertions: a dictionary that maps assertion names to respective expressions.

"queries" defines queries: a dictionary that maps query names to respective expressions.

"objectives" defines optimization objectives: a dictionary that maps objective names to respective expressions.

The expressions that occur in a spec file, such as "alpha", "beta", "eta" constraints, as well as "assertions", "queries", and "objectives", can in principle be any Python expression that can be composed using the operator package[2]. In SMLP these expressions are parsed using the Abstract Syntax Trees library[3]. Currently only a subset of operations from the operator package is supported in these expressions (there has not been a need for others so far):

binaryop add(a, b) [a + b], sub(a, b) [a-b], mul(a, b) [a*b], truediv(a, b) [a/b], pow(a, b) [a**b]

unaryop neg(a) [-a]

[2]https://docs.python.org/3/library/operator.html
[3]https://docs.python.org/3/library/ast.html

bitwiseop and_(a, b) [a&b], or_(a, b) [a|b], inv(a) [$\sim$ a], xor(a, b) [a^b]

cmpop eq(a, b) [a == b], ne(a, b) [a! = b], lt(a, b) [a < b], le(a, b] [a $\leq$ b], gt(a, b) [a > b], ge(a, b) [a $\geq$ b]

The "alpha", "beta", "eta" constraints can also be defined in SMLP command line using options -alpha expression, -beta expression, -eta expression. Assertions can be specified as part of command line, using options -asrt_names and -asrt_exprs. For example, assertions from the spec in Figure 5 can be specified as follows: -asrt_names assert1, assert2, assert3 specifies assertion names as a comma-separated list of names, and -asrt_expr "(y2**3 + p2)/2 > 6; y1 >= 0; y2 > 0" defines the respective expressions as a semicolon separated list of expressions. Similarly, queries can be specified in command line using options -quer_names and -quer_exprs; and optimization objectives can be specified using options -objv_names and -objv_exprs.

Precise handling of constraints "alpha", "beta", "eta", as well as handling of "assertions", "queries", and "objectives", depends on the model exploration modes of SMLP and is described in dedicated subsections of Section 11.

## 8 SMLP input and output

Input files to an SMLP command can be located in different directories and have in general different formats. The most common input files and how to feed them to SMLP is described in Subsection 8.1. All outputs from an SMLP run, on the other hand, are written into the same output directory, as described in Subsection 8.2.

### 8.1 SMLP inputs

training data should be a .csv file. Full path to data should be specified using option -data, without the .csv suffix. For modes where a model is trained, data should include one or more responses. All responses must be numeric (categorical features as responses will be supported in future). The data file is relevant for all modes of SMLP except for the doe mode.

new data should be a .csv file. Full path to data should be specified using option -new_data without the .csv suffix. This data usually not available during model training, and usually is also not labeled: it may not contain the response columns and should contain all features from the training data that were actually used in model training. New data is used to perform prediction with model trained of training data; this model can be generated in the same SMLP run or could have been trained and saved earlier. New data is mainly relevant for mode predict, but new data ca be supplied in model exploration modes as well and in this case predictions on new data will be performed as part of model exploration analysis. When new data has the responses, they must be of the same type as in the training data, and after predictions the model accuracy will be reported for both training and new data.

problem spec should be a .spec file, with the content in json format (so it is loaded using json.load() as a Python dictionary). Full path to spec file should be specified using option -spec, without the suffix .spec. It is required in model exploration modes (query, verify, optimize, etc.).

doe spec should be a .csv file. Full path to DOE (design of Experiments) spec file should be specified using option -doe_spec. Required for the doe mode only, for DOE generation.

## 8.2 SMLP outputs

SMLP communicates its results using files, and it outputs all reports, plots, and collateral files in the same directory. A full path to that output directory can be specified using option -out_dir, and it is recommended to specify it. If not specified, the directory of input data file is used as the output directory. If the latter is not specified (say if a saved model is used for performing prediction), the directory of the new data file is used as the output directory. If the new data file is not specified either, say in case of doe mode, then the directory of the DOE spec file is used as the output directory. Otherwise an error is issued.

The output files may also include a saved trained model and a collection of other files that together have all the information required to rerun the saved model on new data. All files collectively defining a saved model start with the same name prefix. This prefix is a concatenation of the SMLP invocation ID/name specified using option -pref runname, and the saved model name specified using option -model_name modelname. If saved model name is not specified, the prefix for all model related file names is computed by SMLP using the data name that was used for training the model, but this might change in future and it is recommended to always use a model name when saving a trained model.

All the other output file names also have the same prefix, computed by concatenating the SMLP run ID/name specified using option -pref and the name of input data (or new data) file in modes where these data files are provided, or with the name of the saved mode if the latter is used in analysis, or the DOE spec file name in the doe mode. Currently any SMLP mode uses at least one of the following: input (training) data, new data, or DOE spec file, therefore file name prefixes are well defined both for saved model related files as well as SMLP report and collateral files. Assuming a unique ID/name is used for each SMLP run (specified using option -pref), all files generated as a result of that run can be identified uniquely.

# 9 Data processing options

In SMLP we distinguish between two stages of input data processing: a *data preprocessing stage*, followed by a *data preparation stage* for the required type of analysis.

## 9.1 Data preprocessing options

Data *preprocessing* is applied to raw data immediately after loading, and its aim is to process data in order to confirm to SMLP data requirements. That is, this stage of data processing is to make SMLP tool user friendly, and perform some data transformations instead of the user having to do this. Thus, all the reports and visualization of the results will use preprocessed data, and assume the data was passed to SMLP in that form. As an example, if some values in columns were replaced in the preprocessing stage, say 'pass' was replaced by 0 and 'fail' was replaced by 1, the reports will use values 0 and 1 in that column.

Next we explain the main steps performed as part of preprocessing of training data.

### 9.1.1 Selecting features for analysis

If SMLP command includes option -feat x, y, z, then only features $x, y, z$ will be used in analysis (besides the responses); the rest of the features will be dropped.

### 9.1.2 Missing values in responses

Response columns, say $y1, y2$, in training data are defined using option -resp y1, y2. Rows in the training data where at least one response has a missing value will be dropped.

### 9.1.3 Constant features

Constant features (that have exactly on non-NaN value) are dropped.

### 9.1.4 Missing values in features

Missing value imputation is performed with the most_frequent strategy of SimpleImputer class from sklearn package. The locations of missing values prior to imputation is computed as a dictionary and saved as a json file with suffix _missing_values_dict.json for future reference (say to mark respective locations or samples on plots).

### 9.1.5 Boolean typed features

Currently SMLP does not have a need to make a direct usage of boolean type in features (or in responses). Therefore Boolean typed features are treated as categorical features with type object, by converting the Boolean values to strings 'True' and 'False'.

### 9.1.6 Determining types of responses

*Categorical responses:* Categorical responses are supported only if they have two values – it is user responsibility to encode a categorical response with more than two levels (values) into a number of binary responses (say through the one-hot encoding). A categorical response can be specified as a (a) 0/1 feature, (b) categorical feature with two levels; or (c) numeric feature with two values. In all cases, parameters specified through options positive_value and negative_value determine which one of these two values in that response define the positive samples and which ones define the negative ones – both in training data and in new data if the latter has that response column. Then, as part of data preprocessing, the positive_value and the negative_value in the response will be replaced by 1 and 0, respectively, following the convention in statistics that integer 1 denotes positive and 0 denotes negative.
*Numeric response columns:* Float and int columns in input data can define numeric responses. Each such response with more than two values is treated as numeric (and we are dealing with a regression analyses). If a response has two values, than it can still be treated as a categorical/binary response, as described in case (c) of specifying binary responses. Otherwise – that is, when {positive_value, negative_value} is not equal to the set of the two values in the response, the response is treated as numeric.
*Multiple responses*: Multiple responses can be treated in a single SMLP run only if all of them are identified as defining regression analysis or all of them are identified as defining classification analysis. if that is not the case, SMLP will abort with an error message clarifying the reason.

## 9.2 Data preparation for analysis

We now describe data preparation steps supported in SMLP.

### 9.2.1 Processing categorical features

After preprocessing the only supported (and expected) data column types are int, float and categorical, where categorical features can have types object (with values of type string), or category; the category type can be ordered or unordered.

Some of the ML algorithms prefer to use categorical features as is – with string values: for example, feature selection algorithms can use dedicated correlation measures for categorical features. Also, some of the model training algorithms, such as tree based, can deal with categorical features directly, while others, e.g., neural networks and polynomial models, assume all inputs are numeric (int or float). Therefore, depending on the analysis mode (feature selection, model training, model exploration), categorical features might be encoded into integers (and be treated as discrete domains), simply by enumerating the levels (the values) seen in categorical features and replacing occurrences of each level with the corresponding integer. Currently encoding categorical features as integers is the default in model training and exploration modes in SMLP.

Conversely, some ML algorithms (especially, correlations) might prefer to *discretize* numeric features into categorical features, and discretization options in SMLP support discretization of numeric features with target types object and category, ordered or unordered, where the values in the resulting columns can represent integers (as strings, e.g., '5', or as levels, e.g., 5), or other string values (like 'bin5'). Discretization is controlled using the following options:

- discr_algo: discretization algorithm can be uniform, quantile, kmeans, jenks, ordinals, ranks.
- discr_bins: specifies number of required bins.
- discr_labels: if true, string labels (e.g., 'Bin2') will be used to denote levels of the categorical feature resulting from discretization; otherwise integers (e.g., 2) will be used to represent the levels.
- discr_type: the resulting type of the obtained categorical feature; can be specified as object, category, ordered, unordered, and integer.

### 9.2.2 Feature selection for model training

SMLP incorporates the MRMR feature selection algorithm [DP05] for selecting a subset of features that will be used for model training, using Python package mrmr.[4] SMLP option -mrmr_pred 15 instructs the MRMR algorithm to select 15 features, according to the principle of *maximum relevance and minimum redundancy.*

### 9.2.3 Data scaling / normalization

Data scaling is managed separately for the features and the responses. A particular mode of usage (model training and prediction, feature selection or range analysis, pareto optimization, etc.) can decide to scale features and or scale responses. The reports and visualization should use features and responses in the original scale, thus unscaling must be performed.

Features and responses might or might not be scaled, and these are controlled using two options: the option -data_scaler controls which data scaler should be used: the MinMaxScaler class of sklearn package or none (in which case neither features nor responses can be scaled); and Boolean typed options -scale_feat and -scale_resp for controlling feature and response scaling, respectively.

[4]https://github.com/nlhepler/mrmr.

SMLP optimization algorithms operate with data in original scale, while the optimization objectives are scaled (always, in current implementation) to $[0, 1]$ based on the min and max values of each individual objective in the training data. If training data is not provided (e,g., if a saved model is used for model exploration) then the scaling factor is determined based on sampling the objective's values on the model. not implemented

## 9.3 Processing of new data

In model training and exploration modes, most of the above described data processing steps are applied to training data. New data for performing predictions, if supplied, requires related feature processing and sanity checks to ensure that it does not contain any features not used in model training, and categorical features in new data do not have levels that were not present in the same features in training data. Some processing steps, such as missing value imputation in features, are applied to both training and new data.

## 9.4 Output files during data processing

The following information is computed and saved in output files during data processing stages. This information is required for performing predictions based on a saved model as well as in model exploration modes.

- *_data_bounds.json: Dictionary, with feature and response names in training data as the dictionary keys; and the min/max info of these features and responses as the dictionary values.
- *_missing_values_dict.json: Dictionary, with names of features that have at least one missing value as the dictionary keys; and the list of indices of missing values in these features as the dictionary values.
- *_model_levels_dict.json: Dictionary, with names of categorical features in input data as the dictionary keys, and the levels (the values) in these features as the dictionary values.
- *_model_features_dict.json: Dictionary, with names of responses as the dictionary keys; and the names of features used to train model for that response as the dictionary values.
- *_features_scaler.pkl: Object of MinMaxScaler clas from sklearn package, used for scaling features, saved as .pkl file.
- *_responses_scaler.pkl: Object of MinMaxScaler clas from sklearn package, used for features scaling the responses, saved as .pkl file.

# 10 Training ML models with SMLP

SMLP supports training tree-based and polynomial models using the scikit-learn[5] and pycaret[6] packages, and training neural networks using the Keras package with TensorFlow[7]. For systems with multiple outputs (responses), SMLP supports training one model with multiple responses as well as training separate models per response (this is controlled by command-line option -model_per_response). Supporting these two options allows a trade-off between the accuracy of

[5]https://scikit-learn.org/stable/
[6]https://pycaret.org
[7]https://keras.io

```
../src/run_smlp.py -data ../smlp_toy_basic -out_dir ../out -pref test_predict \
-mode predict -resp y1,y2 -feat x1,x2,p1,p2 -model poly_sklearn -save_model t \
-model_name test_predict_model -save_model_config t -mrmr_pred 0 -plots f \
-seed 10 -log_time f -new_data ../smlp_toy_basic_pred_unlabeled
```

Figure 6: Example of SMLP's command to train a polynomial model and perform prediction on new data.

the models (models trained per response are likely to be more accurate) and with the size of the formulas that represent the model for symbolic analysis (one multi-response model formula will be smaller at least when the same training hyper-parameters are used). Conversion of models to formulas into SMLP language is done internally in SMLP (no encoding options are exposed to user in current implementation, which will change once alternative encodings will be developed).

Figure 6 displays an example SMLP command in prediction mode. The option -data specifies full path to data csv file (the .csv suffix is not required). Similarly, option -new_data specifies full path to the new data (usually not available/used during model training and validation). Option -resp defines the names of the responses; and option -feat defines the names of a subset of features from training data to be used in ML model training (the same subset of features is selected form new data to perform prediction on new data). Option -model defines the ML model training algorithm. As an example, the command in Figure 6 trains a polynomial model using the scikit-learn package, and in SMLP model training algorithm naming convention is to suffix the algorithm name with the package name, separated by underscore, to form the full algorithm name. For Keras and pycaret packages, the package name suffixes used are keras and pycaret, respectively, while for algorithms from scikit-learn package we use an abbreviated suffix sklearn.

SMLP supports saving a trained model to reuse it in future on incoming new data. Saving a model is enabled using options -save_model t -model_name modelname, and rerunning a saved model is enabled using options -use_model t -model_name ../modelname. In addition, using options -save_model t -model_name modelname -save_config t one can generate a configuration file that records model training options prior to saving the model and enables one to easily rerun the saved model. The nn_keras models are saved and loaded using the .h5 format, while for models trained using sklearn and pycaret packages the .pkl format is used. Figure 7 gives two example commands to rerun a saved model on new data. The first one repeats the options of the command that trained and saved the model, and the second one uses the configuration file saved during the model training (the configuration file records all the SMLP options used during model training; therefore there is no need to repeat the SMLP options used during model training when reusing the saved model with the configuration file).

Here is the list of reports and collateral files generated in SMLP modes that require ML model training or rerunning of a saved ML model.

- *_{training|test|labeled|new}_predictions_summary.csv: prediction results respectively on training data samples, on test data (also called validation data) samples, on the entire labeled data samples (which includes both training and test data samples), and on new data samples. It is saved as a .csv file that includes the values of the responses as well.
- *_{training|test|labeled|new}_prediction_precisions.csv: prediction previsions per response, respectively on training data samples, on test data (also called validation data) samples, on the entire labeled data samples (which includes both training and test data samples), and on new data samples. It is saved as a .csv file that includes the values of the responses

```
../src/run_smlp.py -model_name ../test_predict_model -out_dir ../out \
-pref test_prediction_rerun -new_data ../smlp_toy_basic_pred_unlabeled \
-config ../test_predict_model_rerun_model_config.json

../src/run_smlp.py -mode predict -resp y1,y2 -feat x1,x2,p1,p2 -out_dir ../out \
-use_model t -model_name ../test_predict_model -model poly_sklearn \
-save_model f -pref model_rerun -mrmr_pred 0 -plots f \
-seed 10 -log_time f -new_data ../smlp_toy_basic_pred_unlabeled
```

Figure 7: Examples of SMLP's commands to use a saved mode to perform prediction on new data.

as well. For regression models currently supported in SMLP, two measures of precision are reported: msqe, and r2_score.

- *_{training|test|labeled|new}_{dt_sklearn|poly_sklearn|nn_keras|...}.png: prediction plots, respectively for training, test, labeled and new data, for the model trained respectively using dt_sklearn, poly_sklearn, nn_keras, or other regression model training algorithms supported in SMLP. These plots can be displayed during SMLP execution, for an interactive review, using option -plots t; otherwise these plots are saved for an offline review.
- *_{dt_sklearn|poly_sklearn|nn_keras|...}_model_complete.{h5|pk}: saved mode in .h5 format for nn_keras and in .pkl format for ML models trained using packages sklearn and caret.
- *_rerun_model_config.json: SMLP venation options configuration file created when saving a trained model, and loaded when re-using the saved model. This configuration file records all option values in the SMLP run that trains the model, and it can be used to rerun the model on a new data using option -config full_path_to_config_file, as described earlier in this section. During rerun, usually options -pref, -new_data and -model_name, with full paths to new data and saved model files (with the model name as prefix), respectively, are specified along with the configuration file, and these option values override the respective option values recorded within the configuration file. Besides the saved model itself, rerunning a saved model requires several other files saved during data processing steps (preprocessing and data preparation for analysis), which are described in Section 9.4.

## 11 ML model exploration with SMLP

SMLP supports the following model exploration modes (we assume that an ML model $M$ has already been trained).

certify Given an ML model $M$, a value assignment $a$ to knobs, and a query query, check that $a$ is a stable witness for query on model $M$. Multiple pairs $(a, \text{query})$ of candidate witness $a$ and query query can be checked in a single SMLP run.

query Given an ML model $M$ and a query query, find a stable witness $a$ for query on $M$.

verify Given an ML model $M$, a value assignment $a$ to knobs, and an assertion assert, verify whether assert is valid on model $M$ for any legal inputs (for the given value assignment $a$ to knobs). SMLP supports verifying multiple assertions in a single run.

synthesize Given an ML model $M$, find a stable configuration of knobs (a value assignment $a$ to knobs) such that required constraints (possibly including assertions), are valid for all legal value assignments to the inputs.

optimize Given an ML model $M$, find a stable configuration of knobs that yields a pareto-optimal values of the optimization objectives (pareto-optimal with respect to the *max-min* optimization problem defined in [BKK24]).

optsyn Given an ML model $M$, find a stable configuration of knobs that yields a pareto-optimal values of the optimization objectives and all constraints and assertions are valid for all legal values of inputs. This mode is a union of the *optimize* and *synthesize* modes, its full name is *optimized synthesis*. All the previous modes can be seen as a special case of optimized synthesis mode.

A formal definition of the tasks accomplished with these model exploration modes can be found in [BKK24]. Formal descriptions combined with informal clarifications will be provided in this section as well. Running SMLP in these modes reduces to solving formulas of the following structure:

$$\exists p \left[\eta(p) \wedge \forall p' \; \forall xy \; [\theta(p,p') \implies (\varphi_M(p',x,y) \implies \varphi_{cond}(p',x,y))]\right] \tag{1}$$

where $p$ denotes model knobs, $x$ denotes inputs, $y$ denotes the outputs, $\eta(p)$ defines constraints on the knobs, $\varphi_M(p',x,y)$ defines the ML model constraints, and the condition $\varphi_{cond}(p',x,y)$ depends on the SMLP mode and will discussed in subsections below. $\eta(p)$ need not be a conjunction of constraints on individual knobs inferred from their range and grid specification; it can define more complex relations between allowed knob values of individual knobs. $\theta(p,p')$ is in general a reflexive predicate, and in SMLP it is used as $\theta_r(p,p') = \|p-p'\| \le r$, where $\|p-p'\|$ is a distance between two configurations $p$ and $p'$, and $r$ is a relative or absolute *radius*; that is, the $\theta_r(p,p')$ region corresponds to a ball (or box) around $p$. $\varphi_M(p,x,y)$ is computed by SMLP internally, based on the ML model specification. $\varphi_{cond}(p',x,y)$ can represent (1) verification conditions, (2) model querying conditions, (3) parameter optimization conditions, or (4) parameter synthesis conditions.

**Definition 1** • *Given a value assignment $p^*$ to knobs $p$, an assignment $x^*$ to inputs $x$ is called a* (concrete) $\theta$-stable witness *to* $\mathsf{query}(p,x,y)$ *for configuration $p^*$ if the following formula is valid:Assume here that $x$ is not an empty vector?*

$$\varphi^{px}_{witn}(p^*,x^*) = \eta(p^*) \wedge \left(\forall p' \; \forall y \; [\theta(p^*,p') \implies (\varphi_M(p',x^*,y) \implies \varphi_{cond}(p',x^*,y))]\right) \tag{2}$$

*where*

$$\varphi_{cond}(p,x,y) = \alpha(p,x) \implies \mathsf{query}(p,x,y).$$

*In this case the value assignment $p^*, x^*$ to inputs and knobs $p, x$ is called a* (concrete) $\theta$-stable witness for $\mathsf{query}(p,x,y)$*, and $x^*$ is called a* (concrete) $\theta$-stable witness to $\mathsf{query}(p^*,x,y)$.

• *A value assignment $p^*$ to knobs $p$ is called a* (universal) $\theta$-stable witness for $\mathsf{query}(p,x,y)$ *if any legal value assignment $x^*$ to inputs $x$ is a (concrete) $\theta$-stable witness to $\mathsf{query}(p,x,y)$ for $p^*$; that is, when the following formula is valid:*

$$\varphi^{p}_{witn}(p^*) = \eta(p^*) \wedge \left(\forall p' \; \forall xy \; [\theta(p^*,p') \implies (\varphi_M(p',x,y) \implies \varphi_{cond}(p',x,y))]\right) \tag{3}$$

- *An assertion* $\mathsf{assert}(p, x, y)$ *is called* $\theta$-valid with respect to knob configuration $p^*$ *if its negation* $\mathsf{query}(p, x, y) = \neg\,\mathsf{assert}(p, x, y)$ *does not have a $\theta$-stable witness $x^*$ for $p^*$. In the latter case assertion* $\mathsf{assert}(p^*, x, y)$ *is called* $\theta$-valid. *Otherwise, if $x^*$ is a $\theta$-stable witness for* $\mathsf{query}(p^*, x, y)$*, then we call $x^*$ a* $\theta$-stable counter-example to $\mathsf{assert}(p^*, x, y)$*, and* $\mathsf{assert}(p^*, x, y)$ *is* $\theta$-falsifiable *(that is, it is not $\theta$-valid).*

The idea behind the concept of $\theta$-validity of an assertion $\mathsf{assert}(p, x, y)$ with respect to a knob configuration $p^*$ is as follows. Under the assumption that the system is in one of the points in the $\theta$-stability region of $p^*$, while the probability for the system of being in one of these points is 1 (due to the assumption), the probability for the system of being in a particular point $p^*$ in the $\theta$-stability region is 0. [8] Now, consider an input assignment $x^*$ such that $\mathsf{query}(p^*, x^*, y) = \neg\,\mathsf{assert}(p^*, x^*, y)$ holds and $x^*$ is not a $\theta$-stable witness to $\mathsf{query}(p^*, x, y)$. Then $x^*$ is a counter-example to $\mathsf{assert}(p^*, x, y)$, but $x^*$ can occur with probability 0; as a consequence, $\mathsf{assert}(p^*, x, y)$ does not have a counter-example with probability greater than 0 and is therefore defined as $\theta$-valid in the above definition.

## 11.1 Certification of candidate stable witness

Certification is defined both for a candidate stable configuration $p^*$ as well as for an assignment $p^*, x^*$ to both inputs and knobs. Still for now perturbation is defined for knobs only, not for inputs. Extension of perturbation to inputs requires more discussion (is this really needed?) Let's first consider the former case. We are given a candidate configuration of knobs $p^*$ (which is a value assignment to knobs $p$), and a query $\mathsf{query}$ on an ML model $M$, and we want to check whether $p^*$ is a stable witness for $\mathsf{query}$. Intuitively, the latter means that $p^*$ satisfies $\mathsf{query}$ on model $M$ for any legal values of inputs $x$, even if the knob values in configuration $p^*$ are perturbed within a predefined radius. Formally, SMLP checks the validity of formula $\varphi_{cert}(p^*)$:

$$\varphi_{cert}(p^*) = \eta(p^*) \wedge \ \big(\forall p' \ \forall xy \ [\theta(p^*, p') \implies (\varphi_M(p', x, y) \implies \varphi_{cond}(p', x, y))]\big) \tag{4}$$

where

$$\varphi_{cond}(p, x, y) = \alpha(p, x) \implies \mathsf{query}(p, x, y).$$

and $\alpha(p, x)$ defines user-given constraints on knobs and inputs.

For the above formula to be valid and not be valid vacuously, we need to check that $\eta(p^*)$ evaluates to constant true and $\alpha(p^*, x)$ is satisfiable. This means that $p^*$ witnesses that $\alpha(p, x) \wedge \eta(p)$ is *consistent*, that is, there exist values of inputs that satisfy $\alpha(p^*, x) \wedge \eta(p^*)$. Formally, consistency check for stable witness certification requires the following formula to be valid:

$$\exists x \ [\alpha(p^*, x) \wedge \eta(p^*)]) \tag{5}$$

Syntactic checks in SMLP imposed on the specification file ensure that all knobs are assigned fixed values in $p^*$. If the consistency check fails, SMLP reports the status of $p^*$ as *not a witness*; otherwise SMLP checks satisfiability of the following formula, which we refer to as *feasibility* part of stable witness certification:

$$\varphi_M(p^*, x, y) \wedge \alpha(p^*, x) \wedge \mathsf{query}(p^*, x, y)) \tag{6}$$

[8] Here we have the situation that an infinite sun of 0s is 1. The number 1 is selected because we are talking about probability and 1 is the maximum probability. Here we want to make connection between stability and probability, and we need an arithmetic / a calculus where an infinite sum of 0s is not 0. More precisely, likely we want $0 \times \aleph_0 = 0$; $0 \times \aleph_1 = 1$; $\ldots$; $0 \times \aleph_i = \aleph_{i-1}$ (0 times the cardinality of rationals; 0 times the cardinality of reals, etc.). Need to figure out what this calculus is – maybe something related to the *Uncertainty Principle* in quantum mechanics.

```
{
        "query1": "not a witness",
        "query2": "stable witness"
}
```

Figure 8: SMLP result in mode "certify".

If the above formula is not satisfiable, then $p^*$ cannot be a stable witness to $\mathsf{query}$, and SMLP reports its status as *not a witness.* Otherwise stability of the candidate witness $p^*$ for $\mathsf{query}$ is checked by proving validity of formula eq. (4), and this is done by checking satisfiability of formula eq. (7), which is the negation of the right conjunct of eq. (4):

$$\theta(p^*, p') \wedge \varphi_M(p^*, x, y) \wedge \alpha(p^*, x) \wedge \neg\, \mathsf{query}(p^*, x, y)) \tag{7}$$

SMLP supports checking multiple witness-query pairs in a single run. The status of each witness is reported in file *prefix_dataname_certify_results.json*, and each witness has one of these three status options:

- *not a witness* –formula eq. (5) is not valid or formula eq. (6) is not satisfiable.
- *witness, not stable* – formula eq. (4) is not valid (formula eq. (5) might still be valid and formula eq. (6) satisfiable).
- *stable witness* – formula eq. (4) is valid (formula eq. (7) is not satisfiable).

A results file might look like one on figure fig. 8:

Now let's consider that we are given an assignment $p^*, x^*$, a query $\mathsf{query}$ on an ML model $M$, and we want to check whether $x^*$ is a stable witness to $\mathsf{query}$ for $p^*$. This requires checking validity of formula eq. (2). The consistency check for this task is simply checking validity of $\alpha(p^*, x^*) \wedge \eta(p^*)$, and the feasibility part for this task is checking satisfiability of

$$\varphi_M(p^*, x,^* y) \wedge \alpha(p^*, x^*) \wedge \mathsf{query}(p^*, x^*, y)) \tag{8}$$

If the above formula is not satisfiable, then $x^*$ cannot be a stable witness to $\mathsf{query}$ for $p^*$, and SMLP reports its status as *not a witness.* Otherwise stability of the candidate witness $p^*, x^*$ to $\mathsf{query}$ is checked by proving validity of formula eq. (2), and this is done by checking satisfiability of formula eq. (9)

$$\theta(p^*, p') \wedge \varphi_M(p^*, x^*, y) \wedge \alpha(p^*, x^*) \wedge \neg\, \mathsf{query}(p^*, x^*, y)) \tag{9}$$

The result file format is the same as for checking stability of configuration $p^*$.

## 11.2 Querying ML model for a stable witness

Given an ML model $M$ and a query $\mathsf{query}(p, x, y)$, the task is to find a stable witness $p^*$ for $\mathsf{query}(p, x, y)$ on $M$. That is, the task is to find $p^*$ such that $\varphi_{cert}(p)$ (defined in eq. (4)) holds; and hence $p^*$ will be a solution for

$$\varphi_{\mathsf{query}} = \exists p\ \varphi_{cert}(p) = \exists p\ \left[\eta(p) \wedge \left(\forall p'\ \forall xy\ [\theta(p, p') \implies (\varphi_M(p', x, y) \implies \varphi_{cond}(p', x, y))]\right)\right] \tag{10}$$

```
{
        "query1": {
                "status": "UNSAT",
                "witness": null
        },
        "query2": {
                "status": "STABLE_SAT",
                "witness": {
                        "p1": 2.0,
                        "y1": 9.0,
                        "y2": 5.0,
                        "x1": 0.0,
                        "p2": 7.0
                }
        }
}
```

Figure 9: SMLP result in mode "query".

where

$$\varphi_{cond}(p,x,y) = \alpha(p,x) \implies \mathsf{query}(p,x,y).$$

For a solution $p^*$ to exist, and for it not to be vacuous, it is necessary $\alpha(p,x) \wedge \eta(p)$ to be consistent, meaning that the following formula must be valid:

$$\exists p, x\ [\alpha(p,x) \wedge \eta(p)]) \tag{11}$$

Furthermore, for a solution to exist, the following formula must be satisfiable, and we refer to this as a *feasibility* check for the task of searching for a stable witness for $\mathsf{query}(p,x,y)$.

$$\eta(p) \wedge \varphi_M(p,x,y) \wedge \alpha(p,x) \wedge \mathsf{query}(p,x,y)) \tag{12}$$

When both the consistency check and the feasibility check succeed, SMLP enters an iterative algorithm for searching for a stable witness $p^*$, and can terminate also by concluding that such a witness does not exist (under some additional assumptions).

SMLP supports querying multiple conditions in one SMLP run. The status of each query is reported in file $prefix\_dataname\_query\_results.json$, and the result might look like the one displayed in Figure fig. 9, where the "witness" field specifies the values of knobs, as well as values of inputs and values of outputs found in the satisfying assignment to eq. (6) that identified the stable witness.

## 11.3 Querying ML model for a concrete stable witness

The task of stable synthesis of knob values for a query $\mathsf{query}$ consists of solving the following formula:

$$\exists p, x\ \left[\eta(p) \wedge \forall p'\ \forall y\ [\theta(p,p') \implies (\varphi_M(p',x,y) \implies \varphi_{cond}(p',x,y))]\right] \tag{13}$$

Let $p^*, x^*$ be a solution to eq. (13). Then according to Definition 1, $x^*$ is a stable witness for $\mathsf{query}(p^*,x,y)$.

First, we find a candidate $p^*, x^*$ by solving

$$\exists p, x\ \left[\eta(p) \wedge \forall y\ [(\varphi_M(p,x,y) \implies \varphi_{cond}(p,x,y))]\right] \tag{14}$$

Then check whether the following formula is valid (by checking its negation for satisfiability):

$$\eta(p^*) \wedge \forall p'\ \forall y\ [\theta(p^*,p') \implies (\varphi_M(p',x^*,y) \implies \varphi_{cond}(p',x^*,y))] \tag{15}$$

If the above formula is valid, then we have shown that $x^*$ is a stable witness for $\mathsf{query}(p^*,x,y)$, and the task is accomplished – SMLP reports $p^*, x^*$ is solution to the synthesis task. Otherwise, search should continue by searching for a solution different from $p^*, x^*$ (and their neighborhood). looks like the full algorithm is same as the synthesis algorithm (interleaving of finding a candidate $p^*$ and searching to its counter-example, except now in search for candidate counter-example the inputs should be bound to values in $x^*$ so that only knobs will be free to assign values in candidate counter-example.

## 11.4 Assertion verification

In assertion verification usually one assumes that the knobs have already been fixed to legal values, and their impact has been propagated through the constraints, therefore usually in the context of assertion verification knobs are not considered explicitly. However, in order to formalize stability also in the context of verification, SMLP assumes that the values of knobs $p$ are assigned constant values $p^*$, but can be perturbed (by environmental effects or by an adversary), therefore treatment of $p^*$ is explicit. Then the problem of verifying an assertion $\mathsf{assert}(p^*,x,y)$ under allowed perturbations $p'$ of knob values $p^*$ controlled by $\theta_r(p^*,p')$, as defined in Definition 1, is exactly the problem of certification of stability of witness $p^*$ for query $\mathsf{assert}(p^*,x,y)$, as formalized in eq. (4), where now

$$\varphi_{cond}(p,x,y) = \alpha(p,x) \implies \mathsf{assert}(p,x,y).$$

Therefore, in current implementation, in mode "verify", SMLP supports verification without stability considerations, that is, assumes the stability predicate $\theta(p,p')$ on knobs $p,p'$ is the identity relation (which means, all stability radii are assumed to be 0 and therefore $\theta(p.p') = p == p'$).

Note that the consistency check eq. (5) and feasibility check eq. (6) for mode "certify" in section 11.1 do not refer to stability predicate $\theta$, and are applicable in mode "verify" in a similar way: If the consistency check fails, this is reported to user, with an expectation that the specification file needs to be fixed (this might have been caused by a type).not implemented. Otherwise, if the feasibility check fails (that is, eq. (6) with $\mathsf{assert}$ in place of $\mathsf{query}$, is unsatisfiable), then either $\mathsf{assert}$ is not valid or it is still possible that the specification is not what was intended. not implemented. Otherwise, formula eq. (16) is checked for satisfiability (recall that $\theta_r(p^*,p'$ is identity in mode "verify" and therefore it does not occur in eq. (16)):

$$\varphi_M(p^*,x,y) \wedge \alpha(p^*,x) \wedge \neg\,\mathsf{assert}(p^*,x,y)) \tag{16}$$

If eq. (16) is satsfiable, SMLP reports the assertion as "FAIL", and otherwise SMLP reports it as "“PASS". An example verification results file might look like the one in Figure fig. 10, where the field "asrt" in the result reports the value of the assertion in the SAT assignment to the negated assertion in formula eq. (16), and is displayed as part of the results as a sanity check for the correctness of the counter-example to $\mathsf{assert}(p^*,x,y)$. Assertion verification result is reported in file $prefix\_dataname\_assertions\_results.json$.rename the report file assertions –> verify, to be uniform

```
{
        "asrt1": {
                "status": "FAIL",
                "asrt": false,
                "model": {
                        "y1": 9.0,
                        "y2": 5.0,
                        "x3": 7.0,
                        "x1": 0.0,
                        "x2": 2.5
                }
        },
        "asrt2": {
                "status": "PASS",
                "asrt": null,
                "model": null
        }
}
```

Figure 10: SMLP result in mode "verify".

### 11.5 Stable synthesis mode

The task of $\theta$-stable synthesis consists of finding a solution to formula eq. (1), where

$$\varphi_{cond}(p, x, y) = \alpha(p, x) \implies (\beta(p, x, y) \wedge \mathsf{assert}(p, x, y)).$$

. That is, $\theta$-stable synthesis reduces to the task of searching for a $\theta$-stable witness $p^*$ for $\mathsf{query}(p, x, y) = \beta(p, x, y) \wedge \mathsf{assert}(p, x, y))$ (here $\mathsf{assert}(p, x, y)$ might represent a conjunction of multiple assertions).

### 11.6 Stable optimization mode

In this subsection we consider the optimization problem for a real-valued function $f$ (in our case, an ML model), extended in two ways:

1. We consider a $\theta$-stable maximum to ensure that the objective function does not drop drastically in a close neighborhood of the configuration where its maximum is achieved.
2. We assume that the objective function besides knobs depends also on inputs, and the function is maximized in the stability $\theta$-region of knobs, for any values of inputs in their respective legal ranges.

We explain these extensions using two plots in Figure 11. The left plot represents optimization problem for $f(p, x)$ when $f$ depends on knobs only (thus $x$ is an empty vector), while the right plot represents the general setting where $x$ is not empty (which is usually not considered in optimization research). In each plot, the blue threshold (in the form of a horizontal bar or a rectangle) denotes the stable maximum around the point where $f$ reaches its (regular) maximum, and the red threshold denotes the stable maximum, which is approximated by our optimization algorithms. In both plots, the regular maximum of $f$ is not stable due to a sharp drop of $f$'s value in the stability region.

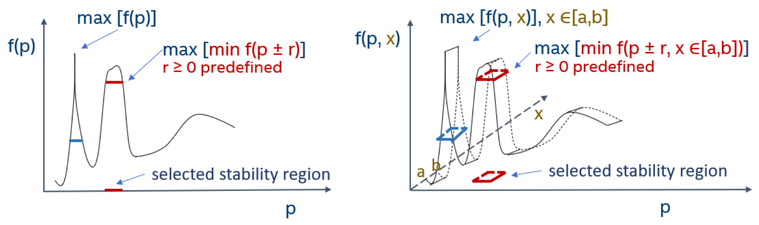

Figure 11: SMLP max-min optimization. On both plots, $p$ denote the knobs. On the right plot we also consider inputs $x$ (which are universally quantified) as part of $f$.

An example of how to run SMLP in mode optimize was given in Section 4, which among other things describes how the objectives can be defined through the command line and through the specification file. When there are multiple objectives, SMLP supports both pareto optimization as well as optimizing for each objective separately (independently from requirements of other objectives). This choice is controlled using option -pareto t/f. SMLP optimization algorithm first performs feasibility check that $\beta$ constraints are feasible under the interface constraints $\alpha$ and $\eta$, and if a solution is found, the latter is used as a first estimate of lower bounds to objective's thresholds that can be achieved, and these bounds along with input and knob values that demonstrate these bounds are (immediately) reported to optimization report file with suffix *_optimization_progress.csv. SMLP then continues search to tighten the objective's upper and lower bounds, and at anytime when lower bounds are improved (in case of maximization problem) the optimization progress report is updated with improved estimates of the optima. If the minimization search terminates under given time and memory requirements, the final results are reported in file with suffix *_optimization_results.csv. Reports *_optimization_progress.json and *_optimization_results.json are also available with more detail compared to the respective *.csv reports..

The stable optimization problem is a special case of stable optimized synthesis problem which is discussed formally in Section 11.7, and we refer the reader to it for a formal treatment of stable synthesis problem. More precisely, the problem of stable optimization is defined using Equation (19) and the problem of stable optimized synthesis problem is defined using Equation (20), from which Equation (19) is obtained as a special case, by assuming that assert$(p, x, y)$ is constant true.

## 11.7 Stable optimized synthesis mode

Let us first consider optimization without stability or inputs, i.e., far low corner in the exploration cube Figure 2. Given a formula $\varphi_M$ encoding the model, and an objective function $o : \mathcal{D}_{par} \times \mathcal{D}_{out} \to \mathbb{R}$, the standard optimization problem solved by SMLP is stated by Formula (17).

$$[[\varphi_M]]_o = \max_p \{z \mid \forall y \ (\varphi_M(p, y) \implies o(p, y) \geq z)\} \tag{17}$$

A solution to this optimization problem is the pair $(p^*, [[\varphi_M]]_o)$, where $p^* \in \mathcal{D}_{par}$ is a value of parameters $p$ on which the maximum $[[\varphi_M]]_o \in \mathbb{R}$ of the objective function $o$ is achieved for the output $y$ of the model on $p^*$. In most cases it is not feasible to exactly compute the maximum. To deal with this, SMLP computes maximum with a specified accuracy. Consider $\varepsilon > 0$. We refer to values $(\tilde{p}, \tilde{z})$ as a solution to the optimization problem with *accuracy* $\varepsilon$, or $\varepsilon$-*solution*, if $\tilde{z} \leq [[\varphi_M]]_o < \tilde{z} + \varepsilon$ holds and $\tilde{z}$ is a lower bound on the objective, i.e., $\forall y[\varphi_M(p, y) \implies o(p, y) \geq \tilde{z}]$ holds.

Now, we consider ***stable optimized synthesis***, i.e., the top right corner of the exploration cube. The problem can be formulated as the following Formula (18), expressing maximization of a lower bound on the objective function $o$ over parameter values under stable synthesis constraints.

$$[[\varphi_M]]_{o,\theta} = \max_p \{z \mid \eta(p) \wedge \forall p' \; \forall xy \; [\theta(p,p') \implies (\varphi_M(p',x,y) \implies \varphi^{\geq}_{cond}(p',x,y,z))]\} \tag{18}$$

where

$$\varphi^{\geq}_{cond}(p',x,y,z) = \alpha(p',x) \implies (\beta(p',x,y) \wedge o(p',x,y) \geq z).$$

The stable synthesis constraints are part of a GEAR formula and include usual $\eta, \alpha, \beta$ constraints together with the stability constraints $\theta$. Equivalently, stable optimized synthesis can be stated as the *max-min* optimization problem, Formula (19)

$$[[\varphi_M]]_{o,\theta} = \max_p \min_{x,p'} \{z \mid \eta(p) \wedge \forall y \; [\theta(p,p') \implies (\varphi_M(p',x,y) \implies \varphi^{\leq}_{cond}(p',x,y,z))]\} \tag{19}$$

where

$$\varphi^{\leq}_{cond}(p',x,y,z) = \alpha(p',x) \implies (\beta(p',x,y) \wedge o(p',x,y) \leq z).$$

In Formula (19) the minimization predicate in the stability region corresponds to the universally quantified $p'$ ranging over this region in (18). An advantage of this formulation is that this formula can be adapted to define other aggregation functions over the objective's values on stability region. For example, that way one can represent the *max-mean* optimization problem, where one wants to maximize the mean value of the function in the stability region rather one the min value (which is maximizing the worst-case value of $f$ in stability region). Likewise, Formula (19) can be adapted to other interesting statistical properties of distribution of values of $f$ in the stability region.

We explicitly incorporate assertions in stable optimized synthesis by defining $\beta(p',x,y) = \beta'(p',x,y) \wedge \mathsf{assert}(p',x,y)$ in $\varphi^{\geq}_{cond}(p',x,y,z)$ of Equation (19), where $\mathsf{assert}(p',x,y)$ are assertions required to be valid in the entire stability region around the selected configuration of knobs $p$:

$$[[\varphi_M]]_{o,\theta} = \max_p \min_{x,p'} \{z \mid \eta(p) \wedge \forall y \; [\theta(p,p') \implies (\varphi_M(p',x,y) \implies \varphi^{\leq}_{cond}(p',x,y,z))]\} \tag{20}$$

where

$$\varphi^{\leq}_{cond}(p',x,y,z) = \alpha(p',x) \implies (\beta(p',x,y) \wedge \mathsf{assert}(p',x,y) \wedge o(p',x,y) \leq z).$$

The notion of $\varepsilon$-solutions for these problems carries over from the one given above for Formula (17).

SMLP implements stable optimized synthesis based on the GearOPT$_\delta$ and GearOPT$_\delta$-BO algorithms [BKK20, BKK22], which are shown to be complete and terminating for this problem under mild conditions. These algorithms were further extended in SMLP to Pareto point computations to handle multiple objectives simultaneously.

SMLP invocation in optsyn mode is similar to running SMLP in optimize mode, with and example of the latter mode discussed in Section 4, with the only difference that the mode is specified as -mode optsyn, and the specification file or the command line for the optsyn mode should contain specification for both assertions and objectives (while in optimize mode assertion specification is not required.) Similarly to the optimize mode, optimization progress and final results are reported in files with suffix *_optimization_progress. {json|csv} and *_optimization_results. {json|csv}. When optsyn is not feasible due to infeasible beta constraints or assertions, no csv/json reports are written ... TODO

## 12 Design of experiments

Most DOE methods are based on understanding multivariate distribution of legal value combinations of inputs and knobs in order to sample the system. When the number of system inputs and/or knobs is large (say hundreds or more), the DOE may not generate a high-quality coverage of the system's behavior to enable training models with high accuracy. Model training process itself becomes less manageable when number of input variables grows, and models are not explainable and thus cannot be trusted. One way to curb this problem is to select a subset of input features for DOE and for model training. The problem of combining feature selection with DOE generation and model training is an important research topic of practical interest, and SMLP supports multiple practically proven ways to select subsets of features and feature combinations as inputs to DOE and training, including the *MRMR* feature selection algorithm [DP05], and a *Subgroup Discovery (SD)* algorithm [Klö96, Wro97, Atz15]. The MRMR algorithm selects a subset of features according to the principle of *maximum relevance and minimum redundancy.* It is widely used for the purpose of selecting a subset of features for building accurate models, and is therefore useful for selecting a subset of features to be used in DOE; it is a default choice in SMLP for that usage. The SD algorithm selects regions in the input space relevant to the response, using heuristic statistical methods, and such regions can be prioritized for sampling in DOE algorithms.

## 13 Root cause analysis

We view the problem of root cause analysis as dual to the stable optimized synthesis problem: while during optimization with stability we are searching for regions in the input space (or in other words, characterizing those regions) where the system response is good or excellent, the task of root-causing can be seen as searching for regions in the input space where the system response is not good (is unacceptable). Thus simply by swapping the definition of excellent vs unacceptable, we can apply SMLP to explore weaknesses and failing behaviors of the system.

Even if a number of witnesses (counter-examples to an assertion) are available, they represent discrete points in the input space and it is not immediately clear which value assignments to which variables in these witnesses are critical to explain the failures. Root causing capability in SMLP is currently supported through two independent approaches: a *Subgroup Discovery (SD)* algorithm that searches through the data for the input regions where there is a higher ratio (thus, higher probability) of failure; to be precise, SD algorithms support a variety of *quality functions* which play the role of optimization objectives in the context of optimization. To find input regions with high probability of failure, SMLP searches for stable witnesses to failures. These capabilities, together with feature selection algorithms supported in SMLP, enable researchers to develop new root causing capabilities that combine formal methods with statistical methods for root cause analysis.

## 14 Model refinement loop

Support in SMLP for selecting DOE vectors to sample the system and generate a training set was discussed in Subsection 12. Initially, when selecting sampling points for the system, it is unknown which regions in the input space are really relevant for the exploration task at hand. Therefore some DOE algorithms also incorporate random sampling and sampling based on previous experience and familiarity with the design, such as sampling nominal cases and corner cases, when these are known. For model exploration tasks supported by SMLP, it is not required to train a model that will be an accurate match to the system everywhere in the legal search space of inputs and knobs. We require to train a model that is an *adequate* representation of the system for the task at hand, meaning that the exploration task solved on the model solves this task for the system as well. Therefore SMLP supports a *targeted model refinement* loop to enable solving the system exploration tasks by solving these tasks on the model instead. The idea is as follows: when a stable solution to model exploration task is found, it is usually the case that there are not many training data points close to the stability region of that solution. This implies that there is a high likelihood that the model does not accurately represent the system in the stability region of the solution. Therefore the system is sampled in the stability region of the solution, and these data samples are added to the initial training data to retrain the model and make it more adequate in the stability region of interest. Samples in the stability region of interest can also be assigned higher weights compared to other samples to help training to achieve higher accuracy in that region. More generally, higher adequacy of the model can be achieved by sampling distributions biased towards prioritizing the stability region during model refinement.

Note that model refinement is required only to be able to learn properties on the system by exploring these properties on the model. As a simple scenario, let us consider that we want to check an assertion $\mathsf{assert}(p, x, y)$ on the system. If a $\theta$-stable counter example $x^*$ to $\mathsf{assert}(p, x, y)$ for a configuration $p^*$ of knobs exists on the model according to Definition 1 (which means that $x^*$ is a $\theta$-stable witness for query $\mathsf{query}(p^*, x, y) = \neg\, \mathsf{assert}(p^*, x, y)$), then the system is sampled in the $\theta$–stability region of $p^*$ (possibly with $x^*$ toggled as well in a small region around $x^*$). If the failure of assertion $\mathsf{assert}(p, x, y)$ is reproduced on the system using the stable witness $x^*$ and a configuration in the $\theta$-stability region of $p^*$, then the model exploration goal has been accomplished (we found a counter-example to $\mathsf{assert}(p, x, y)$ on the system) and model refinement can stop. Otherwise the system samples can be used to refine the model in this region. For that reason, the wider the $\theta$-stability region, the higher the chances to reproduce failure of assertion $\mathsf{assert}(p, x, y)$ on the system.

On the other hand, if the $\mathsf{assert}(p, x, y)$ does not have a $\theta$-stable counter-example, then it is still possible that $\mathsf{assert}$ is not valid on the system but it cannot be falsified on the model due to discrepancy between the system and model responses in some (unknown to us) input space. In this case one can strengthen $\mathsf{assert}(p, x, y)$ to $\mathsf{assert}'(p, x, y)$ (for example, assertion $\mathsf{assert}(p, x, y) = y \geq 3$ can be strengthened to $\mathsf{assert}'(p, x, y) = y \geq 3.01$), or one can weaken the stability condition $\theta_r$ to $\theta_{r'}$ by shrinking the stability radii $r$ to $r'$; or do both. If the strengthened assertion $\mathsf{assert}'(p, x, y)$ has a $\theta_{r'}$-stable counter-example $x'$ for a configuration $p'$ on the model, then $p', x'$ can be used to find a counter-example to the original assertion $\mathsf{assert}(p, x, y)$ on the system in the same way as with $p^*, x^*$ before. If failure of the original assertion $\mathsf{assert}$ is reproduced on the system, the model refinement loop stops, otherwise it can continue using other strengthened versions of the original assertion and or shrinking the stability radii even further. Similar reasoning is applied to other modes of design space exploration. In particular, in the optimization modes, one can confirm or reject on the system an optimization threshold proved on the model, and in the latter case the model refinement loop will be triggered by or providing new data-points that can be used for model refinement in this region, through re-training or incremental training of a new model.

## References

[Atz15] Martin Atzmueller. Subgroup discovery. *WIREs Data Mining Knowl. Discov.*, 5(1):35–49, 2015.

[BKK20] Franz Brauße, Zurab Khasidashvili, and Konstantin Korovin. Selecting stable safe configurations for systems modelled by neural networks with ReLU activation. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*, pages 119–127. IEEE, 2020.

[BKK22] Franz Brauße, Zurab Khasidashvili, and Konstantin Korovin. Combining constraint solving and bayesian techniques for system optimization. In Luc De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 1788–1794. ijcai.org, 2022.

[BKK24] Franz Brauße, Zurab Khasidashvili, and Konstantin Korovin. Smlp: Symbolic machine learning prover, 2024.

[BW51] G. E. P. Box and K. B. Wilson. On the experimental attainment of optimum conditions. *Journal of the Royal Statistical Society. Series B (Methodological)*, 13(1):1–45, 1951.

[DP05] Chris H. Q. Ding and Hanchuan Peng. Minimum redundancy feature selection from microarray gene expression data. *J. Bioinform. Comput. Biol.*, 3(2):185–206, 2005.

[Klö96] Willi Klösgen. Explora: A multipattern and multistrategy discovery assistant. In Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 249–271. AAAI/MIT Press, 1996.

[MSK21] Alex Manukovsky, Yuriy Shlepnev, and Zurab Khasidashvili. Machine learning based design space exploration and applications to signal integrity analysis of 112Gb SerDes systems. In *2021 IEEE 71st Electronic Components and Technology Conference (ECTC)*, pages 1234–1245, 2021.

[Wro97] Stefan Wrobel. An algorithm for multi-relational discovery of subgroups. In Henryk Jan Komorowski and Jan M. Zytkow, editors, *Principles of Data Mining and Knowledge Discovery, First European Symposium, PKDD '97, Trondheim, Norway, June 24-27, 1997, Proceedings*, volume 1263 of *Lecture Notes in Computer Science*, pages 78–87. Springer, 1997.

## A Concrete Formats

### A.1 CSV

A structured line-based text format where the first line describes a list of column headers and each line after it describes a list the values corresponding to the respective column in the data set. A line describes a list of elements $(e_i)_{i=1}^n$, if it corresponds to the concatenation $s(e_1)\texttt{,}s(e_2)\texttt{,}\ldots\texttt{,}s(e_n)$ where $s(v)$ is the ASCII representation of $v$.

The list of column headers shall not contain duplicates. The ASCII representation of ASCII strings made up of printable characters excluding "`,`" is the identity. Floating point approximations are represented as the decimal encoding where "`.`" is the separator between integer and fractional part. Rational numbers $\frac{p}{q}$ are represented by $s(p)\texttt{/}s(q)$ where $s(z)$ is the decimal encoding of $|z|$ prefixed by "`-`" if $z \in \mathbb{Z}$ is negative.