SMLP: Symbolic Machine Learning Prover (User Manual)

Franz Brauße[1], Zurab Khasidashvili[2], and Konstantin Korovin[3]

[1,3]The University of Manchester, UK
[2]Intel, Israel

September 1, 2025

Abstract

SMLP: Symbolic Machine Learning Prover is an open source tool for exploration and optimization of systems represented by machine learning models.[1] SMLP uses symbolic reasoning for ML model exploration and optimization under verification and stability constraints, based on SMT, constraint and NN solvers. In addition its exploration methods are guided by probabilistic and statistical methods.

SMLP is a general purpose tool that requires only data suitable for ML modelling in the csv format (usually samples of the system's input/output). SMLP has been applied at Intel for analyzing and optimizing hardware designs at the analog level. Currently SMLP supports NNs, polynomial and tree models, and uses SMT solvers for reasoning and optimization at the backend, integration of specialized NN solvers is in progress. Key algorithms behind SMLP are described in detail in [BKK22, 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.[2]

Contents

[1]SMLP is available at: `https://github.com/fbrausse/smlp`
[2]`https://www.apache.org/licenses/LICENSE-2.0`

1 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 9. 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. Parameter optimization under safety constraints is one of the main applications of SMLP.

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.

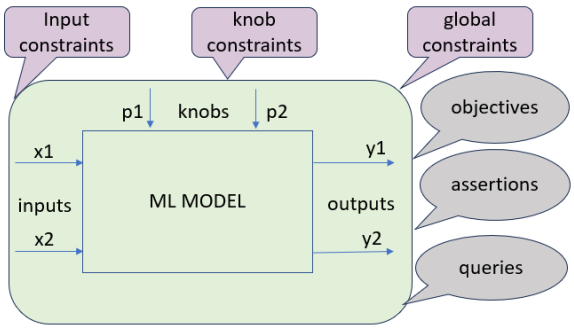

Figure 1: Parameterized system, with interface constraints

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 a machine learning model (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 ($\searrow$-axis), optimization ($\rightarrow$-axis) and stability ($\uparrow$-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 9.

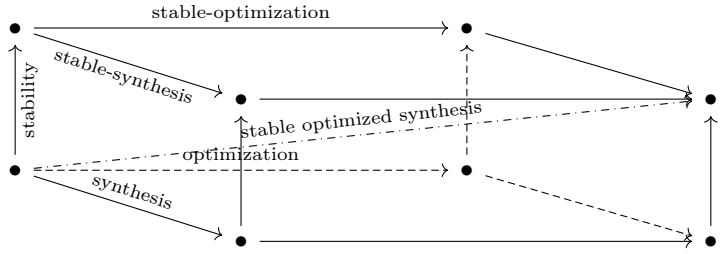

Figure 2: Exploration Cube

2 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 Method-*

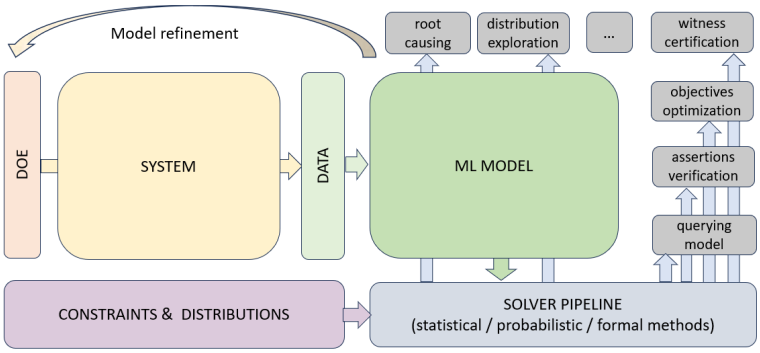

Figure 3: SMLP Tool Architecture

ology (RSM) [BW51], and since then has been widely adopted by the industry. Neural networks 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 9. 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 12.

3 How to run SMLP: a quick start

Command to run SMLP in optimize mode is given in Figure 4. Note that all concrete examples in this manual will be executed from the sub-directory `regr_smlp/code` of the SMLP distribution.

```
../../src/run_smlp.py -data "../data/smlp_toy_basic" -out_dir ./ -pref Test113 \
-mode optimize -pareto t -resp y1,y2 -feat x1,x2,p1,p2 -model dt_sklearn \
-dt_sklearn_max_depth 15 -mrmr_pred 0 -epsilon 0.05 -delta_rel 0.01 -save_model t \
-model_name test113_model -save_model_config t -plots f -seed 10 -log_time f \
-spec ../specs/smlp_toy_basic.spec
```

Figure 4: Example SMLP command in mode optimize, to build a decision tree model and perform an optimization task.

The option -data ../smlp_toy_basic defines the labeled dataset to use for model training and test. The dataset should be provided as smlp_toy_basic.csv file; the .csv suffix itself may be omitted, zip and bzip2 compressed data files are also accepted. This dataset is displayed in Table 1, and it has six columns $x1, x2, p1, p2, y1, y2$.

	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.

The option -mode optimize defines the analysis mode to run, and option -pareto t instructs 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.spec 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 5.

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_depth 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 ϵ and δ 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 ϵ defines a termination criterion for search for optima, and is used to guarantee that the computed optima are not more than ϵ away (after scaling the objectives) from the real optima (of the function defined by the ML model). A formal description of usage of ϵ can be found in Section 9 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 8.

Option -mrmr_pred 0 specifies that all features should be used for training a model, while

```
{
  "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"
  }
}
```

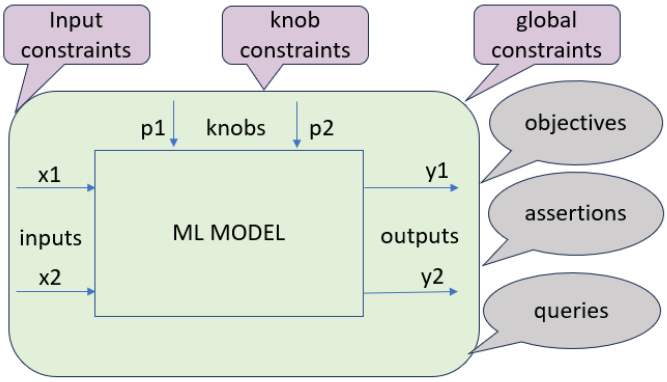

Figure 5: Specification smlp_toy_basic.spec used by SMLP command in Figure fig. 4.

option value greater than 0 defines how many features selected by the MRMR algorithm should be used for model training (see also Section 7.2.3).

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 8 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 6.2 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 -pref 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. See, Section 9.6 for more details on SMLP reports for mode optimize.

4 Symbolic representation of the ML model exploration

The main system exploration tasks handled by SMLP can be defined using the GEAR-fragment of $\exists^*\forall^*$ formulas [BKK20]:

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

where x ranges over inputs, y ranges over outputs, and p, p' range over knobs, $\eta(p)$ are constraints on the knob configuration p, $\varphi_M(p',x,y)$ defines the machine learning model, $\theta(p,p')$ defines stability region for the solution p, and $\varphi_{cond}(p',x,y)$ defines conditions that should hold in the stability region.

In our formalization θ, η and φ_{cond} are quantifier free formulas in the language. These constraints and how they are implemented in SMLP are described below.

$\eta(p)$ Constraints on values of knobs; this formula need not be a conjunction of constraints on individual knobs, can define more complex relations between allowed knob values of individual knobs. $\eta(p)$ can be specified through the SMLP specification file (see Section 5).

$\theta(p,p')$ Stability constraints that define a region around a candidate solution. This can be specified using either absolute or relative radius r in the specification file. This region corresponds to a ball (or box) around p: $\theta(p,p') = \|p - p'\| \leq r$, also denoted as $\theta_r(p)$, in this case was say p is the center point of the region defined by $\theta_r(p)$. In general, our methods do not impose any restrictions on θ apart from reflexivity.

$\varphi_M(p,x,y)$ Constraints that define the function represented by the ML model M, thus $\varphi_M(p,x,y) = (M(p,x) = y)$. In the ML model knobs are represented as designated inputs (and can be treated in the same way as system inputs, or the machine model architecture can reflect the difference between inputs and knobs). $\varphi_M(p,x,y)$ is computed by SMLP internally, based on the ML model specification.

$\varphi_{cond}(p,x,y)$ Conditions that should hold in the θ-region of the solution. These conditions depend on the exploration mode and could be: (1) verification conditions, (2) model querying conditions, (3) parameter optimization conditions, or (4) parameter synthesis conditions. The exploration modes are described in Section 9.

SMLP solver is based on specialized procedures for solving formulas in the GEAR fragment using quantifier-free SMT solvers, GearSAT$_\delta$ [BKK20] and GearSAT$_\delta$-BO [BKK22]. The GearSAT$_\delta$ procedure interleaves search for candidate solutions using SMT solvers with exclusion of θ-regions around counterexamples. GearSAT$_\delta$-BO combines GearSAT$_\delta$ search with Bayesian optimization guidance. These procedures find solutions to GEAR formulas with user-defined accuracy ε and they have been proven to be sound, (δ)-complete and terminating.

5 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 3 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). Values - inf and inf are allowed as the min and max of the range, and can be specified using null. e.g. [-2, null]. The input and knob ranges serve as assumptions in model exploration modes of SMLP.

"rad-abs" absolute stability radius for knobs.

"rad-rel" relative (wrt, to the center point of the region) stability radius for knobs. Only one type of radii is needed per parameter.

"grid" for knobs, which is a list of values that a knob variable is allowed to take within the respective declared ranges, independently from other knobs. The η 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.

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

"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.

"beta" defines 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[3]. These constraints are formally introduced in Section 9.

In SMLP these expressions are parsed using the Abstract Syntax Trees library[4]. 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]

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]

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

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

if-then-else x if cond else y [lte(cond, x, y)]

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 9.

6 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 6.1. All outputs from an SMLP run, on the other hand, are written into the same output directory, as described in Subsection 6.2.

6.1 SMLP inputs

training data should be a .csv file, possibly compressed as .csv.bz2 or .csv.gz. Full or relative path to data should be specified using option -data; the .csv suffix can be omitted in case the data file is not compressed. 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, possibly compressed as .csv.bz2 or .csv.gz. Full or relative path to data should be specified using option -new_data; the .csv suffix can be omitted in case the data file is not compressed. This data usually is 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 a model trained on 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 or relative path to spec file, including the .spec suffix, should be specified using option -spec. It is required in model exploration modes (certify, query, verify, synthesize, optimize, optsyn).

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

6.2 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 model 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.

7 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.

7.1 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.

7.1.1 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.

7.1.2 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.

7.1.3 Constant features

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

7.1.4 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).

7.1.5 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'.

7.1.6 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 analysis). 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.

7.2 Data preparation for analysis

We now describe data preparation steps supported in SMLP.

7.2.1 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, and integer.

7.2.2 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.[5] SMLP option -mrmr_pred 15 instructs the MRMR algorithm to select 15 features, according to the principle of *maximum relevance and minimum redundancy*.

7.2.3 Feature compression for model training

SMLP has several feature selection and transormation techniques incorporated, it can choose principal input features and reduce dimensionality before model training. The following command line arguments dictate feature selection:

- `-feat_select_model`: Determines the method for feature selection. Options are
  - `mrmr`:(Maximum Relevance Minimum Redundancy) Use MRMR feature selection through the mrmr.[6] Python package. This method selects features based upon maximum relevance with the dependent variable and minimum redundancy with other selected features.

[5] `https://github.com/nlhepler/mrmr`.
[6] `https://github.com/nlhepler/mrmr`.

  - `shap`: Use SHAP (SHapley Additive exPlanations) feature selection through the `shap`.[7] package; this ranks feature importance through SHAP values generated in a Random Forest model.
  - `mrmr_shap`: Use this if you want to do this in two steps. First, use MRMR to select features, then apply SHAP to features selected by MRMR.
  - `mrmr_shap_ensemble`: Computes feature importance through MRMR and SHAP independently, rescales absolute values to be bounded to [0, 1] takes the mean and adds them together to generate one overall score for each selected features. The features chosen are the highest ranked following this scoring method.

- `-feat_select_count`: A parameter that determines how many features to retain after feature selection. For example:

For example:

```
-feat_select_model mrmr_shap_ensemble -feat_select_count 15
```

this will choose the top fifteen features via the ensemble scored approach from MRMR and SHAP. Furthermore, SMLP includes other features not just limited to selection, there is an optional feature transformation via PCA. This is determined by:

- `-pca_pred`: Determines the number of principal components to create using `PCA` from the scikit-learn library.[8] . PCA projects the selected features into lower dimension space retaining maximum variance at the same time. For example:

  ```
  -pca_pred 5
  ```

  would create 5 PCA components from the final feature selection.

  Note: PCA transformation is functional for `predict` mode, with components generated and used correctly. However, `optimise` mode is not yet supported, as handling input and knob bounds in the transformed PCA space is still under development.

These methods can be used individually or combined sequentially within SMLP to optimize feature selection and compression tailored to specific dataset characteristics and modeling requirements.

7.2.4 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 subgroup discovery, 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.

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 each individual objective function takes on samples in the training data.

[7] https://github.com/shap/shap

[8] `https://github.com/scikit-learn/scikit-learn/blob/main/sklearn/decomposition/_pca.py`

7.3 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.

7.4 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.

8 ML model training and prediction

In this section we describe SMLP modes train and predict: how to train ML models with SMLP, how to save them, and how to rerun saved models on new data are decried in Section 8.1, Section 8.2, and Section 8.3, respectively. Reports and other collateral files generated in train and predict modes is discussed in Section 8.4.

8.1 Training ML models

SMLP supports training tree-based and polynomial models using the scikit-learn[9] and pycaret[10] packages, and training neural networks using the Keras package with TensorFlow[11]. 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

[9] https://scikit-learn.org/stable/
[10] https://pycaret.org
[11] 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 SMLP command to train a polynomial model and perform prediction on new data.

-model_per_response). Supporting these two options allows a trade-off between the accuracy of 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 command for mode train is similar: the mode is specified using -mode train; and specifying new data with option -new_data is not required.

8.2 Saving ML models

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.

8.3 Rerunning ML models

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).

```
../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 commands to use a saved mode to perform prediction on new data.

8.4 ML model training and prediction reports

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. Recall that new data is available in mode predict and is not available in mode train, and may or may not be available in model exploration modes.

- *_{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 (when available). 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 (when available). It is saved as a .csv file. 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: response value distribution and prediction plots that display real vs predicted values respectively for training, test, labeled and new data (when available), for the model trained respectively using dt_sklearn, poly_sklearn, nn_keras, or other regression model training algorithms supported in SMLP. Generation of response value distribution plots and prediction accuracy plots are controlled using options -resp_plots t and -pred_plots t, respectively. When generated, these plots are saved for an offline review, and can also be displayed during SMLP execution, for an interactive review, using option -plots t.

- *_{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 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.json, 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 7.4.

9 ML model exploration with SMLP

SMLP supports the following model exploration modes (we assume that an ML model M has already been trained). Precise descriptions of these modes are given in the subsequent subsections.

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

query Given an ML model M and a query $\mathsf{query}(p, x, y)$, find a value assignment p^*, x^* to knobs p and to inputs x that serves as a stable witness for $\mathsf{query}(p, x, y)$ on M. Multiple queries can be evaluated in a single SMLP run.

verify Given an ML model M, a configuration p^* of knobs p (that is, a value assignment p^* to knobs p), and an assertion $\mathsf{assert}(p, x, y)$, verify whether $\mathsf{assert}(p', x, y)$ is valid on model M for any assignment p' to knobs in the stability region of p^* and all legal values of inputs x. SMLP supports verifying multiple assertions in a single run.

synthesize Given an ML model M, find a configuration p^* of knobs p such that all required constraints, including assertions, are valid for any configuration p' of knobs in the stability region of p^* and any legal values of inputs x.

optimize Given an ML model M, find a stable configuration p^* of knobs p 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 configuration p^* of knobs p that yields a Pareto-optimal values of the optimization objectives and such that all constraints and assertions are valid for any configuration p' of knobs in the stability region of p^* and legal values of inputs x. This mode is a union of the *optimize* and *synthesize* modes, its full name is *optimized synthesis*.

Table 2 summarizes components relevant to verification, synthesis and optimization in their regular meaning, and the above listed model exploration modes in SMLP. These features include relevance of inputs, knobs, stability, constraints, as well as queries, assertions and objectives, to particular model exploration modes in SMLP. Algorithms for all other modes can be seen as a sub-procedures of optimized synthesis algorithm.

9.1 Exploration basic concepts

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:

$$\exists p \ \left[\eta(p) \wedge \forall p' \ \forall xy \ [\theta(p, p') \implies (\varphi_M(p', x, y) \implies \varphi_{cond}(p', x, y))]\right] \tag{2}$$

mode / feature	inputs	knobs	α	β	η	θ	δ/ϵ	queries	assertions	objectives
verification	yes	no	yes	no	no	no	no	no	yes	no
synthesis	yes	yes	yes	yes	yes	no	no	no	yes	no
optimization	no	yes	yes	yes	yes	no	no	no	no	yes
SMLP-certify	yes	yes	yes	no	yes	yes	no	yes	no	no
SMLP-query	yes	yes	yes	no	yes	yes	no	yes	no	no
SMLP-verify	yes	yes	yes	no	yes	yes	no	no	yes	no
SMLP-synthesize	yes	yes	yes	yes	yes	yes	no	no	yes	no
SMLP-optimize	yes	yes	yes	yes	yes	yes	yes	no	no	yes
SMLP-optsyn	yes	yes	yes	yes	yes	yes	yes	no	yes	yes

Table 2: Mapping SMLP modes to relevant features

where p denotes model knobs, x denotes inputs, y denotes the outputs, $\eta(p)$ defines constraints on the knobs, $\theta(p,p')$ defines the stability region, $\varphi_M(p',x,y)$ defines the ML model constraints, and the condition $\varphi_{cond}(p',x,y)$ depends on the SMLP mode and will be discussed in subsections below. $\eta(p)$ can be constraints on individual knobs or more complex constraints defining relationships between knobs. The stability region $\theta(p,p')$ can be any reflexive predicate and in SMLP we use $\theta_r(p,p') = \|p-p'\| \leq r$, 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* θ-stable witness *to $\varphi_{post}(p,x,y)$ for configuration p^* if the following formula is valid:*

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

where

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

Here, $\varphi_{post}(p,x,y)$ can be either query, assert *or β, described below.*

- *A value assignment p^* to knobs p is called a* θ-stable configuration for *$\varphi_{post}(p,x,y)$ if any legal value assignment x^* to inputs x is a θ-stable witness to $\varphi_{post}(p,x,y)$ for configuration p^*; that is, when the following formula is valid:*

$$\varphi^{p}_{verify}(p^*) = \eta(p^*) \wedge \left(\forall p'\ \forall xy\ [\theta(p^*,p')\implies(\varphi_M(p',x,y)\implies\varphi_{cond}(p',x,y))]\right) \quad (4)$$

- *If x^* is a θ-stable witness for $\varphi_{post}(p,x,y)$ for configuration p^*, then we call x^* a* θ-stable counter-example to assertion $\mathsf{assert}(p^*,x,y) = \neg\,\varphi_{post}(p^*,x,y)$, for configuration p^*.

Equation (3) corresponds to mode certify in SMLP, and Equation (4) corresponds to mode verify, and they will be discussed in detail in Subsections 9.2 and 9.4, respectively. The concept of *θ-stable counter-example* to an assertion is relevant for targeted model refinement loop for verifying assertions, and is discussed in Section 12.

The *interface consistency check* is defined as simply checking satisfiability of:

$$\alpha(p,x) \wedge \eta(p). \quad (5)$$

The *model consistency* check augments the interface consistency check with consistency of the constraints $\varphi_M(p,x,y)$ defining the ML model M in conjunction with α and η constraints: model consistency check is defined as checking satisfiability of:

$$\alpha(p,x) \wedge \eta(p) \wedge \varphi_M(p,x,y). \tag{6}$$

These two consistency checks are common for all model exploration modes, because when one of these checks fails, then model exploration task is not well defined.

9.2 Mode certify: certification of a stable witness

In the certify mode, SMLP is given an assignment p^*, x^* to knobs p and inputs x, a query $\mathsf{query}(p,x,y)$ on an ML model M, and we want to check whether x^* is a stable witness to $\mathsf{query}(p,x,y)$ for p^*, as defined in Definition 1: certification requires checking validity of eq. (3), with $\varphi_{post}(p,x,y) = \mathsf{query}(p,x,y)$ in $\varphi_{cond}(p,x,y)$. Currently SMLP assumes that all knobs in p and all inputs in x are assigned concrete values in p^*, x^*. This requirement can be relaxed.

Example 2 *(Running Example) Let's assume that function $y = f(p,x)$ defined in eq. (7), depicted in Figure 8, represents a model with knob p, input x, and output y, with constraint $\alpha = -2 \le p \le 2 \wedge -1 \le x \le 1$. Let's consider three concrete values for p within its range $[-2,2]$: $p^*_- < 0$, $p^*_0 = 0$, and $p^*_+ > 0$, and two concrete values $x^*_- < 0$ and $x^*_+ > 0$ for x in its range $[-1,1]$. And let $\mathsf{query}(p,x,y) = y \le 0$. Then x^*_- is a θ_r-stable witness to* query *for p^*_-, for any $0 < r \le |p^*_-|$; x^*_- is a θ_0-stable witness to* query *for p^*_0 (meaning, x^*_- satisfies* query *and is θ_r-stable witness for $r = 0$), but is not a θ_r-stable witness for any $r > 0$ (because* query *evaluates to* false *for positive values of p, and θ_r-stability region of p^*_0 contains legal positive values of p for any $r > 0$); and x^*_- is not a witness to* query *for p^*_+. Finally, x^*_+ is not a witness, and therefore not a stable witness, for any legal value of p (because* query *evaluates to* false *for positive values of x).*

$$y = f(p,x) = \begin{cases} 0 & p \le 0 \wedge x \le 0 \\ x & x > 0 \\ p & otherwise \end{cases} \tag{7}$$

SMLP supports certification of multiple queries in a single run, and each query is certified with respect to its corresponding witness (different queries might refer to different witnesses). The general (and the recommended) way of defining a witness per query is using the "witnesses" field in SMLP spec file. The "witnesses" field is a dictionary with query names as keys and the respective values are dictionaries assigning a concrete value to each knob and each input. An example is displayed in Figure 9 where say "query_stable_witness" is the name of a query, "p1" and "p2" are names of knobs, and "x" is the name of the input.

When the same witness is certified against all queries, and witnesses per query are not defined using the "witnesses" field, SMLP applies a sanity check to see whether unique values p^* to knobs p and unique values x^* to inputs x can be inferred from knob and input ranges and knob grids specified in the spec file using the "variables" field. When values p^*, x^* cannot be inferred this way, SMLP aborts certification with a message clarifying the reason.

Below $\vartheta_{p^*x^*}(p,x)$ denotes the formula inferred from value assignments to p,x in a given witness to a query when the witness is specified using the "witnesses" field in the spec file, and is constant true otherwise. For example, if p and x are single variable vectors, $p^* = 3$ and $x^* = 2$, then $\vartheta_{p^*x^*}(p,x) = (p = 3 \wedge x = 2)$. (In the spec file we use Python == in place of =.)

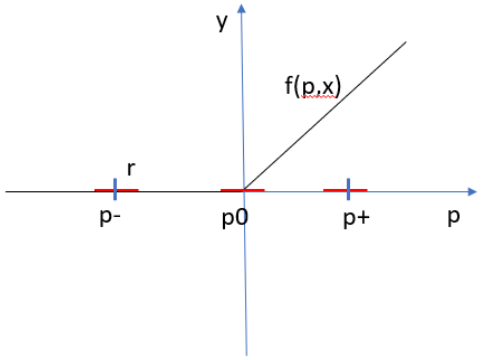

Figure 8: Model exploration running example, x is projected away to simplify the graph.

```
"witnesses": {
        "query_stable_witness": {
                "x": 7,
                "p1": 7.0,
                "p2": 6.000000067055225
        },
        "query_grid_conflict": {
                "x": 6.2,
                "p1": 3.0,
                "p2": 6.000000067055225
        },
        "query_unstable_witness": {
                "x": 7,
                "p1": 7.0,
                "p2": 6.0
        },
        "query_infeasible_witness": {
                "x": 7,
                "p1": 7.0,
                "p2": 6.0
        }
}
```

Figure 9: SMLP example of specifying witnesses in mode certify.

As discussed in Section 9.1, the interface consistency eq. (5) and model consistency eq. (6) checks are performed before starting actual certification of witnesses for stability with respect to the corresponding queries. In addition to these, for the certification problem in eq. (3) to be well defined, we need to check that $\eta(p^*) \wedge \alpha(p^*, x^*)$ evaluates to constant true: this means that p^*, x^* witnesses that $\alpha(p, x) \wedge \eta(p)$ is *consistent*. To re-iterate, the *witness consistency check* for the certification task consists in checking satisfiability of:

$$\vartheta_{p^* x^*}(p, x) \wedge \alpha(p, x) \wedge \eta(p) \tag{8}$$

In the implementation, we are using a stronger version of the witness consistency check eq. (8), which in addition takes into account the ML model constraints, and consists in checking satisfiability of:

$$\vartheta_{p^* x^*}(p, x) \wedge \alpha(p, x) \wedge \eta(p) \wedge \varphi_M(p, x, y) \tag{9}$$

Note that none of the above consistency checks refers to an actual query $\mathsf{query}(p, x, y)$ for which certification is performed. The *feasibility* part for this task is checking satisfiability of

$$\vartheta_{p^* x^*}(p, x) \wedge \alpha(p, x) \wedge \eta(p) \wedge \varphi_M(p, x, y) \wedge \mathsf{query}(p, x, y)) \tag{10}$$

If the above formula is not satisfiable, then x^* cannot be a stable witness to $\mathsf{query}(p, x, y)$ for p^*. Otherwise stability of the candidate witness p^*, x^* to $\mathsf{query}(p, x, y)$ is checked by proving validity of eq. (3), and this is done by checking satisfiability of eq. (11):

$$\vartheta_{p^* x^*}(p, x) \wedge \eta(p) \wedge \theta(p, p') \wedge \alpha(p', x) \wedge \varphi_M(p', x, y) \wedge \neg\, \mathsf{query}(p', x, y)) \tag{11}$$

An example command to run SMLP in certify mode is given in Figure 10.

```
../../src/run_smlp.py -data ../data/smlp_toy_ctg_num_resp -out_dir ./ -pref Test128 \
-mode certify -resp y1,y2 -feat x,p1,p2 -model poly_sklearn -dt_sklearn_max_depth 15 \
-save_model f -use_model f -model_per_response f -quer_names \
query_stable_witness,query_grid_conflict,query_unstable_witness,query_infeasible_witness\
-quer_exprs "y2<=90;y1>=9;y1>=(-10);y1>9" -plots f -seed 10 -log_time f  \
-spec ../specs/smlp_toy_witness_certify.spec
```

Figure 10: Example SMLP command in mode certify.

Certification results are reported in file prefix_dataname_certify_results.json. Figure 11 displays an example results file in certify mode.

1. Fields "smlp_execution", "interface_consistent" and "model_consistent" are common for each (witness, query) pair, and they provide status of the entire execution of SMLP.
2. The field "witness_consistent" specifies result of witness consistency check eq. (9)
3. The field "witness_feasible" specifies result of witness feasibility check eq. (10)
4. The field "witness_status" specifies the witness certification status, and can be one of:

"ERROR" when interface consistency eq. (5), model consistency eq. (6), or witness consistency eq. (9) check fails.

"PASS" when interface consistency eq. (5), model consistency eq. (6), configuration consistency eq. (9) and witness validity eq. (3) are all valid.

```
{
        "query_stable_witness": {
                "witness_consistent": "true",
                "witness_feasible": "true",
                "witness_stable": "true",
                "witness_status": "PASS"
        },
        "query_grid_conflict": {
                "witness_consistent": "false",
                "witness_feasible": "false",
                "witness_stable": "false",
                "witness_status": "ERROR"
        },
        "query_unstable_witness": {
                "witness_consistent": "true",
                "witness_feasible": "true",
                "witness_stable": "false",
                "witness_status": "FAIL"
        },
        "query_infeasible_witness": {
                "witness_consistent": "true",
                "witness_feasible": "false",
                "witness_stable": "false",
                "witness_status": "FAIL"
        },
        "smlp_execution": "completed",
        "interface_consistent": "true",
        "model_consistent": "true"
}
```


Figure 11: SMLP result in mode certify.

"FAIL" when interface consistency eq. (5), model consistency eq. (6), witness consistency eq. (9) are all valid and witness validity eq. (3) is not valid. Query feasibility eq. (10) may or may not be satisfiable, and this info is reported using field "witness_feasible".

"UNKNOWN" otherwise. This can happen when SMLP run terminates before one of the above results can be concluded.

9.3 Mode query: querying for a stable witness

The task of querying ML model for a stable witness to query $\mathsf{query}(p, x, y)$ consists in finding value assignments p^*, x^* for knobs p and inputs x that represent a solution for eq. (12):

$$\exists p, x \ \left[\eta(p) \wedge \forall p' \ \forall y \ \left[\theta(p, p') \implies (\varphi_M(p', x, y) \implies \varphi_{cond}(p', x, y))\right]\right] \tag{12}$$

where

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

According to Definition 1, for any solution p^*, x^* of eq. (12), x^* is a stable witness for $\mathsf{query}(p, x, y)$, for configuration p^*.

Example 3 *(Running Example 2, continued) Consider again the model given by function $f(p,x)$ in eq. (7), Figure 8. For any $p^* \leq -r$, any value $-1 \leq x^* \leq 0$ is a θ_r-stable witness to* query $= y \leq 0$ *for p^*. Hence querying the model $f(p,x)$ will be successful in SMLP for any stability radius $0 \leq r \leq 2$ (recall that the domain of p is $[-2,2]$) and any of the above pairs of values of p^*, x^*, and only those, can be returned by SMLP as a solution to querying the model $f(p,x)$ for condition* query.

As already stated in Section 9.1, the interface consistency eq. (5) and model consistency eq. (6) checks are performed before starting actual querying for a stable witnesses. This ensures that querying is well defined, and cannot fail vacuously in case one of the above checks fail.

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

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

If such p^*, x^* exist, SMLP reports this in the results file using field "query_feasible", by setting it to "true"; otherwise the query fails. If such p^*, x^* exist, SMLP checks whether the following formula is valid (by checking its negation for satisfiability):

$$\eta(p^*) \wedge \forall p' \ \forall y \ [\theta(p^*, p') \implies (\varphi_M(p', x^*, y) \implies \varphi_{cond}(p', x^*, y))] \tag{14}$$

If the above formula is valid, then we have shown that x^* is a stable witness for 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).

An example command to run SMLP in query mode is given in Figure 12.

```
../../src/run_smlp.py -data "../data/smlp_toy_basic" -out_dir ./ -pref Test119 \
-mode query -model system -resp y1,y2 -feat p1,p2 -save_model f -use_model f \
-mrmr_pred 0 -model_per_response t -plots f -seed 10 -log_time f \
-spec ../specs/smlp_toy_system_stable_constant_query.spec
```

Figure 12: Example SMLP command in mode query.

Query results are reported in file prefix_dataname_query_results.json. SMLP supports querying multiple conditions in one SMLP run. Figure 13 displays an example results file in query mode, where the "query_result" field specifies the values of knobs, as well as values of inputs and values of outputs found in the satisfying assignment to eq. (10) that identified the stable witness.

9.4 Mode verify: assertion verification with stability

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 assert(p,x,y) for configuration p^* under allowed perturbations p' of knob values p^* controlled by $\theta(p^*, p')$, is exactly the problem of checking stability of configuration p^* for assertion assert(p,x,y), as defined in Definition 1. To re-iterate, the problem of *verification with stability* is formalized in SMLP as validity of eq. (4).

```
{
        "smlp_execution": "completed",
        "interface_consistent": "true",
        "query_feasible_unstable": {
                "query_feasible": "true",
                "query_stable": "false",
                "query_status": "FAIL",
                "query_result": null
        },
        "query_feasible_stable": {
                "query_feasible": "true",
                "query_stable": "true",
                "query_status": "PASS",
                "query_result": {
                        "p1": 0.0,
                        "y2": 0.0,
                        "p2": 0.0,
                        "y1": 0.0
                }
        },
        "query_infeasible": {
                "query_feasible": "false",
                "query_stable": "false",
                "query_status": "FAIL",
                "query_result": null
        },
        "model_consistent": "true"
}
```


Figure 13: SMLP result in mode query.

```
"configurations": {
        "stable_config": {
                "p1": 7.0,
                "p2": 6.000000067055225
        },
        "grid_conflict": {
                "p1": 3.0,
                "p2": 6.000000067055225
        },
        "unstable_config": {
                "p1": 7.0,
                "p2": 6.0
        },
        "not_feasible": {
                "p1": 7.0,
                "p2": 6.0
        }
 }
```

Figure 14: SMLP specification example to specify "configurations" per assertion, in mode verify.

Example 4 *(Running Example 2, continued) For the model given by function* $y = f(p, x)$ *in eq.* (7)*, Figure 8, let us define* assert $= y \leq 0$*. Then for any legal vale* p^* *of* p*,* assert *fails for positive values of* x *(even if the stability radius* $r = 0$*). However, if the range of* x *is restricted* $-1 \leq x \leq 0$*, then for any knob configuration* $p^* \leq 0$*,* assert *passes verification with stability for any radius* $r \leq |p^*|$*. Note that for configuration* $p^* = 0$*, while* assert *is valid without stability requirements for the restricted range of* x*,* assert *fails verification with stability for any radius* $r > 0$ *(because* $y = f(p, x) > 0$ *for positive values of* p*).*

SMLP supports verification of multiple assertions in a single run, and each assertion is verified with respect to its corresponding configuration (different assertions might refer to different configurations). The general (and the recommended) way of defining a configuration per assertion is using the "configuration" field in SMLP spec file. The "configuration" field is a dictionary with assertion names as keys and the respective values are dictionaries assigning a concrete value to each knob. An example is displayed in Figure 14, where say "stable_configuration" is the name of an assertion and "p1" and "p2" are names of knobs.

When each assertion is verified with respect to the same configuration, and configurations per assertions are not defined using the "configurations" field, SMLP applies a sanity check to see whether unique values p^* to knobs p can be inferred from knob ranges and grids specified in the spec file using the "variables" field. When values p^* cannot be inferred this way, SMLP aborts verification with an error message clarifying the reason.

Below $\vartheta_{p^*}(p)$ denotes the formula inferred form value assignments to p in the configuration for the corresponding assertion when the latter is specified in the spec file using field "configurations", and is constant true otherwise. For example, if p is a variable vector $p1, p2$, then $\vartheta_{p^*}(p)$ might look like $\vartheta_{p^*}(p1, p2) = (p1 = 3 \wedge p2 = 5)$.

As discussed in Section 9.1, the interface consistency eq. (5) and model consistency eq. (6) checks are performed before starting actual assertion verification. In addition to these, for the verification problem eq. (4) to be well defined, 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, *configuration consistency*

check for verification requires, as a necessary condition (but not a sufficient condition), the following formula to be valid:

$$\exists x\ [\vartheta_{p^*}(p) \wedge \alpha(p,x) \wedge \eta(p)] \tag{15}$$

In the implementation, we are using a stronger version of the configuration interface consistency check, called *configuration consistency* check, which in addition takes into account the ML model constraints when checking consistency of the witness; this check subsumes satisfiability check for eq. (15):

$$\exists x\ [\vartheta_{p^*}(p) \wedge \alpha(p,x) \wedge \eta(p) \wedge \varphi_M(p,x,y)] \tag{16}$$

Before performing verification according to eq. (4), SMLP checks satisfiability of the following formula, which we refer to as *assertion feasibility* part of verification with stability:

$$\exists x\ [\vartheta_{p^*}(p) \wedge \alpha(p,x) \wedge \eta(p) \wedge \varphi_M(p,x,y) \wedge \mathsf{assert}(p,x,y)] \tag{17}$$

If the above formula is not satisfiable, then the negated assertion will be true for any legal inputs, which means that the assertion fails everywhere in the legal input space. This is useful info because in such a case it can be that the components of the problem instance (for example, the assertion or the constraints) were not specified correctly.

Next, stability of the configuration p^* for $\mathsf{assert}(p,x,y)$ is checked by proving validity of formula eq. (4), and this is done by checking satisfiability of formula eq. (18), which is the negation of eq. (4):

$$\vartheta_{p^*}(p) \wedge \theta(p,p') \wedge \varphi_M(p',x,y) \wedge \alpha(p',x) \wedge \neg\, \mathsf{assert}(p',x,y)) \tag{18}$$

An example command for mode verify is given in Figure 15

```
../../src/run_smlp.py -data ../data/smlp_toy_ctg_num_resp -out_dir ./ \
-pref Test129 -mode verify -resp y1,y2 -feat x,p1,p2 -model poly_sklearn \
-save_model f -use_model f -model_per_response f -asrt_names \
assert_stable_config,assert_grid_conflict,assert_unstable_config,assert_infeasible \
-asrt_exprs "y2<=90;y1>=9;y1>=(-10);y1>20" -plots f -seed 10 -log_time f  \
-spec ../specs/smlp_toy_configuration_verify.spec
```

Figure 15: Example SMLP command in mode verify.

Verification results are reported in file prefix_dataname_verify_results.json. Figure 16 displays an example results file of SMLP in verify mode.

1. Fields "smlp_execution", "interface_consistent" and "model_consistent" are common for each assertion, and they provide status of the entire execution of SMLP.
2. The field "configuration_consistent" specifies result of eq. (16)
3. The field "assertion_feasible" specifies result of eq. (17)
4. The field "assertion_status" specifies the assertion verification status, and can be one of:

"ERROR" when interface consistency eq. (5) or model consistency eq. (6) or configuration consistency eq. (16) s not valid.

"PASS" when interface consistency eq. (5), model consistency eq. (6), configuration consistency eq. (16) and assertion validity eq. (4) are all valid.

"FAIL" when interface consistency eq. (5), model consistency eq. (6), configuration consistency eq. (16) are all valid and assertion validity eq. (4) is not valid. Assertion feasibility eq. (17) may or may not be satisfiable, and this is reported using field "assertion_feasible".

"UNKNOWN" otherwise. This can happen when SMLP run terminates before one of the above results can be concluded.

9.5 Mode synthesize: parameter synthesis with stability

The task of θ-stable synthesis consists of finding a solution to formula eq. (2), where

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

and $\mathsf{assert}(p,x,y)$ might represent a conjunction of multiple assertions. According to Definition 1, any solution p^* to the synthesis problem, p^* is a stable witness for $\varphi_{post}(p,x,y) = \beta(p,x,y) \wedge \mathsf{assert}(p,x,y)$; the latter expresses the constraints that synthesized design should satisfy (in legal input space).

Example 5 *(Running Example 2, continued) For the model given by function $y = f(p,x)$ in eq. (7), Figure 8, let us define $\beta = y \leq 0$ and* assert = true. *Then, for any legal value p^* of p, β evaluates to* false *for positive values of x; therefore synthesis that guarantees validity of β is not feasible (even for the stability radius $r = 0$). However, if the range of x is restricted to $-1 \leq x \leq 0$, then for any knob configuration $p^* \leq 0$, β is valid (for any values of x in its restricted range) for any stability radius $r \leq |p^*|$, hence stable synthesis is feasible. Note that for configuration $p^* = 0$, β is valid for stability radius $r = 0$, therefore the usual synthesis procedure (that does not take stability requirements into account) might synthesize the model $f(p,x)$ into configuration $f(0,x)$, which will not be robust against perturbations or inaccuracies in measurements or modeling, thus will not be reliable for exploring the system modeled by $f(p,x)$.*

Just like for any other mode of exploration, the interface consistency eq. (5) and model consistency eq. (6) checks are performed before starting actual synthesis procedure. This ensures that synthesis task is well defined, and cannot fail vacuously in case one of the above checks fail.

An example command for mode synthesize is given in Figure 17. In this command, as the ML model we actually use a python expression specified in the spec file using field "system". This is an initial support for specifying systems in SMLP and will be developed and changed in the future, therefore we do not provide any further details here. Please note that support of python expressions as ML models allows one to perform exploration of python expressions with SMLP, in all model exploration modes (e.g., querying, verification, optimization) by taking stability requirements into consideration.

Synthesis result is reported in file prefix_dataname_synthesize_results.json. Figure 18 displays an example results file in synthesize mode. The field "configuration_feasible" reports the validity of

$$\varphi_{feasible} = \exists p,x \left[\eta(p) \wedge \left(\forall y \left[(\varphi_M(p,x,y) \implies \varphi_{cond}(p,x,y))\right]\right)\right] \tag{19}$$

The field "synthesis_result" reports the synthesis results (the synthesized configuration of knob values). The field "configuration_stable" reports that a stable configuration satisfying the synthesis requirements have been found, which is the same as reporting "synthesis_status" as PASS.

```
{
        "assert_stable_config": {
                "configuration_consistent": "true",
                "assertion_status": "PASS",
                "counter_example": null,
                "assertion_feasible": true
        },
        "assert_grid_conflict": {
                "configuration_consistent": "false",
                "assertion_status": "ERROR",
                "counter_example": null,
                "assertion_feasible": "false"
        },
        "assert_unstable_config": {
                "configuration_consistent": "true",
                "assertion_status": "FAIL",
                "counter_example": {
                        "y2": 55.69463654220261,
                        "y1": -58.38640996591811,
                        "p2": 6.125,
                        "p1": 7.5,
                        "x": 1.0
                },
                "assertion_feasible": true
        },
        "assert_infeasible": {
                "configuration_consistent": "true",
                "assertion_status": "FAIL",
                "counter_example": {
                        "y2": 55.69463654220261,
                        "y1": -58.38640996591811,
                        "p2": 6.125,
                        "p1": 7.5,
                        "x": 1.0
                },
                "assertion_feasible": false
        },
        "smlp_execution": "completed",
        "interface_consistent": "true",
        "model_consistent": "true"
}
```


Figure 16: SMLP result in mode verify.

9.6 Mode optimize: multi-objective optimization with stability

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 θ-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 θ-region of knobs, for any values of inputs in their respective legal ranges.

We explain these extensions using two plots in Figure 19. 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.

An example of how to run SMLP in mode optimize was given in Section 3, 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.

Just like for other model exploration modes, the interface consistency eq. (5) and model consistency eq. (6) checks are performed before starting actual optimization procedure. If these checks are successful, SMLP optimization algorithm performs feasibility check that β constraints are feasible under the interface constraints α and η, and if a solution is found, the input and knob values in the satisfying assignment demonstrating the feasible configuration, along with the values of the responses and the objectives, 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 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 9.7, and we refer the reader to it for a formal treatment of stable optimization problem. More precisely, the problem of stable optimization is defined using Equation (22) and the problem of stable optimized synthesis problem is defined using Equation (23),

```
../../src/run_smlp.py -data ../data/smlp_toy_basic -out_dir ./ -pref Test121 \
-mode synthesize -model system -resp y1,y2 -feat p1,p2 -save_model f \
-use_model f -mrmr_pred 0 -model_per_response t -plots f -seed 10 -log_time f \
-spec ../specs/smlp_toy_system_stable_constant_synth_feasible.spec
```

Figure 17: Example SMLP command in mode synthesize.

```
{
        "smlp_execution": "completed",
        "interface_consistent": "true",
        "model_consistent": "true",
        "configuration_feasible": "true",
        "configuration_stable": "true",
        "synthesis_status": "PASS",
        "synthesis_result": {
                "p1": 0.0,
                "p2": 0.0
        }
}
```


Figure 18: SMLP result in mode synthesize.

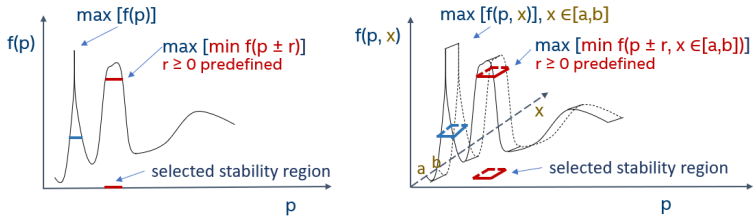

Figure 19: SMLP max-min optimization. On both plots, p denote the knobs. On the right plot we also consider inputs x (which are universally quantified) as part of f.

from which Equation (22) is obtained as a special case, by assuming that assert(p, x, y) is constant true.

An example command for mode optimize is given in Figure 20.

```
../../src/run_smlp.py -data "../data/smlp_toy_basic" -out_dir ./ -pref Test123 \
-mode optimize -pareto t -model system -resp y1,y2 -feat p1,p2 -save_model f \
-use_model f -mrmr_pred 0 -model_per_response t -epsilon 0.00000001 -plots f -seed 10 \
-log_time f -spec ../specs/smlp_toy_system_stable_constant_synth_feasible.spec
```

Figure 20: Example SMLP command in mode optimize.

Figure 21 displays an example results file in optimize mode.

9.7 Mode optsyn: optimized synthesis with stability

Let us first consider optimization without stability or inputs, i.e., far low corner in the exploration cube Figure 2. Given a formula φ_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 (20).

$$[[\varphi_M]]_o = \max_p \{z \mid \forall y \ (\varphi_M(p, y) \implies o(p, y) \geq z)\} \tag{20}$$

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

```
{
        "objv1": {
                "value_in_config": 0.0,
                "threshold_scaled": -0.022943015285783935,
                "threshold": 0.0,
                "max_in_data": 10.7007,
                "min_in_data": 0.24
        },
        "objv2": {
                "value_in_config": 0.0,
                "threshold_scaled": -0.010615194948015235,
                "threshold": 0.0,
                "max_in_data": 102.36396627,
                "min_in_data": 1.0752000000000002
        },
        "y2": {
                "value_in_config": 0.0,
                "value_in_system": 0
        },
        "p1": {
                "value_in_config": 0.0
        },
        "y1": {
                "value_in_config": 0.0,
                "value_in_system": 0
        },
        "p2": {
                "value_in_config": 0.0
        },
        "objv2_scaled": {
                "value_in_config": -0.010615194948015235
        },
        "threshold_lo_scaled": {
                "value_in_config": -0.010615194948015235
        },
        "threshold_lo": {
                "value_in_config": -0.010615194948015235
        },
        "threshold_up_scaled": {
                "value_in_config": -0.006959520828193561
        },
        "threshold_up": {
                "value_in_config": -0.006959520828193561
        },
        "max_in_data": {
                "value_in_config": 1.0
        },
        "min_in_data": {
                "value_in_config": 0.0
        },
        "smlp_execution": "completed",
        "interface_consistent": "true",
        "model_consistent": "true",
        "synthesis_feasible": "true"
}
```


Figure 21: SMLP result in mode optimize.

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* ε, or *ε-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 (21), expressing maximization of a lower bound on the objective function o over parameter values under stable synthesis constraints.

$$[[\varphi_M]]_{o,\theta} = \max_p\{z \mid \eta(p) \wedge \forall p' \; \forall xy \; [\theta(p, p') \implies (\varphi_M(p', x, y) \implies \varphi^{\geq}_{cond}(p', x, y, z))]\} \quad (21)$$

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 η, α, β constraints together with the stability constraints θ. Equivalently, stable optimized synthesis can be stated as the *max-min* optimization problem, Formula (22)

$$[[\varphi_M]]_{o,\theta} = \max_p \min_{x,p'}\{z \mid \eta(p) \wedge \forall y \; [\theta(p, p') \implies (\varphi_M(p', x, y) \implies \varphi^{\leq}_{cond}(p', x, y, z))]\} \quad (22)$$

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 (22) the minimization predicate in the stability region corresponds to the universally quantified p' ranging over this region in (21). 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 (22) 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 (22), where $\mathsf{assert}(p', x, y)$ are assertions required to be valid in the entire stability region around the selected configuration of knobs p:

$$[[\varphi_M]]_{o,\theta} = \max_p \min_{x,p'}\{z \mid \eta(p) \wedge \forall y \; [\theta(p, p') \implies (\varphi_M(p', x, y) \implies \varphi^{\leq}_{cond}(p', x, y, z))]\} \quad (23)$$

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 ε-solutions for these problems carries over from the one given above for Formula (20).

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, discussed in Section 3, 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}.

An example command for mode optsyn is given in Figure 22

```
../../src/run_smlp.py -data "../data/smlp_toy_basic" -out_dir ./ -pref Test125 \
-mode optsyn -pareto t -model system -resp y1,y2 -feat p1,p2 -save_model f \
-use_model f -mrmr_pred 0 -model_per_response t -epsilon 0.00000001 -plots f -seed 10 \
-log_time f -spec ../specs/smlp_toy_system_stable_constant_synth_feasible.spec
```

Figure 22: Example SMLP command in mode optsyn.

Figure 23 displays an example results file in optsyn mode.

10 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.

In the context of DOE, *experiments* are lists of (feature, value) (also called (factor, level)) pairs

$$[(\mathsf{feature}_1, \mathsf{value}_1), \ldots, (\mathsf{feature}_n, \mathsf{value}_n)],$$

and they are rows of the matrix of experiments returned by the supported DOE algorithms. SMLP options that are required to invoke any of the supported DOE heuristics are:

- "doe_factor_level_ranges" A dictionary of levels per feature for building experiments for all supported DOE algorithms. The features are integer features (thus the values are integers). The keys in that dictionary are names of features and the associated values are lists $[\mathsf{val}_1, \ldots, \mathsf{val}_k]$ from which value for that feature are selected to build an experiment. We refer to these lists of values as the *grids* associated to each feature. DOE algorithms that work with two levels only treat these levels as the min and max of the grid range of a numeric variable.

  Example: $\{\text{"Pressure"} : [50, 60, 70], \text{"Temperature"} : [290, 320, 350], \text{"Flowrate"} : [0.9, 1.0]\}$.

```
{
        "objv1": {
                "value_in_config": 0.0,
                "threshold_scaled": -0.022943015285783935,
                "threshold": 0.0,
                "max_in_data": 10.7007,
                "min_in_data": 0.24
        },
        "objv2": {
                "value_in_config": 0.0,
                "threshold_scaled": -0.010615194948015235,
                "threshold": 0.0,
                "max_in_data": 102.36396627,
                "min_in_data": 1.0752000000000002
        },
        "y2": {
                "value_in_config": 0.0,
                "value_in_system": 0
        },
        "p1": {
                "value_in_config": 0.0
        },
        "y1": {
                "value_in_config": 0.0,
                "value_in_system": 0
        },
        "p2": {
                "value_in_config": 0.0
        },
        "objv2_scaled": {
                "value_in_config": -0.010615194948015235
        },
        "threshold_lo_scaled": {
                "value_in_config": -0.010615194948015235
        },
        "threshold_lo": {
                "value_in_config": -0.010615194948015235
        },
        "threshold_up_scaled": {
                "value_in_config": -0.006959520828193561
        },
        "threshold_up": {
                "value_in_config": -0.006959520828193561
        },
        "max_in_data": {
                "value_in_config": 1.0
        },
        "min_in_data": {
                "value_in_config": 0.0
        },
        "smlp_execution": "completed",
        "interface_consistent": "true",
        "model_consistent": "true",
        "synthesis_feasible": "true"
}
```


Figure 23: SMLP result in mode optsyn.

- "doe_algo" Allows to specify the folloving DOE algorithms supported in SMLP:
  - "full_factorial" Builds a full factorial design dataframe from a dictionary of feature value grids, doe_factor_level_ranges. Here the attribute *full* means that all combinations of (feature, value) pairs are used to build experiments, which is not feasible with a large number of features, and with possibly more than two values (levels) in the corresponding grid of values.
  - "fractional_factorial" Builds a 2-level fractional factorial design dataframe from a dictionary doe_factor_level_ranges of feature value grids and given *resolution*. Here the attribute 2-*level* means that every feature is represented with (or ranges over) two values only, and the attribute *factorial* means that, unlike in "full_factorial" design, only a subset of all possible (feature, value) pairs are used to build experiments. A resolution is a way to specify how to group (feature, value) pairs to specify the subset of experiments to build.
  - "plackett_burman" Builds a Plackett-Burman design dataframe from a dictionary of feature value grids. Only min and max values of the range are required.
  - "sukharev_grid" Builds a Sukharev-grid hypercube design dataframe from a dictionary of feature value grids.
  - "box_behnken" Builds a Box-Behnken design dataframe from a dictionary of feature value grids.
  - "box_wilson" Builds a Box-Wilson central-composite design dataframe from a dictionary of feature value grids.
  - "latin_hypercube" Builds simple Latin Hypercube from a dictionary of feature value grids. Latin Hypercube sampling selects required number of sampling points so that no two samples use the same grid value, for any of the features (so selection is history dependent). Latin Hypercube sampling forces the samples drawn to correspond more closely with the input distribution, and it converges faster than the Monte Carlo sampling which also uses random sampling from feature value distributions.
  - "latin_hypercube_sf" Builds a space-filling Latin Hypercube design dataframe from a dictionary of feature value grids. The attribute *space filling* indicates that sampling is done after dividing feature ranges into a predefined number of equal intervals, which are then randomly sampled.
  - "random_k_means" Builds designs with random_k-means_clusters from a dictionary of feature value grids.
  - "maximin_reconstruction" Builds maximin reconstruction matrix from a dictionary of feature value grids.
  - "halton_sequence" Builds Halton matrix based design from a dictionary of feature value grids.
  - "uniform_random_matrix" Builds uniform random design matrix from a dictionary of feature value grids.
- "doe_num_samples" Number of samples (experiments) to generate.

SMLP –help provides detailed information on all DOE options. SMLP uses the pyDOE package `https://pythonhosted.org/pyDOE/` to implement DOE, and any missing details on DOE options and further references can be found there.

An example command for mode doe is given in Figure 24

The generated data will be in file Test34_doe_four_levels_real_doe.csv, the top rows of which are displayed in Figure 25.

11 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. See [BKK20, KRZS11, Wan13, KN21, Kha22] for usage of SD and closely related techniques of *Rule Learning* in validation and test, including for root-causing. 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.

An example command for mode subgroups is given in Figure 28. The generated sungroup descriptions are reported in file with suffix "*_features_ranking.csv".

12 Model refinement loop

Support in SMLP for selecting DOE vectors to sample the system and generate a training set was discussed in Subsection 10. 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

```
../../src/run_smlp.py -doe_spec.csv ../grids/doe_four_levels_real -out_dir ./ \
-pref Test34 -mode doe -doe_algo full_factorial -log_time f
```

Figure 24: Example SMLP command in mode doe.

```
a,b,c
5.2,-1,0.1
7.1,1,1
3.6,0,2
2.3,,4.3
```

```
a,b,c
2.3,-1.0,0.1
3.6,-1.0,0.1
5.2,-1.0,0.1
7.1,-1.0,0.1
2.3,0.0,0.1
3.6,0.0,0.1
5.2,0.0,0.1
7.1,0.0,0.1
2.3,1.0,0.1
```

Figure 25: SMLP doe spec (on the left) and part of generated data (on the right), in mode doe.

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 θ-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 θ-stable witness for query $\mathsf{query}(p^*, x, y) = \neg\,\mathsf{assert}(p^*, x, y)$), then the system is sampled in the θ–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 θ-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 θ-stability region, the higher the chances to reproduce failure of assertion $\mathsf{assert}(p, x, y)$ on the system.

On the other hand, if $\mathsf{assert}(p, x, y)$ does not have a θ-stable counter-example, then it is still possible that 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 θ_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 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.

```
../../src/run_smlp.py -data ../data/smlp_toy_num_resp_mult -out_dir ./ \
-pref Test30 -mode subgroups -psg_dim 3 -psg_top 10 -resp y1,y2 -feat x,p1,p2 \
-plots f -seed 10 -log_time f
```

Figure 26: Example SMLP command in mode subgroups.

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 providing new data-points that can be used for model refinement in this region, through re-training or incremental training of a new model.

13 SMLP NLP Module: Text Preprocessing with SpaCy

SMLP includes an NLP module based on the SpaCy library, implemented via the `SmlpNlp` class. This component provides a configurable, modular text preprocessing pipeline for various natural language processing workflows, including embedding, classification, or retrieval.

13.1 Core Capabilities

The NLP module supports:

- **Lemmatization:** Normalize tokens to their base forms (lemmas).
- **Part-of-speech (POS) tagging:** Annotate each token with grammatical labels.
- **Named Entity Recognition (NER):** Identify entities such as people, locations, and organizations.
- **Sentence segmentation (senter):** Divide text into sentences for further analysis.
- **Token filtering:** Remove stopwords, punctuation, and irrelevant tokens.

13.2 Pipeline Configuration

Users can select individual SpaCy pipeline components using the following options:

- `-nlp_lemmatizer` Enable lemmatization
- `-nlp_tagger` Enable POS tagging
- `-nlp_ner` Enable named entity recognition
- `-nlp_senter` Enable sentence segmentation
- `-nlp_parser` Enable syntactic parsing
- `-nlp_tok2vec` Enable token-to-vector embedding stage
- `-nlp_ruler` Enable attribute ruler for rule-based tagging
- `-nlp_morphologizer` Enable morphological tagging
- `-nlp_spacy_core='en_core_web_sm'` Choose SpaCy language model

Components are dynamically enabled or disabled depending on the task, allowing efficient and task-specific preprocessing.

13.3 Preprocessing Workflow

The `nlp_preprocess()` function executes the following steps:

1. Truncates input to 100,000 characters if needed (SpaCy limitation).
2. Applies the configured pipeline to the input text.
3. Removes tokens that are stop words, punctuation, or of part-of-speech types `SPACE`, `X`, or `PUNCT`.
4. Returns a space-separated string of lemmatized, relevant tokens.

13.4 Integration with SMLP

This NLP module integrates seamlessly with other components, including:

- `SmlpText` for text vectorization and classification
- RAG pipelines for preparing training corpora or filtering candidate passages

Note: The NLP pipeline can be initialized either from a blank SpaCy object or from a pre-trained core model (`en_core_web_sm`, `md`, or `lg`). Components are conditionally loaded or removed for optimal performance.

Best Use Cases: Apply this module prior to text classification (e.g., spam classification or sentiment analysis) and regression tasks, RAG embedding, or any ML application requiring clean, linguistically normalized text input.

14 SMLP Text Module: Feature Engineering from Text

The `SmlpText` module of SMLP enables automatic feature synthesis from raw textual data, supporting multiple vectorization strategies, word embedding methods, and integration with downstream classification or root-cause analysis. It is designed for both supervised and unsupervised workflows and can be tightly coupled with the NLP module (Section 13).

14.1 Purpose and Applications

This module enables users to:

- Preprocess raw textual data using the `SmlpNlp` pipeline
- Embed text as numerical vectors using classical and modern methods
- Integrate vectorized features into ML models or root-cause pipelines
- Support tasks such as text classification, anomaly explanation, and trace/event log analysis

14.2 Text Embedding Methods

The module supports the following embedding types, selected via `-text_embedding`:

- `bow` (Bag-of-Words): Count-based word frequency encoding
- `tfidf`: Term frequency-inverse document frequency encoding
- `word2vec`: Dense embedding from SpaCy models (`en_core_web_md/lg`)
- `glove`: GloVe embeddings from gensim pre-trained models
- `cbow`, `skipgram`: Trained or pre-trained fastText embeddings

14.3 Text Vectorization Pipeline

Given a raw text column, the module:

1. Applies preprocessing via `SmlpNlp`
2. Splits the data into train/test for supervised scenarios
3. Vectorizes text using selected embedding method
4. Returns numerical DataFrames for feature integration

14.4 Supported CLI Parameters

The most important CLI options are:

- `-text_embedding` (Embedding method: `bow`, `tfidf`, `word2vec`, etc.)
- `-text_colname` (Column name of the text field)
- `-ngram_range` (Tuple for n-gram sizes, e.g. `1,3`)
- `-use_wordvec` / `-save_wordvec` (Reuse or persist trained fastText models)
- `-wordvec_model` (Path to pre-trained vector model)

14.5 Integration with SMLP Pipeline

The output of `SmlpText` is a DataFrame of vectorized features. This can be used as:

- Input to the ML model training in classification and regression modes
- Feature space for root-cause analysis using feature selection or subgroup discovery

Use Case Examples: Log message classification, predictive maintenance from failure reports, domain-specific tagging, and embedding event stream text. See Section 15 for more details on text classification, regression, and root cause analysis applications of SMLP text module.

Note: FastText training uses character n-grams controlled by `-fasttext_minn` and `-fasttext_maxn`. GloVe and word2vec embeddings require vector averaging over tokens.

15 Text classification, regerssion, and root cause analysis

Using the text processing capabilities and feature generation capabilities from text discussed in Sections 13 and 14, SMLP supports labeled text data classification and root-cause analysis. Classification task can be that of spam detection, sentiment analysis, or other. While root-cause analysis of text data is performed using the subgroup discovery algorithm discussed in Section 11.

15.1 Text classification and regression

The input data contains text and a label or a numeric value for each entry (each row). SMLP processes the text, generates features, and calls the respective model training and prediction algorithm on the resulting tabular data.

```
../../src/run_smlp.py -data "../data/smlp_toy_basic_text.csv" \
-out_dir ./ -pref Test230 -mode train -resp label -feat text \
-model dt_sklearn -dt_sklearn_max_depth 15 -data_scaler none \
-mrmr_pred 0 -save_model_config f -plots f -pred_plots f \
-resp_plots f -seed 10 -log_time f -pos_val Thor -neg_val Luki \
-text_colname text -text_embedding bow -mrmr_pred 4 -ngram_range 1,2
```

Figure 27: Example SMLP command in mode train for text data.

15.2 Subgroup discovery for text data

The input data contains text data and a label or a numeric value for each entry. SMLP processes the text, generates features, and calls the subgroup discovery algorithm on the resulting tabular data.

```
../../src/run_smlp.py -data "../data/smlp_toy_extracols_text.csv" \
-out_dir ./ -pref Test231 -mode subgroups -psg_dim 3 -psg_top 10 \
-resp label -feat num,text -data_scaler none -save_model_config f \
-plots f -seed 10 -log_time f -pos_val Thor -neg_val Luki \
-text_colname text -text_embedding bow -mrmr_pred 0 -ngram_range 1,2
```

Figure 28: Example SMLP command in mode subgroups for text data.

16 LLM Training from Scratch in SMLP

SMLP provides two alternative mechanisms for training transformer-based language models from scratch. These approaches serve different use cases and levels of control:

16.1 SmlpGenerate: Minimal, Experimental Trainer

The `SmlpGenerate` class is a lightweight wrapper for symbolic model training based on custom Transformer and LanguageModel implementations. It supports:

- Hard-coded Transformer instantiation with fixed architecture
- Manual vocabulary extraction using an NLP preprocessor instance
- Training and generating text via symbolic models

This class is useful for quick experimentation or demonstration of symbolic reasoning concepts but is not intended for general-purpose large-scale model training.

16.2 ScratchTrainer: Full HF-based Training Interface

The `ScratchTrainer` class (in the `smlp_llm.py` module) supports modular, HuggingFace-compatible training from scratch. It uses standard tools including:

- HuggingFace Datasets and Tokenizers
- Byte-Pair Encoding (BPE) tokenizer training from raw text
- Padding, truncation, grouping, and formatting for block-based training
- HuggingFace Trainer with standard logging and checkpointing

This class is fully configurable and can be extended or invoked as a part of the end-to-end SMLP pipeline. The key features include:

- Accepting user-provided or auto-trained tokenizers
- Support for any HF-compatible language model class
- Generation of training and test datasets from raw strings
- Saving and loading trained models and tokenizers

16.3 SmlpScratch Wrapper: Full CLI Integration

To integrate scratch training into the main SMLP workflow, the `SmlpScratch` wrapper class provides:

- CLI parameter dictionary similar to fine-tuning and RAG flows
- `smlp_llm()` method invoked from `smlp_flows.py`
- Support for training and evaluation using HuggingFace Trainer
- Optional text generation after training
- Loading models from memory or disk for reuse

This allows symbolic scratch training to be executed from the command line in the same way as fine-tuning and RAG workflows.

17 Retrieval-Augmented Generation (RAG) in SMLP

SMLP extends its capabilities with Retrieval-Augmented Generation (RAG) for question answering, summarization, and document-based natural language understanding. RAG integrates neural generation with information retrieval, supporting both HuggingFace-based and LangChain-based RAG workflows. These modules allow users to query structured or unstructured corpora (e.g., PDFs, JSON, CSV) and retrieve grounded, contextually relevant responses.

17.1 Supported RAG Backends

- **HuggingFace-based RAG (HF-RAG)**:
  - Built around the `facebook/rag-token-base` model (and compatible variants).
  - Combines a question encoder (typically BART or T5) with a retriever (e.g., FAISS or cosine similarity).
  - Supports full model **fine-tuning** on custom QA-style data. During training, the model learns to generate correct answers from retrieved documents.
  - **Training data format**: Requires at minimum:
    - A set of input `text` documents (used to build the retrieval index).
    - A list of `questions` provided via `-questions` flag.
    - Optionally: a matching list of expected `answers`, for more supervised training.
  - If no explicit `answer` is provided, the retrieved document is used as a pseudo-label for the generated target during training.
  - The training loss is the **sequence generation loss** (token-level cross-entropy) between the predicted output and the expected answer.
  - Outputs are generated via the built-in `generate()` method of the fine-tuned RAG model.
- **LangChain-based RAG (LC-RAG)**:
  - Uses LangChain's retriever-generator-prompt architecture.
  - Compatible with external LLM endpoints such as `Ollama`, `OpenAI`, or custom APIs.
  - Does **not support model retraining**. Instead, LC-RAG focuses on optimizing:
    - Document chunking and embedding strategies.
    - Prompt templates that combine the query and retrieved documents.
    - Retrieval configuration (e.g., top-k, similarity metric).
  - Useful for lightweight deployment and modular experimentation.
  - Outputs are generated dynamically at inference time using external LLM completions.

17.2 RAG Input Formats

SMLP supports the following formats as input sources:

- **PDF**: Single PDF file; parsed using the selected `pdf_mode`.
- **JSON**: List of title-text pairs.
- **CSV**: Must contain columns `title,text`.

17.3 PDF Parsing Modes (Extended)

SMLP supports several modes for converting PDFs into textual passages. Use `-pdf_mode` to select a strategy:

- `langchain`: Default parser. Uses PyPDFLoader or PDFMinerLoader via LangChain. Best for simple PDFs with selectable text.
- `huggingface`: Assumes pre-extracted text; processes token-level chunking.
- `unstructured`: Leverages Unstructured.io for complex layouts, headers, tables, and hierarchical content.
- `ocr`: Uses Tesseract OCR (or similar) to extract text from scanned image-based PDFs.

The choice of PDF mode affects how passages are segmented, the quality of retrieved results, and ultimately the generated answers. For most use cases with standard PDFs, `langchain` suffices. Use `unstructured` for scientific papers or regulatory documents with non-linear structures.

Recommendation: If retrieval fails or appears noisy, inspect passage formatting. Switching to `unstructured` or `huggingface` mode may help extract more coherent and contextually relevant passages.

17.4 Prompt Types (relevant for LC-RAG)

SMLP LC-RAG supports multiple prompt templates, each optimized for different reasoning or response formats. Select via `-prompt_type`. Below are the available options:

- `document_focused` (default): Generates answers grounded strictly in retrieved content. Best for technical and factual documents.
- `chain_of_thought`: Encourages intermediate reasoning steps. Recommended when multi-step logic or justification is desired.
- `extractive`: Aims to extract precise answer spans. Useful when exact phrases or facts from the document are required.
- `strict_qa`: Provides short, direct answers with minimal explanation. Suitable for well-formatted FAQs.
- `summarizer`: Condenses and reformulates the context into a summary related to the query.

Choose prompt types based on the response format expected by downstream users or systems. For instance, `document_focused` is ideal for compliance questions, while `chain_of_thought` supports LLM reasoning in research assistants.

17.5 Common CLI Parameters for HF and LC RAG

- `-rag_base_model_name`: Base model name or path.
- `-rag_trained_model_path`: Path to save or load trained model/artifacts.
- `-rag_text`: Path to input (PDF, JSON, or CSV).
- `-questions`: Comma-separated list of questions.

- `-top_k_passages`: Number of documents to retrieve (default: 3).
- `-index_backend`: One of `faiss` or `cosine`.
- `-max_new_tokens`: Length of generated answers (default: 64).
- `-do_sample`: Enable sampling for generation.
- `-do_train`: Whether to fine-tune (HF) or prepare index (LC).
- `-do_eval`: Whether to evaluate (generate answers).

17.6 HF-RAG Usage

HF-RAG supports both generation and fine-tuning. Fine-tuning uses HuggingFace's `Seq2SeqTrainer` and is more resource-intensive.

Train and Save HF-RAG Model

```
python run_smlp.py -mode rag --rag_type hf \
  --rag_base_model_name facebook/rag-token-base \
  --rag_train t --rag_eval t --do_sample f \
  --rag_text ../texts/toy_smlp.pdf \
  --questions "Is there a published paper on SMLP?" \
  --index_backend faiss \
  --rag_trained_model_path ../raghf_model_dir \
  --output_dir ./ --pref train_hf_rag
```

Figure 29: Example SMLP command in mode rag for text data.

Reuse HF-RAG Model for Generation

```
python run_smlp.py -mode rag --rag_type hf \
  --rag_train f --rag_eval t --do_sample f \
  --rag_base_model_name facebook/rag-token-base \
  --rag_text ../texts/toy_smlp.pdf \
  --questions "Is there a published paper on SMLP?" \
  --rag_trained_model_path ../raghf_model_dir \
  --output_dir ./ --pref infer_hf_rag
```

Figure 30: Example SMLP command in mode rag for text data.

17.7 LC-RAG Usage

LangChain RAG does not retrain models. It prepares retriever and prompt chains using LangChain tools and saves artifacts.

Train and Save LC-RAG Artifacts

Reuse LC-RAG Artifacts for Generation

17.8 Configuration and Reproducibility

SMLP automatically saves configuration files alongside training outputs. These allow consistent re-runs of trained models. Note that complete determinism is not always achieved due to stochastic behaviors in LLM decoding and GPU operations.

17.9 Comparative Notes

- **HF-RAG**: Supports retraining (via `Seq2SeqTrainer`); slower and more memory-intensive; higher control.
- **LC-RAG**: No model retraining; lightweight and fast; depends heavily on retriever/prompt tuning.
- Quality varies per application. HF-RAG may perform better on domain-specific corpora; LC-RAG is modular and agile.

17.10 Advanced Options

SMLP provides fine-grained control over RAG behavior for both HuggingFace-based and LangChain-based pipelines.

HuggingFace RAG (HF-RAG): Key options include:

- `-batch_size`: Per-device batch size during training and evaluation (default: 2).

```
python run_smlp.py -mode rag --rag_type lc \
  --rag_base_model_name deepseek-r1:1.5b \
  --rag_train t --rag_eval t --do_sample f \
  --rag_text ../texts/toy_smlp.pdf \
  --questions "Is there a published paper on SMLP?" \
  --index_backend cosine \
  --rag_trained_model_path ../raglc_model_dir \
  --output_dir ./ --pref train_lc_rag
```

Figure 31: Example SMLP command in mode `rag` for text data.

```
python run_smlp.py -mode rag --rag_type lc \
  --rag_base_model_name deepseek-r1:1.5b \
  --rag_train f --rag_eval t --do_sample f \
  --rag_text ../texts/toy_smlp.pdf \
  --questions "Is there a published paper on SMLP?" \
  --index_backend cosine \
  --rag_trained_model_path ../raglc_model_dir \
  --output_dir ./ --pref infer_lc_rag
```

Figure 32: Example SMLP command in mode `rag` for text data.

- `-epochs`: Number of training epochs (default: 1).
- `-max_input_length`: Maximum input token length for encoder (default: 256).
- `-max_target_length`: Maximum label (answer) length during training (default: 64).
- `-max_new_tokens`: Maximum number of tokens generated at inference time (default: 64).
- `-lr`: Learning rate for optimizer (default: 2e-5).
- `-weight_decay`: Weight decay coefficient for regularization (default: 0.01).
- `-eval_strategy`: Evaluation scheduling (`epoch`, `steps`, `no`).
- `-save_steps`, `-logging_steps`: Save and logging intervals.
- `-report_to`: Reporting backend (`none`, `tensorboard`, `wandb`).
- `-index_backend`: Type of retrieval index (`faiss` currently supported).
- `-trust_remote_code`: Whether to allow loading models with custom remote code (default: True).
- `-device`: Compute device (`cpu` or `gpu`).

LangChain RAG (LC-RAG): Although LC-RAG does not support training, it provides key configuration for retrieval and LLM API:

- `-base_url`: Base URL for Ollama or LLM API (default: `http://localhost:11434`).
- `-embedding_model`: Embedding model used to encode passages (default: `nomic-embed-text`).
- `-retriever_backend`: Retrieval method (`cosine` or `faiss`).
- `-headers_to_split_on`: Markdown headers for document chunking (default: #/##/###).

In both HF and LC modes, parameters like `-do_sample`, `-max_new_tokens`, and `-top_k_passages` directly influence generation quality and passage selection.

Use these options to adapt the pipeline to your hardware constraints, document structure, and application needs.

For full CLI reference, use:

```
python run_smlp.py --help
```

18 Fine-Tuning Language Models in SMLP

SMLP supports fine-tuning large language models (LLMs) on custom datasets for three core tasks: text generation, question answering (QA), and summarization. Fine-tuning enables domain adaptation, performance improvement, or personalization of open-source foundation models. Compared to Retrieval-Augmented Generation (RAG), fine-tuning directly modifies model weights and is suitable when context retrieval is unnecessary or infeasible.

18.1 Supported Fine-Tuning Tasks

- **Text Generation**: Fine-tunes a decoder-only model to produce fluent responses or completions from prompt-like inputs.
- **Question Answering (QA)**:
  - **Generative QA (Seq2Seq)**: Fine-tunes an encoder-decoder model to generate answer from question + context.
  - **Extractive QA (BERT-style)**: Fine-tunes encoder-only model to select answer span from context using start/end logits.
- **Summarization**: Fine-tunes encoder-decoder models to compress a longer passage into a concise abstract or extractive summary.

18.2 Compatible Base Models

Task	Model Type	Examples
Text Generation	Decoder-only (Causal LM)	`TinyLlama`, `LLaMA`, `Falcon`
QA (Generative)	Encoder-Decoder (Seq2Seq)	`google/flan-t5-base`, `t5-small`
QA (Extractive)	Encoder-only	`bert-base-uncased`, `distilbert-base-cased`
Summarization	Encoder-Decoder (Seq2Seq)	`facebook/bart-base`, `t5-small`

18.3 Training Data Format

Fine-tuning datasets must be in JSON Lines (`.json`) format.

18.3.1 Text Generation

Each entry has a single `text` field.

Example `finetune_text_generation.json`**:**

```
{"text": "The capital of France is Paris."}
{"text": "The sun rises in the east."}
{"text": "Large language models are powerful tools."}
```

18.3.2 QA (Generative / Seq2Seq)

Each line includes `input` and `output`.

Example `finetune_qa.json`**:**

```
{"input": "question: What is the capital of France? context: \
France is a country in Europe. Its capital is Paris.",
 "output": "Paris"}

{"input": "question: What is SMLP? context: \
SMLP is an ML model exploration tool.",
 "output": "SMLP is an ML model exploration tool"}
```

18.3.3 QA (Extractive / BERT-style)

Each line contains `question`, `context`, and `answer`.

Example `finetune_qa_with_context.json`**:**

```
{"question": "What is the capital of France?",
 "context": "France is a country in Europe. Its capital is Paris.",
 "answer": "Paris"}

{"question": "What is SMLP?",
 "context": "SMLP is an ML model exploration tool.",
 "answer": "SMLP is an ML model exploration tool"}
```

18.3.4 Summarization

Each line includes `input` and `output` (the summary).

Example `finetune_summarization.json`**:**

```
{"input": "SMLP is a symbolic machine learning framework designed for \
model reasoning and explanation.",
 "output": "SMLP is a symbolic ML framework for explainability."}

{"input": "The earth revolves around the sun once every 365 days, \
which defines the length of a year.",
 "output": "The earth orbits the sun annually."}
```

18.4 Fine-Tuning Commands

```
python run_smlp.py -mode finetune \
  -finetune_task text-generation \
  --finetune_base_model_name TinyLlama/TinyLlama-1.1B-Chat-v1.0 \
  -finetune_dataset ../texts/finetune_text_generation.json \
  --finetune_trained_model_path ./ \
  -finetune_train t -finetune_eval t \
  -finetune_prompt "SMLP is a tool" \
  -finetune_4bit f --finetune_max_seq_length 128 \
  -finetune_epochs 1 -finetune_batch 1 \
  -out_dir ./ -pref Test249
```

Figure 33: Train and evaluate text generation model.

Train and Evaluate Text Generation Model: Figure 18.4

Evaluate Trained Text Generation Model: Figure 18.4

Train and Evaluate Generative QA Model: Figure 18.4

Evaluate Trained Generative QA Model: Figure 18.4

Train and Evaluate Extractive QA Model: Figure 18.4

Evaluate Trained Extractive QA Model: Figure 18.4

Train and Evaluate Summarization Model: Figure 18.4

```
python run_smlp.py -mode finetune \
  -finetune_task text-generation \
  -finetune_base_model_name TinyLlama/TinyLlama-1.1B-Chat-v1.0 \
  -finetune_dataset ../texts/finetune_text_generation.json \
  -finetune_trained_model_path ../finetune_models/test249_model \
  -finetune_train f -finetune_eval t \
  -finetune_prompt "SMLP is a tool" \
  -finetune_4bit f -out_dir ./ -pref Test250
```

Figure 34: Reuse trained model for text generation.

```
python run_smlp.py -mode finetune \
  -finetune_task qa \
  --finetune_base_model_name google/flan-t5-base \
  -finetune_dataset ../texts/finetune_qa.json \
  --finetune_trained_model_path ./ \
  -finetune_train t -finetune_eval t \
  -finetune_prompt "What is SMLP?" \
  -finetune_4bit f --finetune_max_seq_length 128 \
  -finetune_epochs 1 -finetune_batch 1 \
  -out_dir ./ -pref Test251
```

Figure 35: Train and evaluate generative QA model.

```
python run_smlp.py -mode finetune \
  -finetune_task qa \
  -finetune_base_model_name google/flan-t5-base \
  -finetune_dataset ../texts/finetune_qa.json \
  -finetune_trained_model_path ../finetune_models/test251_model \
  -finetune_train f -finetune_eval t \
  -finetune_prompt "What is SMLP?" \
  -finetune_4bit f -out_dir ./ -pref Test252
```

Figure 36: Reuse trained generative QA model.

```
python run_smlp.py -mode finetune \
  -finetune_task qa \
  --finetune_base_model_name bert-base-uncased \
  -finetune_dataset ../texts/finetune_qa_with_context.json \
  --finetune_trained_model_path ./ \
  -finetune_train t -finetune_eval t \
  -finetune_prompt "What is SMLP?" \
  -finetune_context "SMLP is a tool for Machine Learning model \
exploration. The model exploration capabilities include \
configuration optimization, verification of assertions, \
configuration synthesis, querying the ML model, and root-cause analysis." \
  -finetune_4bit f --finetune_max_seq_length 128 \
  -finetune_epochs 1 -finetune_batch 1 \
  -out_dir ./ -pref Test253
```

Figure 37: Train and evaluate extractive QA model.

```
python run_smlp.py -mode finetune \
  -finetune_task qa \
  -finetune_base_model_name bert-base-uncased \
  -finetune_dataset ../texts/finetune_qa.json \
  -finetune_trained_model_path ../finetune_models/test253_model \
  -finetune_train f -finetune_eval t \
  -finetune_prompt "What is SMLP?" \
  -finetune_context "SMLP is a tool for Machine Learning model \
exploration. The model exploration capabilities include \
configuration optimization, verification of assertions, \
configuration synthesis, querying the ML model, and root-cause analysis." \
  -finetune_4bit f -out_dir ./ -pref Test254
```

Figure 38: Reuse trained extractive QA model.

```
python run_smlp.py -mode finetune \
  -finetune_task summarization \
  --finetune_base_model_name google/flan-t5-base \
  -finetune_dataset ../texts/finetune_summarization.json \
  --finetune_trained_model_path ./ \
  -finetune_train t -finetune_eval t \
  -finetune_prompt "SMLP is a symbolic machine learning framework \
designed for model reasoning, model comparison, and symbolic verification. \
It enables efficient workflows for both RAG and finetuning via CLI." \
  -finetune_4bit f --finetune_max_seq_length 128 \
  -finetune_epochs 1 -finetune_batch 1 \
  -out_dir ./ -pref Test255
```

Figure 39: Train and evaluate summarization model.

18.5 Fine-Tuning CLI Parameters

General Parameters

- `-finetune_task`: Task type (`text-generation`, `qa`, `summarization`)
- `-finetune_dataset`, `-finetune_base_model_name`
- `-finetune_trained_model_path`
- `-finetune_train`, `-finetune_eval`

Generation Options

- `-finetune_prompt`, `-finetune_context`
- `-finetune_max_new_tokens`, `-finetune_sample`
- `-finetune_temperature`, `-finetune_top_k`, `-finetune_top_p`
- `-finetune_num_beams`

Training Options

- `-finetune_batch`, `-finetune_epochs`, `-finetune_max_seq_length`
- `-finetune_save_steps`, `-finetune_save_strategy`

LoRA and Quantization

- `-finetune_lora_r`, `-finetune_lora_alpha`, `-finetune_lora_dropout`
- `-finetune_4bit`, `-finetune_fp16`, `-finetune_bf16`

19 SMLP Agent and API

19.1 Overview

The **SMLP Agent** is an intelligent interface that translates natural language requests into structured CLI commands for the SMLP system. It is designed for automation, scripting, and integration with user-facing applications. The agent supports both cloud-based LLMs (e.g., OpenAI) and local LLMs via Ollama.

19.2 Agent Architecture

The agent is composed of modular components:

- **LLMInterpreter**: Uses few-shot prompting (or optionally Retrieval-Augmented Generation) to convert user queries into task plans. Supports OpenAI and Ollama backends.
- **SmlpSpecGenerator**: Converts a high-level task plan into a complete SMLP CLI specification dictionary by merging with default arguments.
- **SmlpExecutor**: Converts a specification dictionary into a CLI argument list and executes the SMLP pipeline (`run_smlp.py`) as a subprocess.

- **SmlpAgent**: Coordinates the entire interpretation, validation, execution, and logging process.

19.3 Execution Flow

1. A natural language query is passed to the agent.
2. The agent generates a task plan using an LLM (via few-shot prompting).
3. It filters and validates the task parameters against the SMLP schema.
4. A specification dictionary is produced and executed via subprocess.

19.4 API Interface

To support programmatic access, a FastAPI-based web server wraps the SMLP Agent. This RESTful API allows remote services (e.g., dashboards, chatbots, automation workflows) to interface with the agent securely.

Key API Endpoints

- `GET /agent/status`: Returns basic health and uptime info. Suitable for monitoring dashboards.
- `GET /agent/logs`: Retrieves the last `n` log events from the agent's internal execution log.
- `POST /agent/text`: Accepts a plain English user query and executes the translated SMLP command.
- `POST /agent/task`: Accepts a pre-structured task specification (as JSON) and executes it directly.
- `POST /agent/spec-preview`: Accepts a natural language query and returns the inferred CLI specification without executing.
- `POST /chat`: Similar to `/agent/text`, but returns a structured chat-compatible response format (e.g., for integration with UIs or messaging platforms).
- `POST /load_prompt`: Dynamically updates the few-shot prompt used by the LLMInterpreter.

19.5 Invoking the Agent and API

To use the SMLP Agent programmatically or through a terminal, follow the steps below.

1. Launching the FastAPI Server

Before sending any requests, ensure that the FastAPI server is running:

```
uvicorn smlp_agent_api:app --reload
```

Purpose: This command starts the SMLP Agent's RESTful API on the local machine at `http://127.0.0.1:8000`. It should be run from the root of the repository where `smlp_agent_api.py` is located.

When to run: Run this once per session (or configure as a background service) to expose SMLP Agent functionality over HTTP.

2. Sending Natural Language Queries via Curl

You can send a plain English task description to the agent using a `curl` command like this:

```
curl -X POST http://127.0.0.1:8000/agent/text \
  -H "Content-Type: application/json" \
  -d '{"query": "run rag on toy pdf and check if SMLP is published"}'
```

Purpose: This sends a POST request to the agent's `/agent/text` endpoint. The query is interpreted using the LLM, converted to a valid SMLP command, and executed. The output is returned in the response.

When to run: Use this command when you want to issue ad hoc queries without needing to write Python code or interact with the UI. It is especially useful for testing and integration scripts.

3. Notes and Tips

- Ensure that dependencies (e.g., Ollama or OpenAI API keys) are set up before launching the agent.
- Logs are saved in `smlp_agent_log.jsonl`, which can be inspected or queried using the `/agent/logs` endpoint.
- You can replace `/agent/text` with other endpoints like `/agent/spec-preview` or `/agent/task` depending on your use case.
- Use `-reload` with Uvicorn during development for auto-reloading on file changes.

4. Alternative Invocation Options

Once the API server is running (via `uvicorn`), you can use the following invocation modes in a second shell or external interface:

(i) Command-line with inline query:

```
curl -X POST http://127.0.0.1:8000/agent/text \
  -H "Content-Type: application/json" \
  -d '{"query": "run rag on toy pdf and answer if a paper on SMLP is published"}'
```

(ii) Command-line using input file: If you have a JSON file containing a query:

```
curl -X POST http://127.0.0.1:8000/agent/text \
  -H "Content-Type: application/json" \
  --data @../regr_smlp/queries/few_shot_query.json
```

(iii) Swagger GUI Interface: Open a browser and visit:

`http://127.0.0.1:8000/docs`

This launches an interactive Swagger UI where you can paste natural language queries into the `/agent/text` endpoint and execute them without using the terminal.

(iv) SMLP Chatbot (via Web UI): An optional interactive chatbot interface is available and described in Section 20.

19.6 Deployment Considerations

The SMLP Agent API is intended to be deployed in secure environments, as it allows execution of arbitrary SMLP tasks via LLM interpretation. Best practices include:

- Running behind an authenticated gateway or VPN.
- Rate-limiting or queuing long-running tasks.
- Logging all queries and responses for auditing.
- Customizing the few-shot prompt to align with use-case expectations.

19.7 Sample Interaction

```
POST /agent/text
{
  "query": "Fine-tune a model on the AG News dataset for 2 epochs"
}
```

Response:

```
{
  "output": "...stdout/stderr of SMLP run..."
}
```

19.8 Extensibility

The agent and API are modular and can be extended with:

- Support for other LLM providers (e.g., Anthropic, local models).
- Advanced RAG-based query interpretation.
- Execution scheduling, timeouts, and job tracking.

20 SMLP Chatbot

An interactive web-based chatbot interface is available for users who prefer natural language input instead of the command line.

Launching the Chatbot

Use the following command to launch the chatbot in a browser:

```
streamlit run smlp_chatbot.py
```

Functionality

- Accepts natural language instructions describing a full SMLP task.
- Communicates with the FastAPI server using the `/chat` endpoint.
- Displays results and maintains a session history.
- Provides a sidebar for uploading or selecting few-shot prompt templates dynamically.

Few-Shot Prompt Support

The chatbot allows:

- Uploading a few-shot prompt as a `.txt` file.
- Specifying a path to a local prompt file.
- Sending this prompt to the API to update the LLM interpreter.

Ensure that the FastAPI server is running before launching the chatbot.

21 SMLP MCP (Model Context Protocol) Support

SMLP provides integration with the *Model Context Protocol* (MCP), enabling external tools and applications to call SMLP functionality in a structured, schema-driven manner. This is achieved via two dedicated Python modules:

- `smlp_mcp_server.py` — Implements the MCP server that exposes selected SMLP commands as *MCP tools*.
- `smlp_mcp_client.py` — Implements an MCP client that connects to the SMLP MCP server and invokes its tools.

21.1 Overview

The MCP server is based on the `fastmcp` framework and wraps SMLP functionality inside MCP *tools*. Each tool has a defined input schema, output schema, and description, which allows automatic discovery and invocation by any MCP-compliant client.

Currently, one tool is provided:

`run_smlp_tool` Runs SMLP with given parameters in MCP-safe mode. Standard output is redirected to an internal buffer to avoid corrupting the MCP protocol stream, while log messages are sent to `stderr` and handled by the MCP client's log handler.

21.2 Transport

At present, the SMLP MCP integration supports only the *local I/O transport channel* (`stdio` transport). This means that the client and server run on the same machine and communicate via standard input and output streams. Network-based MCP transport (e.g., WebSocket) is not yet implemented.

21.3 Running the MCP Server

The MCP server can be started as:

```
$ python smlp_mcp_server.py
```

In typical usage, the server is not launched manually. Instead, it is started as a subprocess by the MCP client using the `StdioTransport`, as described in the next subsection.

21.4 Running the MCP Client

The client connects to the server and invokes the `run_smlp_tool` functionality. Example:

```
$ python smlp_mcp_client.py
```

The client will:

1. Start the MCP server (`smlp_mcp_server.py`) as a subprocess.
2. Query the list of available tools from the server.
3. Call `run_smlp_tool` with user-provided parameters (mode, dataset, features, response, etc.).
4. Print the tool's return value, including captured `stdout` from SMLP.
5. Display any server log messages received via the MCP logging channel.

21.5 Logging and Debugging

The MCP server routes:

- All `logging` module output to `stderr`, which is displayed by the client's log handler.
- All `print()` output to an internal buffer, returned as `stdout_log` in the tool result.

This separation ensures MCP protocol messages are not corrupted and that debugging output can be easily viewed.

21.6 Limitations

- Only `stdio` transport is currently supported.
- Concurrent client connections are not supported in the current implementation.

Future work will include support for network transport protocols, multiple tool definitions, and remote execution.

22 SMLP Multi-Component Task Graph

The `smlp_task_graph.py` module allows chaining multiple SMLP tasks into a graph-style pipeline using LangGraph.

Overview

`smlp_task_graph.py` provides:

- A structured workflow system built on LangGraph.
- Execution of a user-provided natural language query using `SmlpAgent`.
- Evaluation of the resulting output, which can be logged or used for further decision making.

Components

- **State:** Tracks the query, messages, raw result, and evaluation string.
- **Nodes:**
  - `smlp_task`: Executes the user's instruction using SMLP.
  - `evaluate`: Generates a summary of the SMLP execution.
- **Graph:** A LangGraph pipeline with edges from task execution to evaluation.

Invocation Example

```
python smlp_task_graph.py
```

A sample query is embedded for testing. Results are printed with a summary of the SMLP run.

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.

[Kha22] Zurab Khasidashvili. Accelerating system-level debug using rule learning and subgroup discovery techniques. *CoRR*, abs/2207.00622, 2022.

[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.

[KN21] Zurab Khasidashvili and Adam J. Norman. Feature range analysis. *Int. J. Data Sci. Anal.*, 11(3):195–219, 2021.

[KRZS11] Yoav Katz, Michal Rimon, Avi Ziv, and Gai Shaked. Learning microarchitectural behaviors to improve stimuli generation quality. In Leon Stok, Nikil D. Dutt, and Soha Hassoun, editors, *Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10, 2011*, pages 848–853. ACM, 2011.

[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.

[Wan13] Li-C. Wang. Data mining in design and test processes: basic principles and promises. In Cheng-Kok Koh and Cliff C. N. Sze, editors, *International Symposium on Physical Design, ISPD'13, Stateline, NV, USA, March 24-27, 2013*, pages 41–42. ACM, 2013.

[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.