GenForge — A Multi-Population GP Toolkit with SPFP

GenForge User Manual & Tutorial

Multi-Population Genetic Programming for Classification & Regression with Semantic Preserving Feature Partitioning (SPFP)

The GenForge Team

September 22, 2025

Version 0.2.0

GenForge package emblem.

Preface

This manual documents `GenForge`, a Python package for multi-population Genetic Programming that supports classification and regression via `gpclassifier` and `gpregressor`, and semantic preserving feature partitioning via `SPFPPartitioner`. It covers installation, end-to-end examples, configuration and API references, plotting/reporting utilities, and design rationales.

How to cite. If you use `GenForge` or SPFP in academic work, please cite the GenForge repository and the accompanying manuscripts referenced throughout this manual.

Contents

List of Figures

List of Tables

1 Introduction & Quickstart

1.1 What is GenForge?

GenForge is a multi-population Genetic Programming (GP) toolkit for classification and regression. It couples (i) *functional diversity* via multiple co-evolving populations with (ii) *ensemble mixing* across populations, and augments the pipeline with *Semantic Preserving Feature Partitioning* (SPFP) to form feature groups that preserve information about the target. The package exposes three top-level modules:

- gpclassifier — multi-population GP for supervised classification,
- gpregressor — multi-population GP for supervised regression,
- SPFPPartitioner — a feature partitioner producing num_groups index sets (*findex*) to wire features to GP populations.

1.2 High-level architecture

At a high level, users (optionally) run SPFPPartitioner to obtain a partition $\{\mathcal{I}_1, \ldots, \mathcal{I}_P\}$ of input features, map each partition to a GP population, then train a GP model whose ensemble head mixes per-population outputs. Figure 1.1 sketches the flow.

1.3 Installation

The package is available from its GitHub repository and can be installed from source.

Listing 1.1: Clone and install from GitHub.

```
git clone https://github.com/maisamkhorshidi/genforge.git
cd genforge
pip install -e .
```

Python compatibility. `GenForge` targets Python 3.9+. The package has no hard dependency on scikit-learn; light shims for standard scaler and label encoding are provided within `genforge.spfp`.

1.4 First run

1.4.1 Classification: `gpclassifier`

Listing 1.2: Minimal classification example with `gpclassifier`.

```python
from genforge.gpclassifier import gpclassifier
import numpy as np

# Toy splits (N×D arrays and 1D labels)
X_train, y_train = np.load("X_train.npy"), np.load("y_train.npy")
X_val, y_val = np.load("X_val.npy"), np.load("y_val.npy")
X_test, y_test = np.load("X_test.npy"), np.load("y_test.npy")

# Optional: feature partitions (list of lists), one list per population
# findex = [[...], [...], ...]

params = {
    "userdata_name": "demo_clf",
    "userdata_xtrain": X_train, "userdata_ytrain": y_train,
    "userdata_xval": X_val, "userdata_yval": y_val,
    "userdata_xtest": X_test, "userdata_ytest": y_test,
    # "userdata_pop_idx": findex, # (optional) wire features to populations
    "runcontrol_pop_size": 64,
    "runcontrol_generations": 50,
    "runcontrol_useparallel": False,
    "functions_name": [["times","minus","plus","divide"]],
    "gene_max_genes": [5],
    "tree_max_depth": [10],
    "selection_elite_fraction": [0.05],
    "selection_elite_fraction_ensemble": [0.05],
    "selection_p_ensemble": [0.5],
    "runcontrol_plotfitness": True,
    "runcontrol_plotrankbest": True,
```

```
    "runcontrol_plotrankall": False,
    "runcontrol_plotsavefig": True,
}

gp = gpclassifier.initialize(**params)
gp.evolve()
gp.report(ensemble_row=0) # writes HTML report (e.g., ensemble_0.html)
```

1.4.2 Regression: `gpregressor`

Listing 1.3: Minimal regression example with `gpregressor`.

```
from genforge.gpregressor import gpregressor
import numpy as np

X_train, y_train = np.load("Xtr.npy"), np.load("ytr.npy")
X_val, y_val = np.load("Xva.npy"), np.load("yva.npy")
X_test, y_test = np.load("Xte.npy"), np.load("yte.npy")

params = {
    "userdata_name": "demo_reg",
    "userdata_xtrain": X_train, "userdata_ytrain": y_train,
    "userdata_xval": X_val, "userdata_yval": y_val,
    "userdata_xtest": X_test, "userdata_ytest": y_test,
    # "userdata_pop_idx": findex, # (optional)
    "runcontrol_pop_size": 64,
    "runcontrol_generations": 50,
    "runcontrol_useparallel": False,
    "functions_name": [["times","minus","plus","divide"]],
    "gene_max_genes": [5],
    "tree_max_depth": [10],
    "selection_elite_fraction": [0.05],
    "selection_elite_fraction_ensemble": [0.05],
    "selection_p_ensemble": [0.5],
    "runcontrol_plotfitness": True,
    "runcontrol_plotrankbest": True,
    "runcontrol_plotrankall": False,
    "runcontrol_plotsavefig": True,
```

```
}

gp = gpregressor.initialize(**params)
gp.evolve()
gp.report(ensemble_row=0) # writes HTML report (e.g., ensemble_0.html)
```

1.4.3 Feature partitioning: `SPFPPartitioner`

Listing 1.4: Quickstart for SPFP feature partitioning.

```
from genforge.spfp.spfp_partition import SPFPPartitioner
import numpy as np

X, y = np.load("X.npy"), np.load("y.npy")

spfp = SPFPPartitioner(
    n_groups=5, # number of feature groups to construct
    n_bins=100, # discretization bins
    objective="micor", # "mi", "cor", or "micor"
    override=False,
    backward=False,
    perfs=0.10, # minimum ratio of features per group
    remp=0.60, # random removal ratio between groups
    random_state=7,
    verbose=1
)

spfp.fit(X, y) # optional: pass X_test, y_test for diagnostics
groups = spfp.partition() # build partitions
partitions = spfp.get_partitions() # [[fidx...], [fidx...], ...]
```

1.5 Reproducibility notes

Set `random_state` (where available) and/or any internal `SEED`, fix `num_threads`, and pin versions of dependencies. Genetic search is inherently stochastic; for apples-to-apples comparisons, keep all seeds and configuration identical across runs.

1.6 Where to next

The next chapters dive into:

1. Installation details and environment setup,
2. SPFP mathematics and algorithmic details,
3. `gpclassifier` configuration and API,
4. `gpregressor` configuration and API,
5. Plotting and HTML reports,
6. Reproducibility and experiment management.

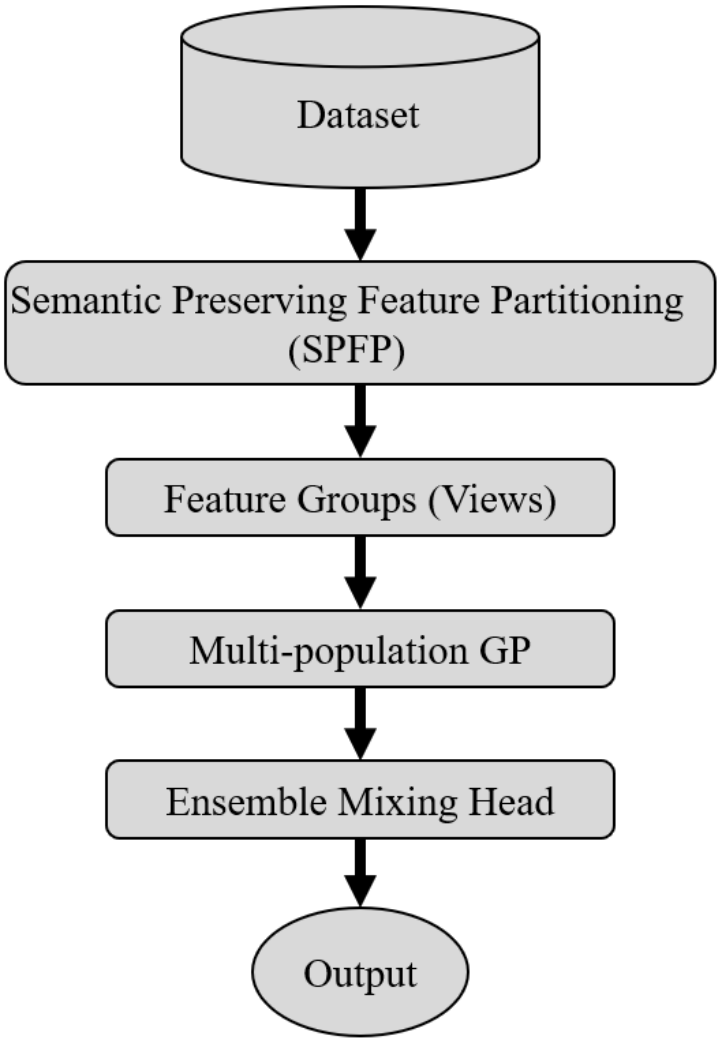


Figure 1.1: GenForge data flow: (1) optional SPFP partitions features into P groups (*findex*); (2) each group maps to a GP population; (3) a linear mixing head combines population outputs into the final prediction.

2 Installation and Environment Setup

This chapter describes how to install GenForge, verify your environment, and prepare your workspace for running gpclassifier, gpregressor, and SPFPPartitioner.

2.1 System Requirements

Table 2.1: Supported environments and core dependencies.

Component	Recommended setup
Operating system	Linux, macOS, or Windows 10/11
Python	3.9–3.12
Core libraries	`numpy`, `scipy`, `joblib`, `matplotlib` (for plots)
Optional	`xgboost` (benchmarks), `pandas` (I/O convenience)
Not required	`scikit-learn` (replaced by internal `StandardScaler`, `LabelEncoder`)

Build tools (source installs). On Linux, ensure `gcc`/`g++` and `make`. On macOS, install Xcode Command Line Tools (`xcode-select -install`). On Windows, install Microsoft C++ Build Tools.

2.2 Install from PyPI

If GenForge is published on PyPI, install with:

Listing 2.1: Install from PyPI.

```
python -m pip install --upgrade pip
pip install genforge
# Optional extras (plots, xgboost benchmarks):
pip install matplotlib xgboost
```

2.3 Install from Source

Clone and install in editable mode:

Listing 2.2: Clone and editable install.

```
git clone https://github.com/<your-org>/genforge.git
cd genforge
python -m pip install --upgrade pip
pip install -e .[dev]
```

Local development layout. A typical tree:

Listing 2.3: Example project layout.

```
your-project/
  data/
    X_train.npy y_train.npy X_val.npy y_val.npy X_test.npy y_test.npy
  runs/ # logs, reports, plots
  scripts/ # experiment scripts
  venv/ # optional virtual environment
```

2.4 Verifying the Installation

Run a quick import/version check:

Listing 2.4: Import and version check.

```
import genforge
print("GenForge version:", getattr(genforge, "__version__", "unknown"))

# Either top-level or submodule imports (depending on your __init__.py):
from genforge import gpclassifier, gpregressor, SPFPPartitioner
# or:
# from genforge.gpclassifier import gpclassifier
# from genforge.gpregressor import gpregressor
# from genforge.spfp import SPFPPartitioner

print("Imports OK.")
```

If imports succeed and the version prints, your environment is ready.

2.5 Parallelism and Reproducibility

CPU parallelism. Fitness evaluation in GenForge can run in parallel. Typical flags include `runcontrol_useparallel` and `runcontrol_n_jobs`. Set `runcontrol_batch_size` to balance work per worker.

Determinism. For reproducible results, set all available seeds: `random_state` in SPFPPartitioner, and any GP-level configuration seeds (e.g., for population initialization). Pin dependency versions in a `requirements.txt` or `pyproject.toml` for consistent environments.

2.6 Data Preparation

Feature scaling. Use the internal `StandardScaler` (genforge.spfp) to avoid external dependencies. For label encoding, use the internal `LabelEncoder`.

Splits. Provide NumPy arrays: `X_train` (N by D), `y_train` (N,), and analogous validation/test splits. If you plan to use SPFPPartitioner, run it on the same training split that your GP will consume.

2.7 Assets to Include for This Chapter

Remark 2.1 (Asset checklist)**.** Place the following under `Figure/` and tell us the filenames to reference them precisely.

1. Environment layout screenshot or diagram (e.g., `fig_install_layout.png`); we will reference it as Figure 2.1.
2. A short terminal capture of successful `pip install genforge` (optional; include as a figure or listing).

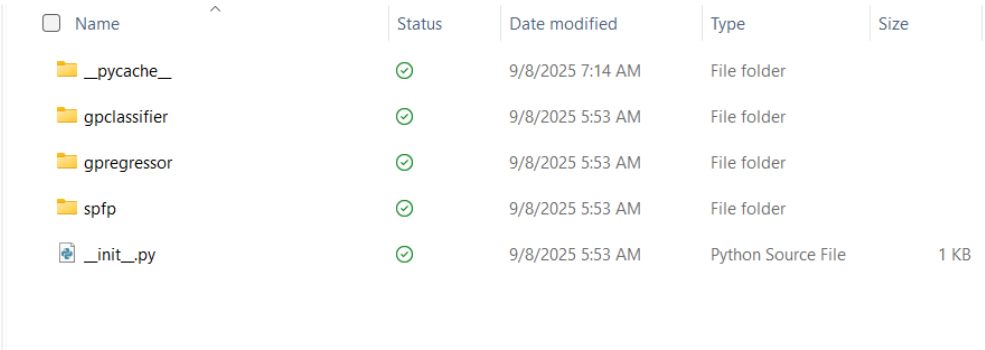


Figure 2.1: Example local project layout for `GenForge` experiments.

3 GPClassifier & GPRegressor Configurations

3.1 Overview

3.2 Overview

gpclassifier and gpregressor configured through a typed container `ClassifierConfig` and `RegressorConfig`. At initialization the configuration is *validated and resolved*: defaults are applied, scalar values are broadcast to all populations where allowed, the function set is imported, and shapes/ranges are checked. The random seed in `runcontrol.random_state` is normalised into a NumPy `Generator` stored as `runcontrol.RNG`. Upon resolution during `gpclassifier(cfg)` and `gpregressor(cfg)` (equivalently, `gp.initialize(cfg)` when an initializer is exposed), the fully checked and expanded parameters are written to the runtime dictionaries `gp.config` and `gp.userdata` for the engine to consume; see Listing 3.1 and Listing 3.2 for a minimal import and initialization example.

The `ClassifierConfig` dataclass is organised into ten branches of parameters that are validated and then propagated to `gp.config` / `gp.userdata`: `runcontrol`, `user`, `selection`, `nodes`, `operator`, `gene`, `tree`, `fitness`, `softmax`, and `log`. Also, the `ClassifierConfig` dataclass is organised into ten branches of parameters that are validated and then propagated to `gp.config` / `gp.userdata`: `runcontrol`, `user`, `selection`, `nodes`, `operator`, `gene`, `tree`, `fitness`, `linregression`, and `log`. Note that many parameters and branches are common across `ClassifierConfig` and `RegressorConfig`. In this chapter, each branch is documented in turn, with the post-resolution (effective) types and semantics enforced by the resolver. This makes the behaviour of downstream components—population initialisation, genetic operators, heads, plotting and reporting—explicit and reproducible for any given configuration.

Unless noted otherwise, many parameters accept either a scalar or a boolean that is applied uniformly to every population, or a list whose length equals `runcontrol.num_pop`. During resolution, scalars and booleans are broadcast and inputs are coerced to the accepted forms listed below (for example, numeric

literals are cast to the target dtype, sequences are validated for length and range, and strings that name functions or optimisers are normalised against the supported sets). This broadcasting and coercion are part of the same resolution step that writes the final values into `gp.config` and `gp.userdata`, so what you see in those dictionaries is exactly what the training/evaluation code will use.

Unless noted otherwise, the parameters are common among `ClassifierConfig` and `RegressorrConfig`.

Listing 3.1: Minimal usage of Importing `gpclassifier` and `ClassifierConfig` for classification task.

```
from genforge.gpclassifier import gpclassifier
from genforge.gpclassifier import ClassifierConfig

cfg = ClassifierConfig()

# Required user data (NumPy arrays)
cfg.user.xtrain = X_train # shape (N, D)
cfg.user.ytrain = y_train # shape (N,)
cfg.user.xval = X_val # optional, or None
cfg.user.yval = y_val # optional, or None
cfg.user.xtest = X_test # optional, or None
cfg.user.ytest = y_test # optional, or None

# Minimal run control
cfg.runcontrol.pop_size = 64
cfg.runcontrol.generations = 50
cfg.runcontrol.random_state = 7 # int | None | np.random.Generator | np.random.RandomState

gp = gpclassifier.initialize(cfg) # resolves and validates, and initializes
gp.evolve()
gp.report(ensemble_row=0) # writes an HTML report
```

3.3 Type conventions used in this chapter

To keep the notation compact:

- `int`, `float`, `bool`, `str` are Python types.
- `list[int]` means a Python list of `int`.

- "scalar or list[·]" means a single value broadcast to all populations or a per-population list whose length equals `runcontrol.num_pop`.
- Arrays are NumPy `ndarray`.

Listing 3.2: Minimal usage of Importing `gpregressor` and `RegressorConfig` for classification task.

```
from genforge.gpregressor import gpregressor
from genforge.gpregressor import RegressorConfig

cfg = RegressorConfig()

# Required user data (NumPy arrays)
cfg.user.xtrain = X_train # shape (N, D)
cfg.user.ytrain = y_train # shape (N,)
cfg.user.xval = X_val # optional, or None
cfg.user.yval = y_val # optional, or None
cfg.user.xtest = X_test # optional, or None
cfg.user.ytest = y_test # optional, or None

# Minimal run control
cfg.runcontrol.pop_size = 64
cfg.runcontrol.generations = 50
cfg.runcontrol.random_state = 7 # int | None | np.random.Generator | np.random.RandomState

gp = gpregressor.initialize(cfg) # resolves and validates, and initializes
gp.evolve()
gp.report(ensemble_row=0) # writes an HTML report
```

3.4 Type conventions used in this chapter

To keep the notation compact:

- `int`, `float`, `bool`, `str` are Python types.
- `list[int]` means a Python list of `int`.
- "scalar or list[·]" means a single value broadcast to all populations or a per-population list whose length equals `runcontrol.num_pop`.

- Arrays are NumPy `ndarray`.

3.5 User Data

`user.name` (`str`). (Optional, default: *Example_Run*) A run label used in logs and reports.

`user.xtrain` (`ndarray` 2D). (Required) Training features of shape `(N, D)`. Enforced to be exactly 2D.

`user.ytrain` (`ndarray` 1D or 2D single column). (Required)Training labels. Internally flattened to 1D; labels are remapped to `0..K-1` if needed (a warning is issued).

`user.xval`, `user.yval` (`ndarray` or `None`). (Optional) Validation split. Shapes are checked; if present, the number of columns must match `user.xtrain`.

`user.xtest`, `user.ytest` (`ndarray` or `None`). (Optional) Test split with the same constraints.

`user.pop_idx` (`list[int]` or `list[list[int]]` or `None`). (Optional, default: [0,..., len(`user.xtrain`)]) Feature indices per population. If multiple populations are used, this should be `list[list[int]]` with valid column indices per group.

`user.initial_population` (`list` or `None`). (Optional, default: None) Initial population per population id; when provided the outer list must have length `runcontrol.num_pop`.

`user.stats` (`bool`). (Optional, default: True) Collect additional runtime diagnostics for reporting.

3.6 Run Control

`runcontrol.num_pop` (`int`). (Optional, default: `None`) Number of GP populations. If `user.pop_idx` is a list of groups, it resolves to its length.

`runcontrol.pop_size` (`int`). (Optional, default: `25`) Individuals per population.

`runcontrol.generations` (`int`). (Optional, default: `150`) Maximum generations to evolve.

`runcontrol.batch_job` (`int`). (Optional, default: `5`) Batch size per worker in parallel mode.

`runcontrol.stallgen (int).` (Optional, default: `20`) Early-stopping window (no improvement) used for diagnostics.

`runcontrol.verbose (int).` (Optional, default: `1`) Console print frequency / verbosity.

`runcontrol.quiet (bool).` (Optional, default: `False`) Quiet logging (no console output).

`runcontrol.useparallel (bool).` (Optional, default: `False`) Enable parallel fitness evaluation.

`runcontrol.n_jobs (int).` (Optional, default: `1`) Worker processes if parallel is enabled.

`runcontrol.usecache (bool).` (Optional, default: `True`) Use fitness cache when cloning individuals.

`runcontrol.minimisation (bool).` (Optional, default: `True`) Minimise fitness (else maximise).

`runcontrol.tolfit (float).` (Optional, default: `1e-9`) Improvement tolerance for convergence checks.

`runcontrol.plotfitness (bool).` (Optional, default: `False`) Toggle Fitness vs. Generation plot.

`runcontrol.plotrankall (bool).` (Optional, default: `False`) Toggle RankAll plots.

`runcontrol.plotrankbest (bool).` (Optional, default: `False`) Toggle RankBest plots.

`runcontrol.plotformat (str or list[str]).` (Optional, default: `"png"`) Image format(s) to save.

`runcontrol.plotfolder (str or None).` (Optional, default: `None`) Plot output folder; if `None` resolves to CWD.

`runcontrol.plotsavefig (bool).` (Optional, default: `True`) Save figures to disk.

`runcontrol.plotlive (bool).` (Optional, default: `False`) Live drawing during the run (if supported).

`runcontrol.plotbackend (str).` (Optional, default: `"auto"`) Matplotlib backend: e.g., `"Agg"`, `"QtAgg"`, `"TkAgg"`, `"MacOSX"`.

`runcontrol.track_individuals (bool).` (Optional, default: `False`) Track individuals across generations (memory-intensive).

runcontrol.resultfolder (str or None). (Optional, default: None) Output folder for run artifacts; if None resolves to CWD.

run.adaptgen (bool). (Specific to RegressorConfig) (Optional, default: False) Activates the adaptive hierarchical symbolic abstraction module.

runcontrol.adaptgen (int). (Specific to RegressorConfig) (Optional, default: 15) The number of generations at which, after no fitness improvement, the adaptive hierarchical symbolic abstraction is performed.

Randomness and reproducibility.

runcontrol.random_state (None or int or np.random.Generator or np.random.RandomState). (Optional, default: None) User-provided seed/state kept for provenance.

runcontrol.RNG (np.random.Generator). (Resolved, no user default) Created at initialisation from runcontrol.random_state; the engine routes all randomness through this generator.

3.7 Selection

(All entries accept a scalar or a per-population list unless noted.)

selection.tournament_size (int). (Optional, default: 2) Tournament size.

selection.elite_fraction (float). (Optional, default: 0.05) Elite fraction within a population.

selection.elite_fraction_ensemble (float). (Optional, default: 0.05) Elite fraction for cross-population ensemble building.

selection.tournament_lex_pressure (bool). (Optional, default: True) Lexicographic tie-breaking pressure.

selection.tournament_p_pareto (float). (Optional, default: 0.0) Probability of using a Pareto tournament.

selection.p_ensemble (float). (Optional, default: 0.0) Probability of sampling ensemble fitness during selection.

3.8 Nodes: Ephemeral Constants and Function Set

Ephemeral Random Constants (ERCs)

`nodes.const.about` (`str`). (Optional, default: `"Ephemeral random constants"`) Informational string (preserved).

`nodes.const.p_ERC` (`float` or list). (Optional, default: `0.1`) Probability of generating an ERC when creating a leaf.

`nodes.const.p_int` (`float` or list). (Optional, default: `0.0`) Probability that an ERC is integer-typed.

`nodes.const.range` (`list[2]` or list of pairs). (Optional, default: `[-10, 10]`) Sampling range; each `[min, max]` must satisfy `min < max`.

`nodes.const.num_dec_places` (`int` or list). (Optional, default: `4`) Decimal places for rounding real-valued ERCs.

Function Set

`nodes.functions.name` (`list[str]` or `list[list[str]]`). (Required) Operator names per population. Should be provided by the user as the name of corresponding operator script in the working path, e.g. `["times","minus","plus"]`

3.9 Genetic Operators

`operator.p_mutate` (`float` or list). (Optional, default: `0.14`) Probability of mutation.

`operator.p_cross` (`float` or list). (Optional, default: `0.84`) Probability of crossover.

`operator.p_direct` (`float` or list). (Optional, default: `0.02`) Probability of direct reproduction.

`operator.mutate_par` (`list[float]` of length `6`, or per-pop list). (Optional, default: `[0.9, 0.05, 0.05, 0.0, 0.0, 0.0]`) Sub-operator probabilities (normalised if sum $\neq 1$).

`operator.mutate_par_cumsum` (`list[list[float]]`). (Resolved, created at initialisation) Cumulative sums used for fast sampling.

`operator.mutate_gaussian_std` **(`float` or list).** (Optional, default: `0.1`) Std. dev. for Gaussian perturbation of numeric ERCs.

3.10 Gene-Level Controls

`gene.p_cross_hi` **(`float` or list).** (Optional, default: `0.2`) Probability of high-level (inter-gene) crossover.

`gene.hi_cross_rate` **(`float` or list).** (Optional, default: `0.5`) Probability any given gene participates in high-level crossover.

`gene.multigene` **(`bool` or list).** (Optional, default: `True`) Allow multiple genes per individual.

`gene.max_genes` **(`int` or list).** (Optional, default: `5`) Maximum number of genes per individual.

3.11 Tree Constraints

`tree.build_method` **(`int` or list).** (Optional, default: `3`) Tree builder method id `3` = ramped half-and-half, `2` = grow, `1` = full.

`tree.max_nodes` **(`int` or `np.inf` or list).** (Optional, default: `np.Inf`) Maximum node count per tree.

`tree.max_depth` **(`int` or list).** (Optional, default: `4`) Maximum tree depth.

`tree.max_mutate_depth` **(`int` or list).** (Optional, default: `4`) Depth cap for mutation-generated subtrees.

3.12 Fitness and Objectives

`fitness.terminate` **(`bool`).** (Optional, default: `False`) Enable early termination when threshold met.

`fitness.complexityMeasure` **(`int`).** (Optional, default: `1`) Complexity penalty mode: `1` (expressional complexity) or `0` (node count).

3.13 Softmax Head (Specific to `ClassifierConfig`)

These parameters control the softmax head used in `gpclassifier`. Parameters are broadcast per population unless a list is provided.

`softmax.optimizer_type` (`str` or list). (Optional, default: `rmsprop`) Name of the optimizer. Allowed values include `sgd`, `adam`, `rmsprop`, `adamw`, `nadam`, `adagrad`, `adadelta`, `ftrl`, `adamax`, `sgdnm`, `lbfgs`.

`softmax.optimizer_param` (`None` or `dict` or `list[dict]`). (Optional, default: `None`) Per-optimizer hyperparameters merged onto validated defaults. Unknown keys are ignored with a warning. Examples: `learning_rate`, `beta1`, `beta2`, `epsilon`, `weight_decay`.

`softmax.initializer` (`str` or list). (Optional, default: `glorot_uniform`) Weight/bias initializer family (e.g., `glorot_uniform`, `he_normal`, `random_normal`).

`softmax.regularization` (`None` or `str` or list). (Optional, default: `None`) Regularization type: `l1`, `l2`, or `hybrid`. If `hybrid` is used with a scalar `regularization_rate`, a warning is issued (hybrid expects a pair).

`softmax.regularization_rate` (`float` or list). (Optional, default: `0.01`) Regularization strength. For `softmax.regularization=hybrid`, pass a pair like `[l1, l2]`.

`softmax.batch_size` (`int` or list). (Optional, default: `32`) Mini-batch size.

`softmax.epochs` (`int` or list). (Optional, default: `1000`) Maximum training epochs.

`softmax.patience` (`int` or list). (Optional, default: `10`) Early-stopping patience.

`softmax.buffer_size` (`None` or `int` or list). (Optional, default: `None`) Optional shuffle/prefetch buffer size.

`softmax.shuffle` (`bool` or list). (Optional, default: `True`) Shuffle batches each epoch.

`softmax.verbose` (`int` or list). (Optional, default: `0`) Trainer verbosity.

3.14 ElasticNetCV Head (Specific to `RegressorConfig`)

These parameters control the ElasticNetCV regression head used in `gpregressor`. Parameters are broadcast per population unless a list is provided.

`linreg.l1_ratio` (`float` or list). (Optional, default: `1.0`) Mixing ratio between L1 and L2 penalties: `0.0` corresponds to Ridge, `1.0` to Lasso, intermediate values interpolate to Elastic Net.

`linreg.alphas` (`None` or `list[float]` or list of lists). (Optional, default: `None`) Explicit list of alpha values to try. If provided, this overrides `linreg.eps` and `linreg.n_alphas`.

`linreg.n_alphas` (`int` or list). (Optional, default: `100`) Number of alpha values in the automatically generated grid (ignored if `linreg.alphas` is set).

`linreg.eps` (`float` or list). (Optional, default: `1e-3`) When `linreg.alphas` is `None`, the alpha grid is logarithmically spaced between $\alpha_{\min} = \text{eps} \times \alpha_{\max}$ and $\alpha_{\max}$.

`linreg.fit_intercept` (`bool` or list). (Optional, default: `True`) Whether to fit an intercept. If features and targets are already centered, this can be set to `False`.

`linreg.copy_x` (`bool` or list). (Optional, default: `True`) If `True`, training data is copied; if `False`, it may be overwritten/centered in place (small performance gain, use with care).

`linreg.max_iter` (`int` or list). (Optional, default: `1000`) Maximum coordinate-descent iterations per fit. Increase if convergence warnings appear.

`linreg.tol` (`float` or list). (Optional, default: `1e-4`) Optimization stopping tolerance on the objective. Smaller values yield more precise but slower convergence.

`linreg.cv` (`int` or list). (Optional, default: `5`) Number of cross-validation folds.

`linreg.n_jobs` (`None` or `int` or list). (Optional, default: `None`) Number of parallel jobs for cross-validation. `None` means 1 unless in `joblib.parallel_backend` context.

`linreg.verbose` (`int` or list). (Optional, default: `0`) Verbosity level. `0` = silent, higher values increase output.

`linreg.positive` **(`bool` or list).** (Optional, default: `False`) Constrain coefficients to be non-negative. Useful for interpretability but can worsen fit if data are not scaled/centered.

`linreg.selection` **(`str` or list).** (Optional, default: `"cyclic"`) Coordinate-descent sweep order. Allowed values:

- `"cyclic"` (deterministic sweeps),
- `"random"` (randomised order, sometimes faster on large problems).

Head seed handling. The softmax head derives its reproducible integer seed from the resolved `runcontrol.RNG`; it does not accept a separate external seed. If `runcontrol.random_state` is a NumPy `Generator` (or a legacy `RandomState`), the integer is drawn from that source so that repeated runs with the same RNG are reproducible. If `runcontrol.random_state` is `None` or an `int`, behavior follows standard NumPy semantics.

3.15 Resolved dictionaries

After validation, two internal dictionaries are constructed:

`config` **(`dict`).** Resolved engine configuration with sections `runcontrol`, `selection`, `nodes`, `operator`, `gene`, `tree`, `fitness`, and `softmax`. Values are fully broadcast and type-checked.

`userdata` **(`dict`).** Immutable copies of user arrays and metadata: `name`, `stats`, population indices, `xtrain/xval/xtest`, `ytrain/yval/ytest`, and binarised labels used by the classifier.

Label remapping. If the unique sorted labels in `user.ytrain` are not `0..K-1`, they are remapped internally (a warning is emitted). Predictions and reports use the remapped indexing unless you invert-map externally.

3.16 Common validation rules (summary)

- Arrays must have expected dimensionality: features strictly 2D; labels 1D or a single 2D column (flattened internally).

- Probabilities must lie in `[0,1]` unless specified otherwise.
- ERC ranges `[min, max]` must satisfy `min < max`.
- Lists provided for per-population parameters must match `runcontrol.num_pop`; scalars are broadcast.
- Unknown optimizer types are rejected; unknown optimizer hyperparameters are warned and ignored.
- No global `np.random` seeding: all randomness flows through `runcontrol.RNG`.

3.17 Complete example

Listing 3.3: Putting it together: a compact configuration for two populations.

```
from genforge.gpclassifier import gpclassifier, ClassifierConfig
import numpy as np

cfg = ClassifierConfig()

# data
cfg.user.name = "demo_clf_two_pop"
cfg.user.xtrain = X_train
cfg.user.ytrain = y_train
cfg.user.xval = X_val
cfg.user.yval = y_val
cfg.user.xtest = X_test
cfg.user.ytest = y_test

# tie features to populations (two groups)
cfg.user.pop_idx = [list(range(0, 20)), list(range(20, 40))]

# run control
cfg.runcontrol.num_pop = 2
cfg.runcontrol.pop_size = 64
cfg.runcontrol.generations = 50
cfg.runcontrol.useparallel = True
```

```
cfg.runcontrol.parallel.n_jobs = 4
cfg.runcontrol.parallel.batch_job = 8
cfg.runcontrol.random_state = 123

# nodes/functions
cfg.nodes.functions.name = [
    ["times","minus","plus","divide"],
    ["times","minus","plus","divide","sin"]
]
cfg.nodes.const.p_ERC = [0.3, 0.3]
cfg.nodes.const.p_int = [0.2, 0.2]
cfg.nodes.const.range = [[-3.0, 3.0], [-1.0, 1.0]]
cfg.nodes.const.num_dec_places = [3, 3]

# selection
cfg.selection.tournament_size = [5, 5]
cfg.selection.elite_fraction = [0.05, 0.05]
cfg.selection.elite_fraction_ensemble = [0.05, 0.05]
cfg.selection.tournament_lex_pressure = [True, True]
cfg.selection.tournament_p_pareto = [0.5, 0.5]
cfg.selection.p_ensemble = [0.5, 0.5]

# operator
cfg.operator.p_mutate = [0.40, 0.40]
cfg.operator.p_cross = [0.50, 0.50]
cfg.operator.p_direct = [0.10, 0.10]
cfg.operator.mutate_par = [[0.20,0.20,0.20,0.20,0.10,0.10]] # broadcasted
cfg.operator.mutate_gaussian_std = [0.10, 0.10]

# gene / tree
cfg.gene.multigene = [True, True]
cfg.gene.max_genes = [5, 5]
cfg.gene.p_cross_hi = [0.3, 0.3]
cfg.gene.hi_cross_rate = [0.5, 0.5]

cfg.tree.build_method = [0, 0]
cfg.tree.max_nodes = [np.inf, np.inf]
cfg.tree.max_depth = [10, 10]
```

```
cfg.tree.max_mutate_depth = [8, 8]

# fitness
cfg.fitness.terminate = False
cfg.fitness.complexityMeasure = 0

# softmax head
cfg.softmax.optimizer_type = ["adam", "adam"]
cfg.softmax.optimizer_param = [{"learning_rate": 0.001}, {"learning_rate": 0.001}]
cfg.softmax.initializer = ["glorot_uniform", "glorot_uniform"]
cfg.softmax.regularization = ["l2", "l2"]
cfg.softmax.regularization_rate = [1e-4, 1e-4]
cfg.softmax.batch_size = [64, 64]
cfg.softmax.epochs = [100, 100]
cfg.softmax.patience = [10, 10]
cfg.softmax.buffer_size = [None, None]
cfg.softmax.shuffle = [True, True]
cfg.softmax.verbose = [0, 0]


gp = gpclassifier.initialize(cfg)
gp.evolve()
```

4 SPFP: Mathematics and Algorithmic Details

Semantic Preserving Feature Partitioning (`SPFPPartitioner`) constructs P feature groups (*findex*) to drive multi-population genetic programming. This chapter formalises the notation, discretisation, scoring, forward selection with growth constraints, optional backward pruning, and the final group formation policy.

4.1 Notation and Discretisation

Let $X \in \mathbb{R}^{N \times D}$ be the input matrix and $y \in \mathbb{R}^N$ the target. The pipeline standardises X feature-wise to zero mean and unit variance, and label-encodes y if needed. For entropy-based quantities, variables are discretised into B equal-distance bins (`n_bins`).

For any index set $S \subseteq \{1, \ldots, D\}$, define the discrete random vector X_S as the concatenation of the binned features $\{X_j\}_{j \in S}$. Denote the joint entropy

$$H(X_S) \;=\; -\sum_{\mathbf{x}} p(\mathbf{x}) \, \log p(\mathbf{x}), \tag{4.1}$$

and similarly $H(X_S, y)$ and $H(y)$. `SPFPPartitioner` uses the normalised mutual information (NMI) with respect to $H(y)$:

$$\mathrm{NMI}(X_S; y) \;=\; \frac{H(X_S) + H(y) - H(X_S, y)}{H(y)}. \tag{4.2}$$

All joint entropies are computed by a joint-histogram routine with consistent bin edges (training edges reused for test, when given).

4.2 Per-Feature Scoring and Ranking

For each feature $i \in \{1, \ldots, D\}$, compute

$$H_i := H(X_{\{i\}}), \tag{4.3}$$

$$H_{i,y} := H(X_{\{i\}}, y), \tag{4.4}$$

$$\mathrm{mi}_i := \frac{H_i + H(y) - H_{i,y}}{H(y)}. \tag{4.5}$$

On the *continuous, standardised* features and label-encoded y, compute the Pearson correlation ρ_i, and define the auxiliary score

$$\mathrm{micor}_i \;=\; \mathrm{mi}_i + |\rho_i|. \tag{4.6}$$

The initial *stream* of candidate features is sorted by `mi` in descending order.[1]

4.3 Group-Wise Forward Selection

`SPFPPartitioner` builds $P =$ `num_groups` groups sequentially. At the beginning of a group, let A be the set of remaining candidates (a copy of the current stream). Define the group-level reference values

$$H_A := H(X_A), \qquad H_{A,y} := H(X_A, y), \qquad \mathrm{NMI}_A := \frac{H_A + H(y) - H_{A,y}}{H(y)}. \tag{4.7}$$

Let S be the set of *selected* indices for the current group (initially empty). Define the running quantities $H_S := H(X_S)$ and $H_{S,y} := H(X_S, y)$, and the current normalised mutual information

$$\mathrm{NMI}_S \;=\; \frac{H_S + H(y) - H_{S,y}}{H(y)}. \tag{4.8}$$

Stopping rule. Selection continues while

$$\Big(H_{S,y} < \alpha \cdot \lfloor H_{A,y} \rfloor_\delta\Big) \;\vee\; \Big(H_S < \alpha \cdot \lfloor H_A \rfloor_\delta\Big) \;\vee\; \Big(|S| < \lceil \pi \cdot D \rceil\Big), \tag{4.9}$$

and there remain candidates in A. Here $\alpha =$ `criterion_add` (`wcriadd`), $\pi =$ `perfs`, and $\lfloor \cdot \rfloor_\delta$ is a fixed-decimal floor with $\delta = 10^{-\texttt{floor\_decimals}}$ (the reference uses 4 decimals in this context).

[1] This mirrors the reference script, which sorts by `mi` and then constructs groups from that stream.

First pick. On the first iteration, select the top-ranked candidate in A (the head of the stream), add it to S, and update H_S, $H_{S,y}$.

Composite objective. From the second iteration onward, maintain per-candidate components for each $j \in A$ (with current $S \neq \varnothing$):

$$\text{objmi}_j := \text{mi}_j, \qquad \text{objcor}_j := |\rho_j|, \tag{4.10}$$

$$\text{objint}_j := \frac{1}{|S|} \sum_{k \in S} \frac{-H(y) - H(X_{\{j,k\}}, y) + H_{j,y} + H_{k,y}}{H(y)}, \tag{4.11}$$

$$\text{objred}_j := \frac{1}{|S|} \sum_{k \in S} \frac{H_j + H_k - H(X_{\{j,k\}})}{H(y)}, \tag{4.12}$$

and the additive composite

$$\text{obj}_j \;=\; \text{objmi}_j + \text{objcor}_j + \text{objint}_j + \text{objred}_j. \tag{4.13}$$

Equations (4.11) and (4.12) mirror the reference implementation: conditional interaction uses $I(X_j; X_k \mid y)$ scaled by $H(y)$, and redundancy uses $I(X_j; X_k)$ scaled by $H(y)$; both are averaged over the currently selected set S.

Selection policy. On iteration $t \geq 2$, `SPFPPartitioner` chooses the next candidate by one of the following branches (exactly as implemented):

(i) **Objective gate.** If both coverage thresholds in (4.9) are already met but $|S| < \lceil \pi D \rceil$, choose the maximiser of a user-chosen primary score `objective` $\in$ {`"mi"`, `"cor"`, `"micor"`}:

$$j^\star \in \arg\max_{j \in A} \begin{cases} \text{mi}_j, & \texttt{"mi"} \\ |\rho_j|, & \texttt{"cor"} \\ \text{mi}_j + |\rho_j|, & \texttt{"micor"} \end{cases}$$

(ii) **Override.** If `override` is `True`, choose $j^\star \in \arg\max_{j \in A} \text{obj}_j$.

(iii) **Growth-gated search.** Otherwise, scan candidates in decreasing obj_j order while enforcing

conservative growth:

$$d_S(j) := \frac{H(X_{S\cup\{j\}}) - H_S}{\alpha \cdot \lfloor H_A \rfloor_\delta}, \qquad d_{S,y}(j) := \frac{H(X_{S\cup\{j\}}, y) - H_{S,y}}{\alpha \cdot \lfloor H_{A,y} \rfloor_\delta}. \tag{4.14}$$

For the running maxima H_S and $H_{S,y}$, let $\gamma =$ `growth_magnitude` (`whmag`). If

$$\Big(H_{S,y} < \alpha \lfloor H_{A,y} \rfloor_\delta \;\text{ or }\; H_S < \alpha \lfloor H_A \rfloor_\delta\Big) \;\text{ and }\; \Big(H(X_{S\cup\{j\}}, y) \le \min(\gamma H_{S,y},\, \alpha \lfloor H_{A,y} \rfloor_\delta) \;\text{ or }\; H(X_{S\cup\{j\}}) \le \text{r}$$

record $d_S(j)$ and $d_{S,y}(j)$ and *temporarily* suppress obj_j (set to $-\infty$) to continue scanning. After the scan:

$$j^\star \;=\; \begin{cases} \arg\max\limits_{j:\, d_S(j)>0 \wedge d_{S,y}(j)>0} \big(d_S(j) + d_{S,y}(j)\big), & \text{if any} \\ \arg\max\limits_{j:\, d_{S,y}(j)>0} d_{S,y}(j), & \text{else if any} \\ \arg\max\limits_{j:\, d_S(j)>0} d_S(j), & \text{else if any} \\ \text{first scanned candidate}, & \text{otherwise.} \end{cases}$$

Once $j^\star$ is chosen, update $S \leftarrow S \cup \{j^\star\}$, recompute $H_S, H_{S,y}$, and refresh (4.11)–(4.13). This loop continues until the stopping rule (4.9) fails.

4.4 Optional Backward Pruning

If enabled (`backward` = `True`), pruning iteratively removes features from S as long as:

$$H_{S,y} \ge \beta \cdot \lfloor H_{S,y}^{\text{base}} \rfloor_\delta, \qquad H_S \ge \beta \cdot \lfloor H_S^{\text{base}} \rfloor_\delta, \qquad |S| > \lceil \pi D \rceil, \tag{4.15}$$

where $\beta =$ `criterion_rem` (`wcrirem`), and $H_S^{\text{base}}, H_{S,y}^{\text{base}}$ are computed on the full current S before pruning starts.

For each $k \in S$, form the leave-one-out set $S \setminus \{k\}$, compute its entropies (consistent with forward definitions), its normalised mutual information, and interaction/redundancy summaries analogous to (4.11)–(4.12). Let

$$\text{obj}_k^{\text{back}} \;=\; \text{NMI}_{S\setminus\{k\}} \;+\; \frac{1}{|S|-1} \sum I(\cdot;\cdot \mid y) \;-\; \tfrac{1}{|S|-1} \sum I(\cdot;\cdot), \tag{4.16}$$

mirroring the script's sign convention in pruning.[2] Define admissible removals via threshold tests on H_S and $H_{S,y}$ (and an additional NMI floor tied to NMI_A in (4.7)), and remove the k with the *smallest* $\mathrm{obj}_k^{\mathrm{back}}$ among admissible candidates. Repeat while (4.15) holds.

4.5 Forming Multiple Groups and Partition Policy

At the end of a group, `SPFPPartitioner` samples without replacement a fraction ρ = `remp` of the selected indices S and removes those from the global stream. The remaining features stay available for subsequent groups. This random withholding encourages disjointness between groups and spreads high-value features across populations.

After P groups or exhaustion of the stream, `SPFPPartitioner` returns

$$\mathcal{P} = \{S_1, S_2, \ldots, S_P\},$$

where each S_g is the (possibly empty) set of feature indices selected for group g.

4.6 Parameter Summary

Table 4.1: SPFP key parameters and recommended defaults (matching the reference implementation).

Name	Meaning	Typical default
`num_groups`	Number of groups P to construct	5
`n_bins`	Bins for equal-distance discretisation	100
`objective`	Primary score in branch (i)	`"mi"` or `"micor"`
`criterion_add`	α in (4.9)	0.999–1.0
`criterion_rem`	β in (4.15)	0.999–1.0
`growth_magnitude`	γ growth cap in gated search	1.01
`perfs`	π minimal ratio of D per group	0.10 (dataset-dependent)
`remp`	ρ random withholding rate	0.60
`override`	Use composite objective directly	`False`
`backward`	Enable pruning phase	`False`
`random_state`	Seed for reproducibility	(set as needed)

[2] In the reference script, the redundancy term is subtracted during backward pruning and added during forward selection.

4.7 Complexity and Implementation Notes

- **Entropy evaluations.** The dominant cost arises from repeated joint entropies such as $H(X_{S\cup\{j\}})$, $H(X_{S\cup\{j\}}, y)$, $H(X_{\{j,k\}})$, and $H(X_{\{j,k\}}, y)$. The reference uses parallel map for the pairwise terms involving the most recent selection.
- **Normalisation by $H(y)$.** All MI-derived terms are scaled by $H(y)$, exactly as in the script, to keep units comparable across datasets.
- **Flooring.** Threshold comparisons use decimal flooring on H_A and $H_{A,y}$ with 4 decimals (see (4.9)).
- **Correlation.** Correlations are computed on the continuous, standardised X against label-encoded y; only $|\rho|$ is used.
- **Forward vs. backward signs.** Redundancy contributes positively in forward selection (added to (4.13)) but is penalised in pruning (subtracted in (4.16)).

4.8 Pseudocode

Listing 4.1: SPFP high-level pseudocode (mirrors the reference).

```
# Inputs: X (N x D), y (N,), num_groups P, n_bins, objective, criterion_add alpha,
# criterion_rem beta, growth_magnitude gamma, perfs pi, remp rho, override flag,
# backward flag, random_state.
# Pre: standardise X; label-encode y; bin X, y for entropy ops.

# 1) Per-feature scores
for i in 1..D:
    Hx[i] = H({i})
    Hxy[i] = H({i} + {y})
    mi[i] = (Hx[i] + Hy - Hxy[i]) / Hy
    cor[i] = abs(pearson(x_i, y_cont))
micor[i] = mi[i] + cor[i]
stream = argsort(mi)[desc] # initial order

# 2) Build P groups
for g in 1..P:
    A = list(stream) # remaining candidates
```

```
    S = [] # selected indices
    H_A = H(A); H_Ay = H(A + {y})
    while (H(S,y) < alpha*floor(H_Ay) or H(S) < alpha*floor(H_A) or len(S) < ceil(pi*D)) and A:
        if len(S) == 0:
            j_star = A[0]
        else:
            update objective components objmi, objcor, objint, objred
            obj = objmi + objcor + objint + objred
            if thresholds met but len(S) < ceil(pi*D):
                j_star = argmax objective_score in {"mi","cor","micor"}
            elif override:
                j_star = argmax obj
            else:
                gated-scan over sorted obj to compute dS, dSy per candidate
                choose by (dS>0 and dSy>0) -> max dS+dSy
                       elif dSy>0 -> max dSy
                       elif dS>0 -> max dS
                       else fallback to first scanned
        S.append(j_star); A.remove(j_star)
        refresh H(S), H(S,y), and pairwise {H(j_star,f), H(j_star,f,y)} for f in A
        update interaction/redundancy matrices and objective components

    # Optional backward pruning
    if backward:
        while guards with beta thresholds and len(S) > ceil(pi*D):
            compute leave-one-out metrics and back-objective
            remove admissible k with smallest back-objective

    # Partition policy: randomly remove floor(rho*len(S)) from global stream
    remove_sample = random.sample(S, floor(rho*len(S)))
    stream = stream \ remove_sample
    save group S
return {S_1, ..., S_P}
```

5 gpclassifier: Legacy Configuration & API

This chapter documents the gpclassifier module: data requirements, the training pipeline, configuration keys (with defaults), integration with SPFPPartitioner, and the expected outputs (plots and HTML reports).

5.1 Overview & Data Requirements

Task. gpclassifier implements multi-population Genetic Programming (GP) for supervised classification. Multiple co-evolving populations contribute to an ensemble head that mixes population outputs into final class scores.

Inputs. Provide NumPy arrays:

- `userdata_xtrain`: shape (N_{tr}, D),
- `userdata_ytrain`: shape $(N_{\text{tr}},)$ with integer labels,
- (optional) `userdata_xval`, `userdata_yval`,
- (optional) `userdata_xtest`, `userdata_ytest`.

If you use feature partitions from SPFPPartitioner, pass them via `userdata_pop_idx` (see Section 5.4).

5.2 Training Pipeline

At a high level, a training call:

1. builds initial populations (optionally restricted by `userdata_pop_idx`),
2. evolves for `runcontrol_generations` using selection, crossover, and mutation under structural limits,

3. fits an ensemble mixing head on population outputs (using the validation split when available),
4. evaluates and emits artifacts (plots, HTML reports, run snapshots).

5.3 Configuration Keys

The initializer expects a flat dictionary of keys. Below we group the most commonly used keys by purpose. Values shown as *defaults* are representative and may be dataset-dependent.

5.3.1 User data & run metadata

Table 5.1: User data and metadata keys for `gpclassifier`.

Key	Default	Description
`userdata_name`	`"run"`	Logical run name; used in filenames.
`userdata_xtrain`	—	Training features, (N_{tr}, D).
`userdata_ytrain`	—	Training labels, $(N_{\text{tr}},)$.
`userdata_xval`	`None`	Validation features (optional).
`userdata_yval`	`None`	Validation labels (optional).
`userdata_xtest`	`None`	Test features (optional).
`userdata_ytest`	`None`	Test labels (optional).
`userdata_pop_idx`	`None`	Feature partitions (list of lists) from `SPFPPartitioner`.

5.3.2 Run control & parallelism

Table 5.2: Run control and parallel settings.

Key	Default	Description
`runcontrol_generations`	`50`	Number of generations.
`runcontrol_pop_size`	`64`	Individuals per population.
`runcontrol_useparallel`	`False`	Enable parallel fitness.
`runcontrol_n_jobs`	`None`	Workers (if parallel).
`runcontrol_batch_size`	`None`	Work chunk per worker.

5.3.3 Function sets, tree/gene constraints

Table 5.3: Structural configuration (function sets, depth, genes).

Key	Default	Description
`functions_name`	`[["times","minus","plus","divide"]]`	Function set per population.
`tree_max_depth`	`[10]`	Max depth per population.
`gene_max_genes`	`[5]`	Max genes per chromosome (per population).

5.3.4 Selection & ensemble

Table 5.4: Selection and ensemble hyperparameters.

Key	Default	Description
`selection_elite_fraction`	`[0.05]`	Elitism per population.
`selection_elite_fraction_ensemble`	`[0.05]`	Elitism for ensemble stage.
`selection_p_ensemble`	`[0.5]`	Probability of choosing ensemble candidates.

5.3.5 Plotting & reporting

Table 5.5: Plot and report toggles.

Key	Default	Description
`runcontrol_plotfitness`	`True`	Plot Fitness vs. Generation.
`runcontrol_plotrankbest`	`True`	Plot RankBest curve.
`runcontrol_plotrankall`	`False`	Plot RankAll curves.
`runcontrol_plotsavefig`	`True`	Save plots to disk.

Remark 5.1 (About defaults)**.** Exact defaults can vary across releases. The keys above mirror the common usage patterns in the examples. When in doubt, consult the repository `README.md` and the source of `gp_class.py`.

5.4 Using SPFP Partitions

If you precompute feature partitions with `SPFPPartitioner` (see Chapter 4), pass them as a Python list of lists via `userdata_pop_idx`. The number of populations should match the number of groups:

Listing 5.1: Wiring SPFP partitions to populations.

```
from genforge.gpclassifier import gpclassifier
from genforge.spfp.spfp_partition import SPFPPartitioner

# X, y are NumPy arrays
spfp = SPFPPartitioner(n_groups=4, objective="micor", random_state=13)
spfp.fit(X, y)
findex = spfp.get_partitions() # e.g., [[0,7,9],[1,2,5],...]

params = {
    "userdata_name": "clf_with_spfp",
    "userdata_xtrain": X_train, "userdata_ytrain": y_train,
    "userdata_xval": X_val, "userdata_yval": y_val,
    "userdata_xtest": X_test, "userdata_ytest": y_test,
    "userdata_pop_idx": findex, # <- map groups to populations
    "runcontrol_generations": 50,
    "runcontrol_pop_size": 64,
    "functions_name": [["times","minus","plus","divide"]],
    "gene_max_genes": [5], "tree_max_depth": [10],
    "selection_elite_fraction": [0.05],
    "selection_elite_fraction_ensemble": [0.05],
    "selection_p_ensemble": [0.5],
    "runcontrol_plotfitness": True,
    "runcontrol_plotrankbest": True,
    "runcontrol_plotsavefig": True,
}
gp = gpclassifier.initialize(**params)
gp.evolve()
gp.report(ensemble_row=0)
```

5.5 Training & Evaluation Workflow

A minimal end-to-end script looks like Listing 5.2. Replace the paths to your arrays, and adjust `functions_name`, tree depth, and generations to suit your problem scale.

Listing 5.2: Minimal `gpclassifier` workflow.

```python
from genforge.gpclassifier import gpclassifier
import numpy as np

X_train, y_train = np.load("X_train.npy"), np.load("y_train.npy")
X_val, y_val = np.load("X_val.npy"), np.load("y_val.npy")
X_test, y_test = np.load("X_test.npy"), np.load("y_test.npy")

params = {
    "userdata_name": "demo_clf",
    "userdata_xtrain": X_train, "userdata_ytrain": y_train,
    "userdata_xval": X_val, "userdata_yval": y_val,
    "userdata_xtest": X_test, "userdata_ytest": y_test,
    "runcontrol_generations": 50,
    "runcontrol_pop_size": 64,
    "runcontrol_useparallel": False,
    "functions_name": [["times","minus","plus","divide"]],
    "gene_max_genes": [5],
    "tree_max_depth": [10],
    "selection_elite_fraction": [0.05],
    "selection_elite_fraction_ensemble": [0.05],
    "selection_p_ensemble": [0.5],
    "runcontrol_plotfitness": True,
    "runcontrol_plotrankbest": True,
    "runcontrol_plotsavefig": True,
}

gp = gpclassifier.initialize(**params)
gp.evolve()

# Write an HTML ensemble report for a chosen row (e.g., best on validation)
gp.report(ensemble_row=0) # creates an HTML file in the run's folder
```

5.6 Outputs: Plots and HTML Reports

Plots. If `runcontrol_plotfitness` or `runcontrol_plotrankbest` are enabled, the trainer writes PNGs (or configured formats) to the run's output directory. Typical figures include:

- Fitness vs. Generation (per population and/or aggregate),
- RankBest vs. Generation,
- (optional) RankAll vs. Generation.

HTML report. `gp.report(ensemble_row=...)` writes an HTML summary for the chosen ensemble row, typically including:

- per-population best trees (rendered),
- ensemble mixing weights and bias,
- validation/test metrics,
- run configuration snapshot.

Remark 5.2 (Assets to include)**.** Please export and place the following under `Figure/` (or your chosen figures folder), then recompile:

1. `fig_clf_fitness.png`: Fitness vs. Generation from an actual run (referenced as Figure 5.1).
2. `fig_clf_report.png`: A cropped screenshot of the HTML report (referenced as Figure 5.2).

5.7 Tips & Common Pitfalls

- **Mismatched partitions.** When using `SPFPPartitioner`, ensure `userdata_pop_idx` has the same length as the number of populations implied by your configuration.
- **Function set arity.** The available primitives (‘"times"‘, ‘"minus"‘, ‘"plus"‘, ‘"divide"‘, etc.) must match the arity expected by the tree builder; mixing incompatible sets can stall evolution.
- **Over-constrained structures.** Very small `tree_max_depth` or `gene_max_genes` can severely limit search; if fitness does not improve, relax these bounds.
- **Parallel evaluation.** If enabling `runcontrol_useparallel`, set `runcontrol_n_jobs` explicitly for predictable CPU usage.

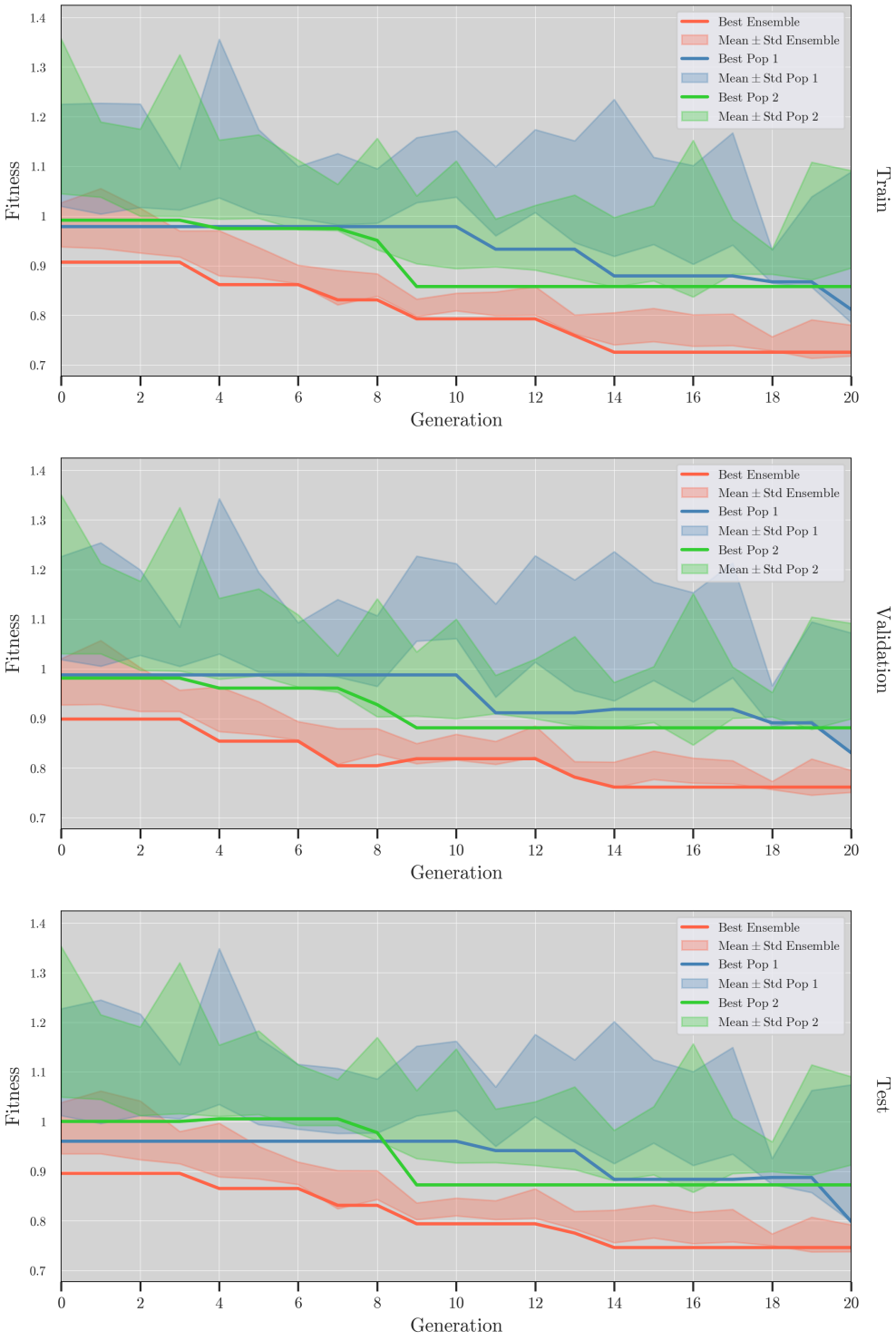


Figure 5.1: Example `gpclassifier` Fitness vs. Generation.

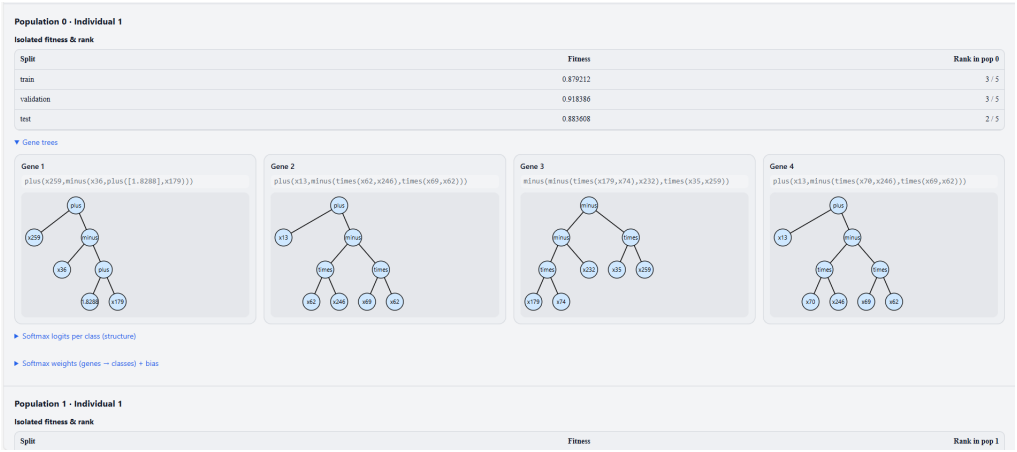


Figure 5.2: Rendered `gpclassifier` HTML report.

6 gpregressor: Legacy Configuration & API

This chapter documents the gpregressor module: data requirements, training workflow, configuration keys (with representative defaults), how to integrate SPFPPartitioner partitions, and the produced artifacts (plots and HTML reports).

6.1 Overview & Data Requirements

Task. gpregressor implements multi-population Genetic Programming (GP) for supervised regression. Multiple co-evolving populations produce scalar outputs that are linearly mixed by an ensemble head to form the final prediction.

Inputs. Provide NumPy arrays:

- `userdata_xtrain`: shape (N_{tr}, D),
- `userdata_ytrain`: shape $(N_{\text{tr}},)$ with real targets,
- (optional) `userdata_xval`, `userdata_yval`,
- (optional) `userdata_xtest`, `userdata_ytest`.

If you use feature partitions from SPFPPartitioner, pass them via `userdata_pop_idx` (see Section 6.4).

6.2 Training Pipeline

A standard training call:

1. initializes populations (optionally restricted by `userdata_pop_idx`),
2. evolves for `runcontrol_generations` with selection, crossover, and mutation under structural limits,

3. fits an ensemble mixing head on population outputs (validation-aware when available),

4. evaluates and emits plots/reports/run-state.

6.3 Configuration Keys

6.3.1 User data & run metadata

Table 6.1: User data and metadata keys for `gpregressor`.

Key	Default	Description
`userdata_name`	`"run"`	Logical run name.
`userdata_xtrain`	—	Training features, (N_{tr}, D).
`userdata_ytrain`	—	Training targets, $(N_{\text{tr}},)$.
`userdata_xval`	`None`	Validation features (optional).
`userdata_yval`	`None`	Validation targets (optional).
`userdata_xtest`	`None`	Test features (optional).
`userdata_ytest`	`None`	Test targets (optional).
`userdata_pop_idx`	`None`	Feature partitions (list of lists) from `SPFPPartitioner`.

6.3.2 Run control & parallelism

Table 6.2: Run control and parallel settings.

Key	Default	Description
`runcontrol_generations`	`50`	Number of generations.
`runcontrol_pop_size`	`64`	Individuals per population.
`runcontrol_useparallel`	`False`	Enable parallel fitness.
`runcontrol_n_jobs`	`None`	Workers (if parallel).
`runcontrol_batch_size`	`None`	Work chunk per worker.

Table 6.3: Structural configuration (function sets, depth, genes).

Key	Default	Description
`functions_name`	`[["times","minus","plus","divide"]]`	Function set per population.
`tree_max_depth`	`[10]`	Max depth per population.
`gene_max_genes`	`[5]`	Max genes per chromosome (per population).

6.3.3 Function sets, tree/gene constraints

6.3.4 Selection & ensemble

Table 6.4: Selection and ensemble hyperparameters.

Key	Default	Description
`selection_elite_fraction`	`[0.05]`	Elitism per population.
`selection_elite_fraction_ensemble`	`[0.05]`	Elitism for ensemble stage.
`selection_p_ensemble`	`[0.5]`	Probability of ensemble candidate selection.

6.3.5 Plotting & reporting

Table 6.5: Plot and report toggles.

Key	Default	Description
`runcontrol_plotfitness`	`True`	Plot Fitness vs. Generation.
`runcontrol_plotrankbest`	`True`	Plot RankBest curve.
`runcontrol_plotrankall`	`False`	Plot RankAll curves.
`runcontrol_plotsavefig`	`True`	Save plots to disk.

Remark 6.1 (About defaults)**.** Defaults are representative and may vary by release. Consult the repository `README.md` and the source `gp_regress.py` for authoritative values.

6.4 Using SPFP Partitions

When using `SPFPPartitioner` (see Chapter 4), pass the resulting feature index groups via `userdata_pop_idx`. Ensure the number of groups equals the number of populations implied by your configuration:

Listing 6.1: Wiring SPFP partitions to `gpregressor`.

```
from genforge.gpregressor import gpregressor
from genforge.spfp.spfp_partition import SPFPPartitioner

spfp = SPFPPartitioner(n_groups=3, objective="micor", random_state=19)
spfp.fit(X, y)
findex = spfp.get_partitions()

params = {
    "userdata_name": "reg_with_spfp",
    "userdata_xtrain": X_train, "userdata_ytrain": y_train,
    "userdata_xval": X_val, "userdata_yval": y_val,
    "userdata_xtest": X_test, "userdata_ytest": y_test,
    "userdata_pop_idx": findex, # <- map groups to populations
    "runcontrol_generations": 50,
    "runcontrol_pop_size": 64,
    "functions_name": [["times","minus","plus","divide"]],
    "gene_max_genes": [5], "tree_max_depth": [10],
    "selection_elite_fraction": [0.05],
    "selection_elite_fraction_ensemble": [0.05],
    "selection_p_ensemble": [0.5],
    "runcontrol_plotfitness": True,
    "runcontrol_plotrankbest": True,
    "runcontrol_plotsavefig": True,
}
gp = gpregressor.initialize(**params)
gp.evolve()
gp.report(ensemble_row=0)
```

6.5 Training & Evaluation Workflow

Listing 6.2 shows a minimal end-to-end example. Adjust the structural constraints and function set to match your problem scale and variable types.

Listing 6.2: Minimal `gpregressor` workflow.

```
from genforge.gpregressor import gpregressor
import numpy as np

X_train, y_train = np.load("Xtr.npy"), np.load("ytr.npy")
X_val, y_val = np.load("Xva.npy"), np.load("yva.npy")
X_test, y_test = np.load("Xte.npy"), np.load("yte.npy")

params = {
    "userdata_name": "demo_reg",
    "userdata_xtrain": X_train, "userdata_ytrain": y_train,
    "userdata_xval": X_val, "userdata_yval": y_val,
    "userdata_xtest": X_test, "userdata_ytest": y_test,
    "runcontrol_generations": 50,
    "runcontrol_pop_size": 64,
    "runcontrol_useparallel": False,
    "functions_name": [["times","minus","plus","divide"]],
    "gene_max_genes": [5],
    "tree_max_depth": [10],
    "selection_elite_fraction": [0.05],
    "selection_elite_fraction_ensemble": [0.05],
    "selection_p_ensemble": [0.5],
    "runcontrol_plotfitness": True,
    "runcontrol_plotrankbest": True,
    "runcontrol_plotsavefig": True,
}

gp = gpregressor.initialize(**params)
gp.evolve()

# Write an HTML ensemble report for a chosen row (e.g., best on validation)
gp.report(ensemble_row=0)
```

6.6 Outputs: Plots and HTML Reports

Plots. If `runcontrol_plotfitness` and/or `runcontrol_plotrankbest` are enabled, the trainer saves figures to the run's output directory. Common plots:

- Fitness vs. Generation,
- RankBest vs. Generation,
- (optional) RankAll vs. Generation.

HTML report. `gp.report(ensemble_row=...)` writes a self-contained HTML summary with:

- best trees per population (rendered),
- ensemble mixing weights and bias,
- validation/test metrics (e.g., MAE, RMSE as available),
- configuration snapshot.

Remark 6.2 (Assets to include)**.** Please export and place the following under `Figure/`, then recompile:

1. `fig_reg_fitness.png`: Fitness vs. Generation from a real run (for Figure 6.1).
2. `fig_reg_report.png`: Cropped screenshot of the regression HTML report (for Figure 6.2).

6.7 Tips & Common Pitfalls

- **Scale of targets.** Extremely large or small target magnitudes can affect numerical stability. Consider rescaling `userdata_ytrain` (and corresponding splits) if necessary.
- **Partitions vs. populations.** When using `SPFPPartitioner`, ensure the number of groups equals the number of populations; otherwise, features may be unassigned or over-constrained.
- **Search budget.** If fitness stagnates, increase `runcontrol_generations`, `runcontrol_pop_size`, or relax `tree_max_depth`/`gene_max_genes`.
- **Parallel evaluation.** When enabling `runcontrol_useparallel`, set `runcontrol_n_jobs` explicitly for predictable CPU usage.

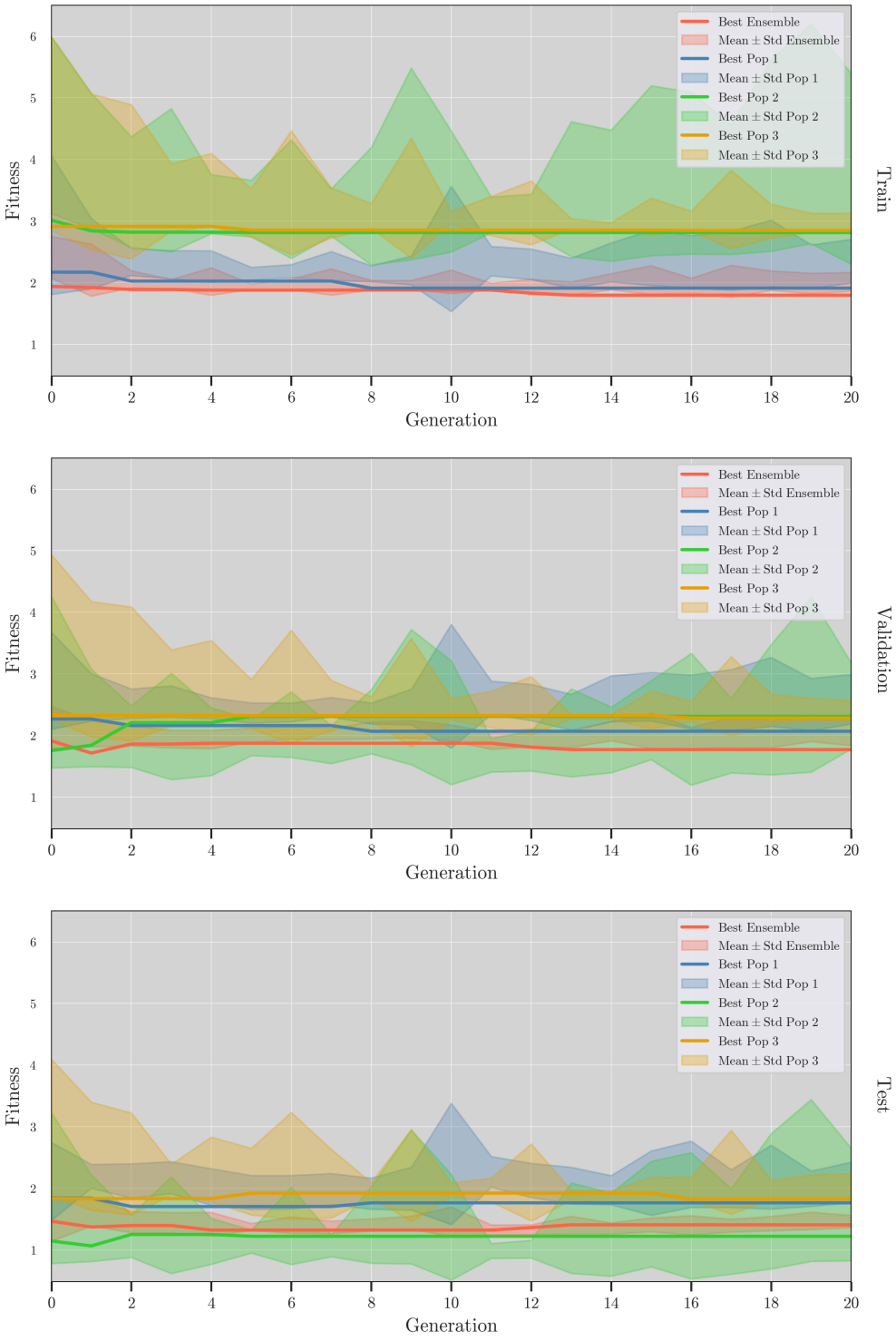
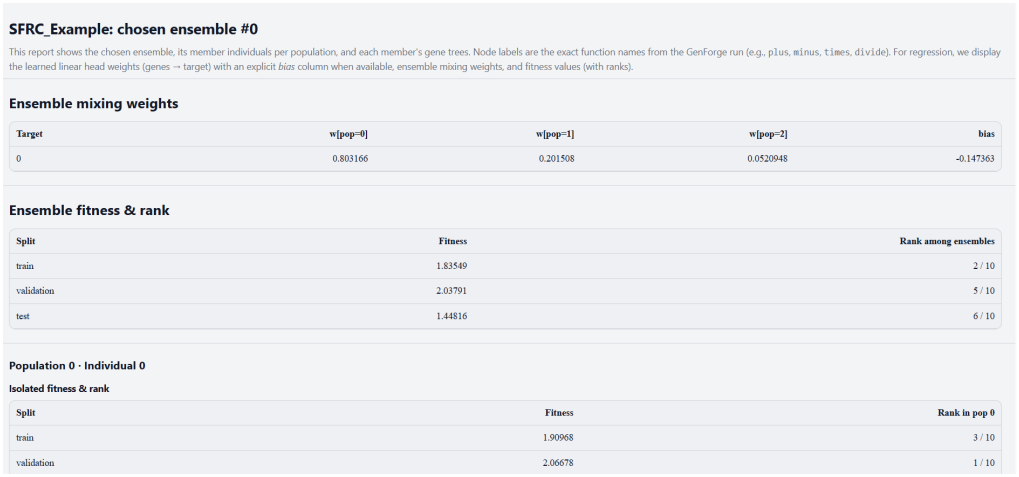


Figure 6.1: Example `gpregressor` Fitness vs. Generation.



SFRC_Example: chosen ensemble #0

This report shows the chosen ensemble, its member individuals per population, and each member's gene trees. Node labels are the exact function names from the GenForge run (e.g., `plus`, `minus`, `times`, `divide`). For regression, we display the learned linear head weights (genes → target) with an explicit *bias* column when available, ensemble mixing weights, and fitness values (with ranks).

Ensemble mixing weights

Target	w[pop=0]	w[pop=1]	w[pop=2]	bias
0	0.803166	0.201508	0.0520948	-0.147363

Ensemble fitness & rank

Split	Fitness	Rank among ensembles
train	1.83549	2 / 10
validation	2.03791	5 / 10
test	1.44816	6 / 10

Population 0 · Individual 0

Isolated fitness & rank

Split	Fitness	Rank in pop 0
train	1.90968	3 / 10
validation	2.06678	1 / 10

Figure 6.2: Rendered `gpregressor` HTML report.

7 SPFPPartitioner: Configuration & API

This chapter documents the SPFPPartitioner module as shipped in GenForge: its public constructor, methods, expected inputs/outputs, and how to integrate the produced feature partitions (*findex*) with gpclassifier and gpregressor.

7.1 Overview

Goal. SPFPPartitioner (Semantic Preserving Feature Partitioning) groups the D input features into P disjoint (or near-disjoint) index sets, each intended to feed one GP population. The grouping is driven by entropy-based relevance to the label and by pairwise interaction/redundancy controls, following the exact mathematics in Chapter 4.

Inputs. Provide `numpy.ndarray` arrays:

- `X`: shape (N, D); features are internally standardised (zero mean, unit variance) with the built-in `StandardScaler`.
- `y`: shape $(N,)$; labels are internally encoded by the built-in `LabelEncoder` (classification) or used as-is (regression targets).

Outputs. After fitting, SPFPPartitioner exposes `partitions_` (list of P lists of feature indices) and optional diagnostics (per-feature MI/correlation tables, group entropies, timings).

7.2 Constructor Parameters

Remark 7.1. All MI-related terms are normalised by $H(y)$, and correlation uses the absolute Pearson coefficient on the *continuous, standardised* `X` against the encoded `y`, exactly as in the reference script.

Table 7.1: Public constructor of `SPFPPartitioner`. All defaults mirror the reference implementation.

Parameter	Default	Meaning
`n_groups`	`5`	Number of groups P to construct.
`n_bins`	`100`	Equal-distance discretisation bins for entropy ops.
`objective`	`"micor"`	Primary score in gated branch: `"mi"`, `"cor"`, or `"micor"`.
`criterion_add`	`0.999`	α in forward stopping rule (Section 4.3).
`criterion_rem`	`0.999`	β in optional backward pruning (Section 4.4).
`growth_magnitude`	`1.01`	γ cap during growth-gated search.
`perfs`	`0.10`	π: minimum ratio of D per group.
`remp`	`0.60`	ρ: random withholding rate between groups.
`override`	`False`	If `True`, pick by composite objective directly.
`backward`	`False`	Enable pruning after forward selection.
`floor_decimals`	`4`	Decimal precision for threshold floors.
`random_state`	`None`	Seed for reproducibility.
`verbose`	`1`	Verbosity level (e.g., progress prints).

7.3 Public Methods

Some implementations also expose `get_params()` / `set_params(**kw)` for convenience; consult the class docstring in `spfp_partition.py`.

7.4 Minimal Usage

Listing 7.1: Compute feature partitions with `SPFPPartitioner`.

```
from genforge.spfp.spfp_partition import SPFPPartitioner
import numpy as np

# Load arrays
X = np.load("X.npy") # shape (N, D)
y = np.load("y.npy") # shape (N,)

spfp = SPFPPartitioner(
    n_groups=5,
```

Table 7.2: Core methods of `SPFPPartitioner`.

Signature	Description
`fit(X, y)`	Runs standardisation/encoding, per-feature scoring, group-wise forward selection (and optional backward pruning), then inter-group withholding. Populates `partitions_` and diagnostics.
`partition()`	Builds groups using the statistics prepared by `fit`. Typically called internally by `fit`; exposed for advanced workflows.
`fit_transform(X, y)`	Convenience wrapper: `fit` then return a lightweight result (e.g., `partitions_`).
`get_partitions()`	Returns the list of groups: `[[idx...], [idx...], ...]`.
`transform(X)`	Optional helper to select columns per group; returns a list of views or a dict `{group_id: X[:, idx]}`.
`get_diagnostics()`	Returns a dictionary of MI/correlation tables and group entropy summaries, useful for reporting.
`get_binning_edges()`	Returns the per-feature discretisation edges used by the joint histogram.

```
    n_bins=100,
    objective="micor", # "mi" | "cor" | "micor"
    criterion_add=0.999,
    criterion_rem=0.999,
    growth_magnitude=1.01,
    perfs=0.10,
    remp=0.60,
    override=False,
    backward=False,
    random_state=13,
    verbose=1
)

spfp.fit(X, y)
findex = spfp.get_partitions() # list of lists with feature indices

for g, idx in enumerate(findex):
    print(f"Group {g}: {idx}")
```

Diagnostics (optional).

Listing 7.2: Inspect SPFP diagnostics.

```
diag = spfp.get_diagnostics()
# e.g., per-feature MI/correlation table, group entropies, timings:
mi_table = diag.get("feature_table") # implementation-dependent
group_info = diag.get("group_summaries")
print("Keys:", list(diag.keys()))
```

7.5 Wiring Partitions to GP

Use the produced `findex` to constrain populations in your GP run:

Listing 7.3: Using SPFP partitions with `gpclassifier` (analogous for `gpregressor`).

```
from genforge.gpclassifier import gpclassifier
from genforge.spfp.spfp_partition import SPFPPartitioner
import numpy as np

X_train = np.load("X_train.npy")
y_train = np.load("y_train.npy")
X_val = np.load("X_val.npy")
y_val = np.load("y_val.npy")
X_test = np.load("X_test.npy")
y_test = np.load("y_test.npy")

# Build 3 groups for 3 populations
spfp = SPFPPartitioner(n_groups=3, objective="micor", random_state=7)
spfp.fit(X_train, y_train)
findex = spfp.get_partitions()

params = {
    "userdata_name": "clf_with_spfp",
    "userdata_xtrain": X_train, "userdata_ytrain": y_train,
    "userdata_xval": X_val, "userdata_yval": y_val,
    "userdata_xtest": X_test, "userdata_ytest": y_test,
    "userdata_pop_idx": findex, # <- map groups to populations
    "runcontrol_generations": 50,
    "runcontrol_pop_size": 64,
```

```
    "functions_name": [["times","minus","plus","divide"]],
    "gene_max_genes": [5], "tree_max_depth": [10],
    "selection_elite_fraction": [0.05],
    "selection_elite_fraction_ensemble": [0.05],
    "selection_p_ensemble": [0.5],
    "runcontrol_plotfitness": True,
    "runcontrol_plotrankbest": True,
    "runcontrol_plotsavefig": True,
}

gp = gpclassifier.initialize(**params)
gp.evolve()
gp.report(ensemble_row=0)
```

Remark 7.2. Ensure the number of groups equals the number of populations implied by your GP configuration; otherwise, some populations may receive empty feature sets.

7.6 Notes & Gotchas

- **Scaling and encoding.** `SPFPPartitioner` applies its own feature standardisation and label encoding; pass raw `X`, `y` to it.
- **Bin edges.** Training-set binning edges are re-used consistently for any test-time diagnostics you request, to keep entropies comparable.
- **Objective choice.** `objective` only affects a specific selection branch (when coverage thresholds are met but the minimum size is not). Otherwise, growth-gated search or override governs selection.
- **Reproducibility.** Set `random_state` to stabilise stream ordering and the inter-group withholding step.

8 Plots & HTML Reports

GenForge ships with lightweight plotting utilities and an HTML report mechanism to inspect the final ensemble and its member programs. This chapter explains how to enable/save plots, what each figure means, and how to generate the HTML report from gpclassifier and gpregressor.

8.1 Enabling and Saving Plots

Both gpclassifier and gpregressor support the following run-control switches (names match the code in gp_class.py and gp_regress.py). If a key is not present for your estimator, it is silently ignored.

Table 8.1: Common plot flags and output controls.

Parameter	Meaning	Typical values
runcontrol_plotfitness	Enable Fitness vs. Generation plot	True/False
runcontrol_plotrankbest	Enable RankBest vs. Generation plot	True/False
runcontrol_plotrankall	Enable RankAll vs. Generation plot	True/False
runcontrol_plotsavefig	Save figures to disk	True/False
runcontrol_plotformat	Image formats to save	e.g., ["png","pdf"]
runcontrol_plotfolder	Output folder for plots	e.g., "./plots/"
runcontrol_plotbackend	Matplotlib backend (regressor supports)	"Agg", "TkAgg"
runcontrol_plotlive	Live drawing during run (regressor)	True/False

Example configuration snippet (classification) activating all three plots and saving to ./plots/ in png and pdf:

Listing 8.1: Plot configuration example for gpclassifier.

```
params.update({
    "runcontrol_plotfitness": True,
```

```
    "runcontrol_plotrankbest": True,
    "runcontrol_plotrankall": True,
    "runcontrol_plotsavefig": True,
    "runcontrol_plotformat": ["png", "pdf"],
    "runcontrol_plotfolder": "./plots/"
})
```

The same flags apply to `gpregressor`; additionally, the regressor honors `runcontrol_plotbackend` and `runcontrol_plotlive` if present.

8.2 Fitness vs. Generation

The Fitness curve tracks the best (or aggregate) fitness value per generation. For minimisation runs (the default, `runcontrol_minimisation` = `True`), downward is better; for maximisation, upward improves.

Tips. Sudden jumps typically indicate successful crossover/mutation; stagnation may suggest increasing `runcontrol_pop_size`, relaxing constraints, or injecting diversity.

8.3 RankBest vs. Generation

RankBest reports the rank (1 is best) of the best individual across generations within each population or globally (implementation-dependent). It is a compact progress signal that is less sensitive to fitness scale.

8.4 RankAll vs. Generation

RankAll visualises the distribution of ranks over time (e.g., median and spread). It complements RankBest by revealing whether the whole population improves or only a few elites do.

8.5 HTML Ensemble Report

After evolution, call `report` to emit an HTML page that summarises the chosen ensemble row (e.g., best on the validation split), its mixing weights, and member programs.

Listing 8.2: Emit an HTML report from a trained run.

```
# Classification
```

```
gp = gpclassifier.initialize(**params)
gp.evolve()
gp.report(ensemble_row=0) # writes an HTML file for row 0

# Regression
gr = gpregressor.initialize(**params)
gr.evolve()
gr.report(ensemble_row=0) # same idea
```

What is included. The exact layout depends on your version, but typically contains:

- **Ensemble mixing weights** (including bias term), mapping population outputs to the final prediction.
- **Member programs** (trees) with human-readable expressions.
- **Per-split metrics** (training/validation/test).
- **Run metadata** (seeds, wall time, configuration summary).

Location and naming. By default, reports are written alongside run artifacts (plots/logs). If your build has a configurable output root, set it via the run-control keys used in your environment (e.g., `runcontrol_plotfolder` for figures).

8.6 Interpreting the Ensemble

Mixing weights. Large-magnitude weights highlight populations contributing most to the final prediction. If you use `SPFPPartitioner`, you can attribute contributions back to feature groups by mapping each population to its *findex*.

Program inspection. Member trees often reveal structure such as repeated sub-expressions, protected divisions, or domain-specific interactions (e.g., `times`, `minus`). Use these clues to refine the function set or add domain constraints.

8.7 Assets to Include for This Chapter

Remark 8.1 (Asset checklist)**.** Please export and place the following files under `Figure/` (or update paths accordingly); then recompile to populate Figures 8.1 to 8.4.

1. `fig_fitness_vs_generation.png`: Fitness plot (Figure 8.1).
2. `fig_rankbest_vs_generation.png`: RankBest plot (Figure 8.2).
3. `fig_rankall_vs_generation.png`: RankAll plot (Figure 8.3).
4. `fig_html_report_screenshot.png`: Screenshot of a generated HTML report (Figure 8.4)—capture the ensemble weights and a couple of member trees.

8.8 Troubleshooting

- **Figures do not appear.** Ensure `runcontrol_plotsavefig` = `True`, the folder in `runcontrol_plotfolder` exists, and your process has write permission.
- **Blank plots on a headless server.** Set `runcontrol_plotbackend` = `"Agg"` (supported by the regressor) and disable live drawing.
- **HTML report not created.** Confirm you called `report` after training and that the `ensemble_row` you requested exists.
- **Stale ToC/LoF/LoT entries.** Compile twice. If labels still show `??`, verify file paths in `\graphicspath` and figure filenames.

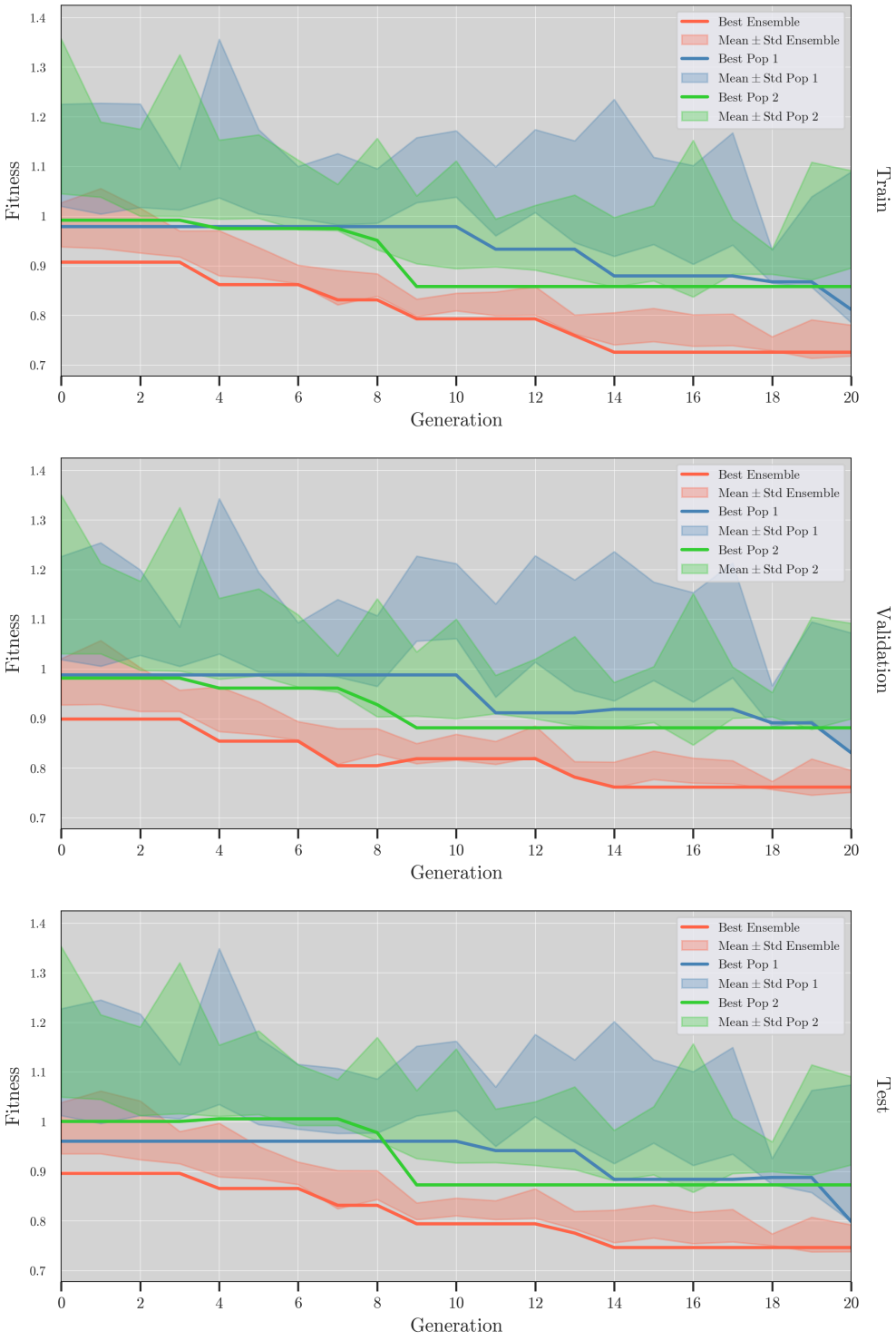


Figure 8.1: Fitness vs. Generation for a representative run.

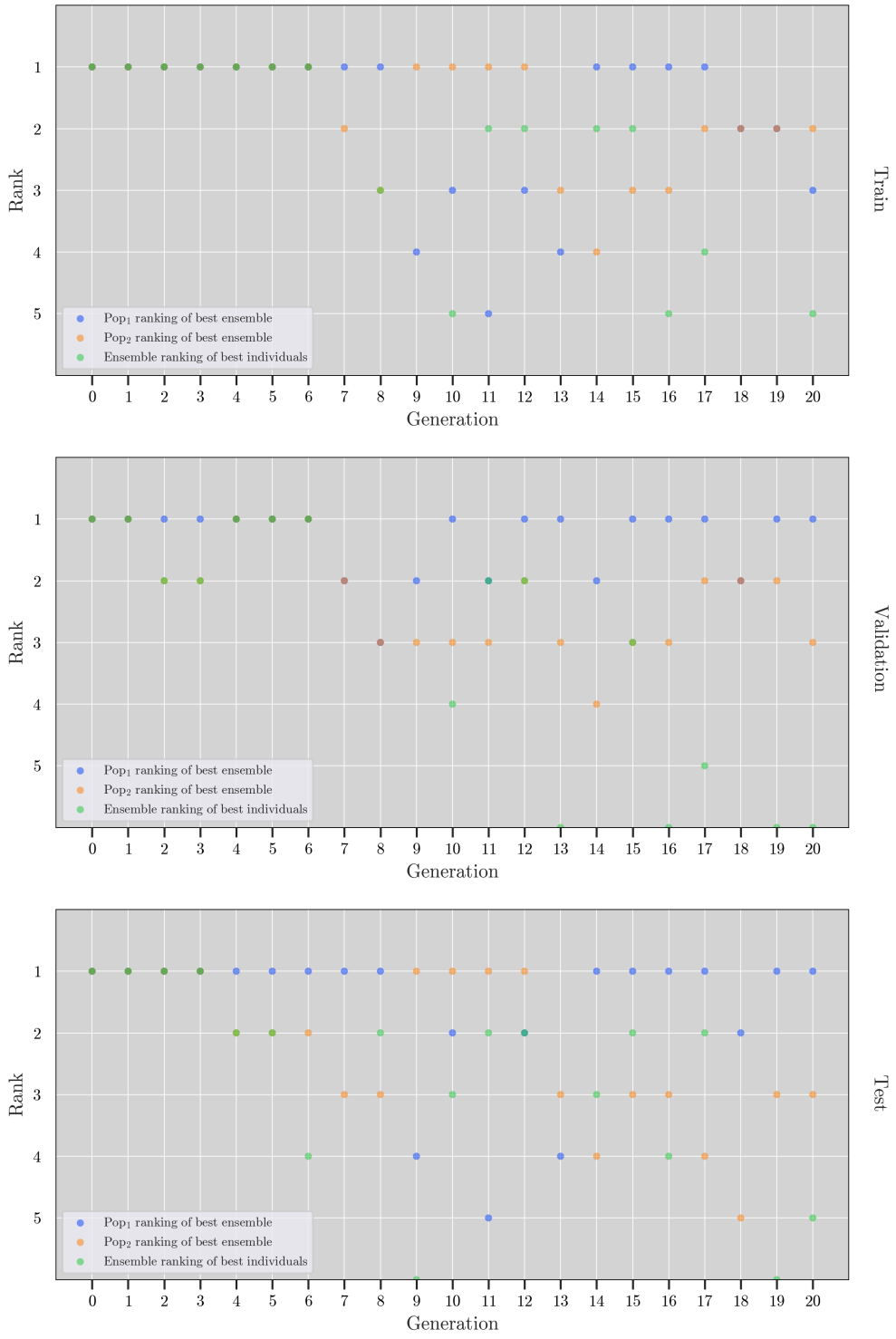


Figure 8.2: RankBest vs. Generation (lower is better).

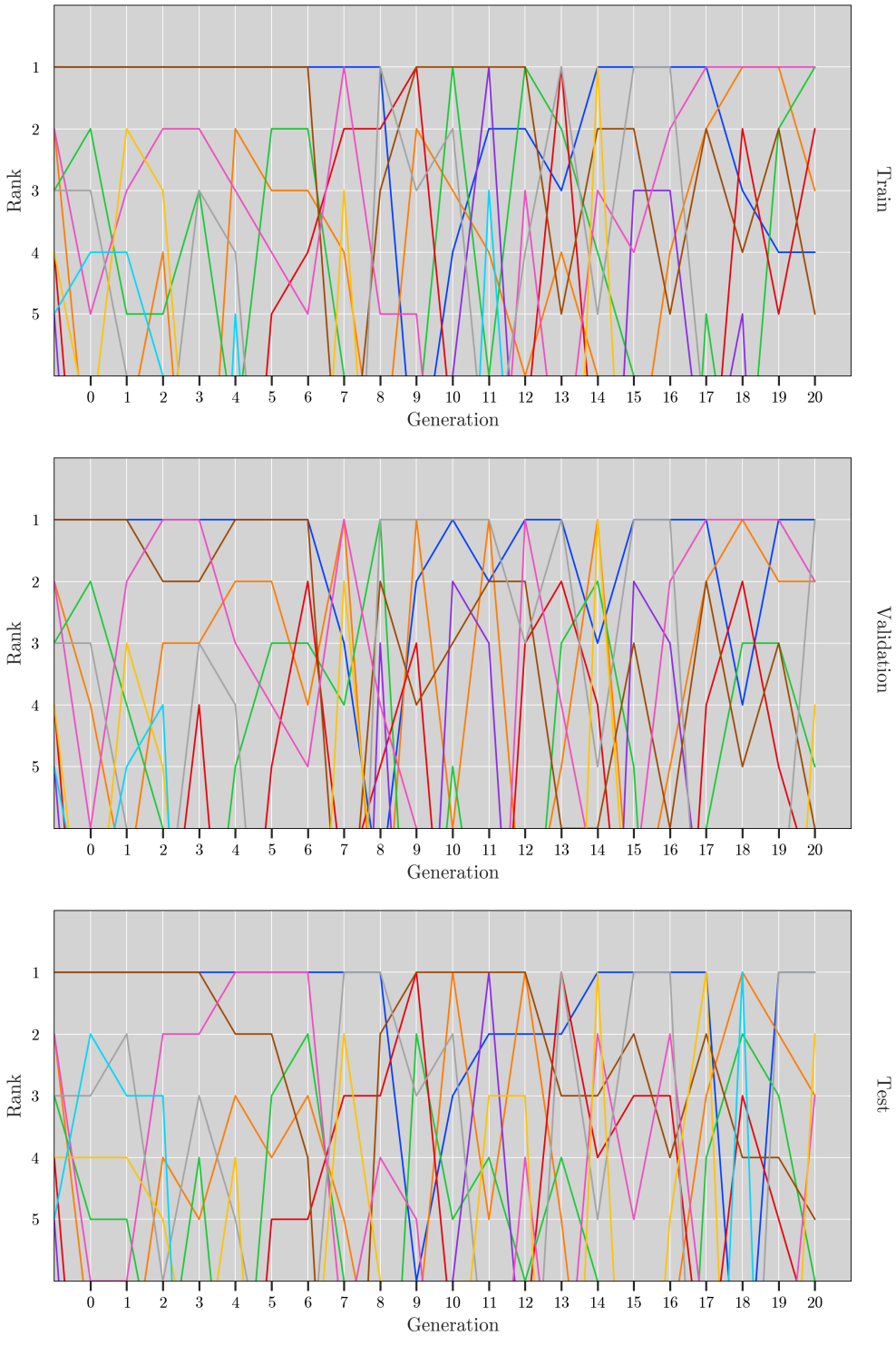


Figure 8.3: RankAll vs. Generation showing median and dispersion.

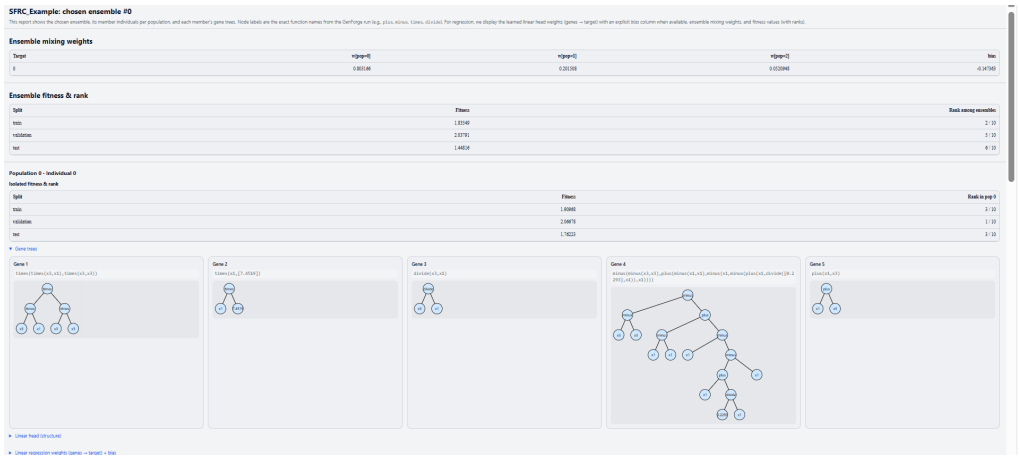


Figure 8.4: HTML report (example) summarising an ensemble row.

9 Reproducibility & Experiment Management

This chapter outlines practical steps to make GenForge runs repeatable and easy to manage at scale: seeding, controlling parallelism and BLAS threads, keeping data splits consistent, organising artifacts, and comparing runs.

9.1 Goals and Scope

Reproducibility in evolutionary methods is nuanced: identical seeds and configurations should yield comparable outcomes, but parallelism, floating-point reductions, and BLAS libraries may introduce small deviations. The guidance below focuses on *practical determinism*—repeatable enough for fair comparisons and ablation studies.

9.2 Seeding and Determinism

Python/NumPy. Always seed the Python and NumPy RNGs at process start:

Listing 9.1: Seeding Python and NumPy.

```python
import os, random, numpy as np
os.environ["PYTHONHASHSEED"] = "0" # make hashed iterations stable
random.seed(1337)
np.random.seed(1337)
```

SPFPPartitioner. Set `random_state` in the SPFPPartitioner constructor for a stable partition:

Listing 9.2: Seeding SPFP.

```python
from genforge.spfp.spfp_partition import SPFPPartitioner
spfp = SPFPPartitioner(random_state=1337, n_groups=5, n_bins=100, objective="micor")
```

gpclassifier / gpregressor. If your build exposes a run seed (e.g., `userdata_seed` or similar in your environment), set it in `params`. When no explicit run seed exists, use the global RNG seeds (Listing 9.1) and keep `runcontrol_useparallel` disabled for strict determinism:

Listing 9.3: Minimal deterministic GP run.

```
from genforge.gpclassifier import gpclassifier
params = {
    "userdata_name": "deterministic_demo",
    # "userdata_seed": 1337, # if your build supports one
    "runcontrol_useparallel": False,
    "runcontrol_generations": 10,
    "runcontrol_pop_size": 32,
    "functions_name": [["times","minus","plus","divide"]],
    "gene_max_genes": [5], "tree_max_depth": [10],
    "selection_elite_fraction": [0.05],
    "selection_elite_fraction_ensemble": [0.05],
    "selection_p_ensemble": [0.5],
}
gp = gpclassifier.initialize(**params)
gp.evolve()
```

9.3 Controlling BLAS & Threading

Numeric libraries (OpenBLAS, MKL, BLIS, numexpr) spawn their own threads, which can affect both speed and determinism. Set thread caps *before* importing NumPy/Scipy:

Table 9.1: Useful environment variables for threading control.

Variable	Recommended setting (example)
`OMP_NUM_THREADS`	`1` (strict) or a small integer
`OPENBLAS_NUM_THREADS`	`1`
`MKL_NUM_THREADS`	`1`
`VECLIB_MAXIMUM_THREADS`	`1` (Accelerate, macOS)
`NUMEXPR_NUM_THREADS`	`1`

Example launcher (Linux/macOS):

Listing 9.4: Launch with controlled threads.

```
OMP_NUM_THREADS=1 OPENBLAS_NUM_THREADS=1 MKL_NUM_THREADS=1 \
VECLIB_MAXIMUM_THREADS=1 NUMEXPR_NUM_THREADS=1 \
python run_experiment.py
```

If you rely on parallel fitness evaluation, prefer a *single BLAS thread* with multiple Python workers via `runcontrol_useparallel` and (if available) `runcontrol_n_jobs`. This reduces contention.

9.4 Data Splits and Consistency

Use the same training split for both `SPFPPartitioner` partitioning and GP evolution to avoid leakage. For classification with imbalanced classes, stratify externally when preparing `X_train`/`y_train`. For time series, avoid shuffling across temporal boundaries.

Remark 9.1 (One source of truth for splits)**.** Persist `X_train.npy`, `y_train.npy`, `X_val.npy`, `y_val.npy`, `X_test.npy`, `y_test.npy` and reuse them across all runs. Document the split provenance (random seed, date, filter criteria).

9.5 Configuration Management

Keep run configurations as plain `dict` objects and persist them (JSON/YAML). This ensures you can regenerate the same run later.

Listing 9.5: Save and reload run configuration.

```
import json, pathlib
cfg_path = pathlib.Path("runs/demo_clf/config.json")
cfg_path.parent.mkdir(parents=True, exist_ok=True)
with cfg_path.open("w") as f:
    json.dump(params, f, indent=2, sort_keys=True)

# ... later
with cfg_path.open() as f:
    params2 = json.load(f)
```

Table 9.2: Suggested minimal keys to track per run (extend as needed).

Key	Purpose
`userdata_name`	Human-readable run name
`runcontrol_generations`	Evolution length
`runcontrol_pop_size`	Population size
`runcontrol_useparallel`	Enable/disable parallel evaluation
`functions_name`	Function set per population
`gene_max_genes`, `tree_max_depth`	Structural limits
`selection_elite_fraction`	Elitism fraction
`selection_p_ensemble`	Ensemble selection probability
`userdata_pop_idx`	(Optional) `SPFPPartitioner` *findex* mapping
`random_state`	(Where applicable) run seed

9.6 Run Naming and Artifact Layout

Adopt a predictable folder layout to store plots, HTML reports, and configs. A simple pattern is `runs/<name>/...`:

Listing 9.6: Example artifact tree (customise to taste).

```
runs/
  demo_clf/
    config.json
    plots/
      fitness.png
      rankbest.png
      rankall.png
    reports/
      ensemble_row0.html
    logs/
      stdout.txt
      metrics.csv
```

If your build already writes plots/reports to a default location (e.g., `runcontrol_plotfolder`), point it to `runs/<name>/plots/` for consistency.

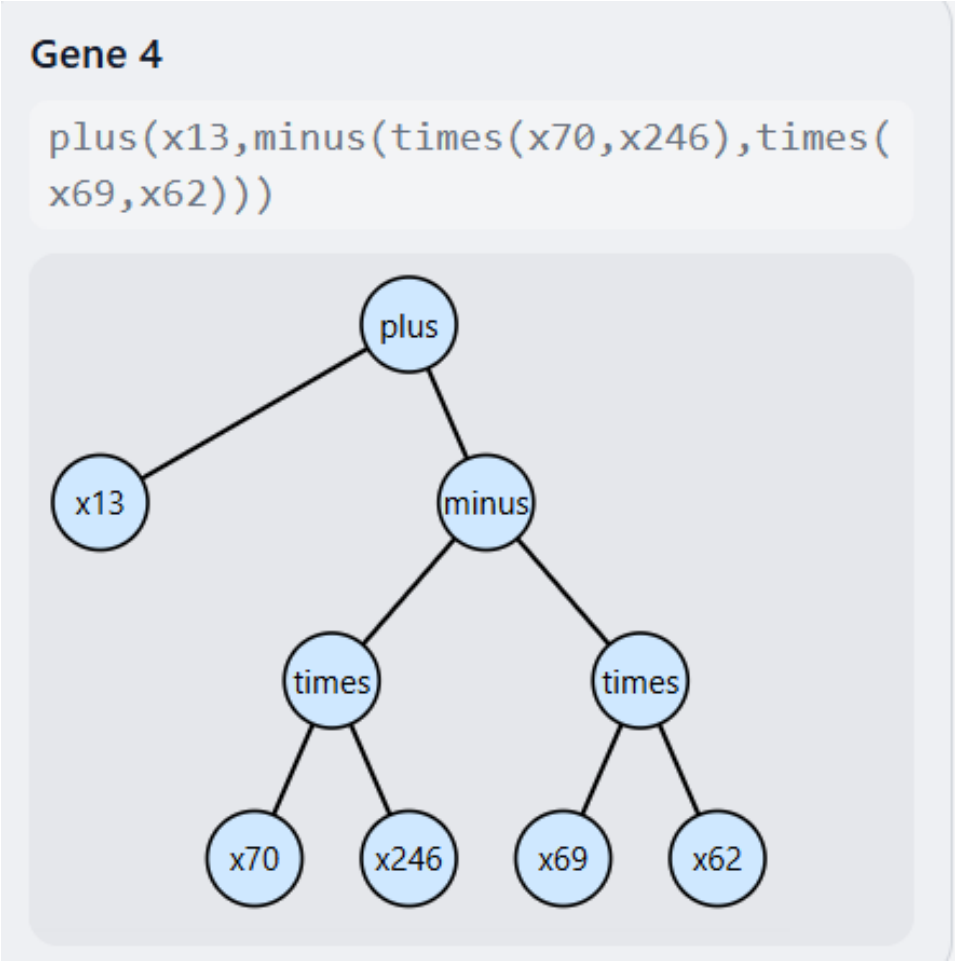


Figure 9.1: Example `runs/` directory with plots, reports, and logs.

9.7 Logging Metrics and Comparing Runs

Log core metrics to a CSV during/after training (validation accuracy, test MAE, wall time). A minimal schema:

Listing 9.7: Append a metric row to CSV.

```
import csv, time, pathlib
csvp = pathlib.Path("runs/demo_clf/metrics.csv")
csvp.parent.mkdir(parents=True, exist_ok=True)
with csvp.open("a", newline="") as f:
```

```
    w = csv.writer(f)
    if f.tell() == 0:
        w.writerow(["run","split","metric","value","step","timestamp"])
    w.writerow(["demo_clf","val","accuracy",0.842,50,int(time.time())])
```

Aggregate multiple runs in a notebook or script to produce comparison tables and stability plots.

9.8 Strict vs. Practical Determinism

- **Strict (bit-wise) determinism:** Disable parallelism (`runcontrol_useparallel` = `False`), pin BLAS threads to 1 (Table 9.1), avoid nondeterministic reductions, and run on the same machine/OS with identical library versions.
- **Practical determinism:** Allow parallel evaluation but keep BLAS threads at 1 and fix seeds. Expect tiny variations in tie-breakers.

9.9 End-to-End Example

A compact script that (i) seeds everything, (ii) computes `SPFPPartitioner` partitions, (iii) runs a GP classifier with those partitions, and (iv) writes plots/reports.

Listing 9.8: Seeded pipeline: SPFP + GPClassifier.

```
import os, random, json, numpy as np, pathlib
from genforge.spfp.spfp_partition import SPFPPartitioner
from genforge.gpclassifier import gpclassifier

# --- seeds and threads
os.environ.update({
    "PYTHONHASHSEED": "0",
    "OMP_NUM_THREADS": "1", "OPENBLAS_NUM_THREADS": "1",
    "MKL_NUM_THREADS": "1", "VECLIB_MAXIMUM_THREADS": "1",
    "NUMEXPR_NUM_THREADS": "1"
})
random.seed(1337); np.random.seed(1337)

# --- data (replace with your arrays)
X_train = np.load("data/X_train.npy")
```

```
y_train = np.load("data/y_train.npy")
X_val = np.load("data/X_val.npy")
y_val = np.load("data/y_val.npy")
X_test = np.load("data/X_test.npy")
y_test = np.load("data/y_test.npy")

# --- SPFP partitions
spfp = SPFPPartitioner(n_groups=4, n_bins=100, objective="micor",
                    perfs=0.10, remp=0.60, random_state=1337, verbose=1)
spfp.fit(X_train, y_train)
findex = spfp.get_partitions() # [[feature indices per group], ...]

# --- GP configuration
params = {
    "userdata_name": "spfp_gp_demo",
    "userdata_xtrain": X_train, "userdata_ytrain": y_train,
    "userdata_xval": X_val, "userdata_yval": y_val,
    "userdata_xtest": X_test, "userdata_ytest": y_test,
    "userdata_pop_idx": findex, # wire features to populations
    "runcontrol_generations": 30,
    "runcontrol_pop_size": 48,
    "runcontrol_useparallel": False, # stricter determinism
    "runcontrol_plotfitness": True,
    "runcontrol_plotrankbest": True,
    "runcontrol_plotsavefig": True,
    "runcontrol_plotfolder": "runs/spfp_gp_demo/plots/",
    "runcontrol_plotformat": ["png"],
    "functions_name": [["times","minus","plus","divide"]],
    "gene_max_genes": [5], "tree_max_depth": [10],
    "selection_elite_fraction": [0.05],
    "selection_elite_fraction_ensemble": [0.05],
    "selection_p_ensemble": [0.5],
}
runroot = pathlib.Path("runs/spfp_gp_demo")
(runroot / "plots").mkdir(parents=True, exist_ok=True)
with (runroot / "config.json").open("w") as f:
    json.dump(params, f, indent=2)

```

```
# --- train and report
gp = gpclassifier.initialize(**params)
gp.evolve()
gp.report(ensemble_row=0)
```

9.10 Troubleshooting

- **Different results across machines.** Pin versions in `requirements.txt` or `pyproject.toml`; align BLAS libraries and thread settings (Table 9.1).
- **Plots missing.** Ensure `runcontrol_plotsavefig` = `True`, and `runcontrol_plotfolder` exists and is writable.
- **HTML not generated.** Call `report(ensemble_row=...)` after training and verify the requested row index exists.
- **SPFP partitions change run-to-run.** Set `random_state` in SPFPPartitioner and reuse the same training split.

10 Advanced Usage & Extensibility

This chapter collects power–user patterns and extension points for `GenForge`: custom primitive function sets, feature–group wiring via `SPFPPartitioner`, constants configuration, selection/ensemble controls, parallel evaluation, and practical performance tips.

10.1 Wiring SPFP Partitions to Multi-Population GP

Given P feature groups produced by `SPFPPartitioner` (see Chapter 4), map them to GP populations by passing the per–group index lists through `userdata_pop_idx`. The number of populations equals the number of groups.

Listing 10.1: Using SPFP partitions with a multi-population run.

```
from genforge.spfp.spfp_partition import SPFPPartitioner
from genforge.gpclassifier import gpclassifier
import numpy as np

# X: (N, D), y: (N,)
X, y = np.load("X.npy"), np.load("y.npy")

# 1) Build P disjoint (or partially overlapping) groups
spfp = SPFPPartitioner(n_groups=5, n_bins=100, objective="micor",
                    perfs=0.10, remp=0.60, random_state=13, verbose=1)
spfp.fit(X, y)
groups = spfp.get_partitions() # e.g., [[2,7,5,...], [1,3,...], ...]

# 2) Hand groups to GP via userdata_pop_idx
params = {
    "userdata_name": "adv_demo",
    "userdata_xtrain": X, "userdata_ytrain": y,
    "userdata_xval": X, "userdata_yval": y, # toy split for demo
```

```
    "userdata_xtest": X, "userdata_ytest": y,
    "userdata_pop_idx": groups, # <- key line
    "runcontrol_generations": 30,
    "runcontrol_pop_size": 64,
    "runcontrol_useparallel": True,
    "runcontrol_n_jobs": 8,
    "functions_name": [["times","minus","plus","divide"]]
}

gp = gpclassifier.initialize(**params)
gp.evolve()
gp.report(ensemble_row=0)
```

Notes.

- Ensure `userdata_pop_idx` has length P (number of populations).
- If you omit `userdata_pop_idx`, each population can use all features.
- You may reuse the same SPFP partition across runs for fair comparisons.

10.2 Custom Primitive Function Sets

10.2.1 Selecting built-in tokens

Specify per–population tokens in `functions_name`:

Listing 10.2: Per-population primitive sets.

```
params["functions_name"] = [
    ["times","minus","plus","divide"], # pop 0
    ["times","minus","plus","divide"], # pop 1
    ["times","minus","plus"], # pop 2
    ["times","plus"], # pop 3
]
```

The tokens `"times"`, `"minus"`, `"plus"` are always available. If your build includes protected division, `"divide"` performs safe division (avoid NaNs/inf). Additional tokens included in your local fork/build can be added the same way.

10.2.2 Passing custom callables

You can inject Python callables via `functions_function`. Each callable must accept `numpy` arrays and be *elementwise* safe.

Listing 10.3: Augmenting with custom protected ops.

```
import numpy as np

def safe_div(a, b, eps=1e-12):
    return a / (np.where(np.abs(b) < eps, np.sign(b)*(eps), b))

def safe_log(a, eps=1e-12):
    return np.log(np.where(a > eps, a, eps))

params["functions_name"] = [["times","minus","plus"]] # tokens
params["functions_function"] = [[safe_div, safe_log]] # extra callables
```

Best practices.

- Make all custom ops *broadcasting*-compatible on `numpy` arrays.
- Guard against domain errors (e.g., negative input to `log`, zero denominator in `divide`).
- Keep the set compact; too many primitives can slow convergence.

10.3 Constants & Terminals

Random constants (ERCs) are governed by `const_*` keys. The exact knobs exposed by your build include (not exhaustive):

- `const_p_ERC`: probability of injecting an ERC terminal,
- `const_min`, `const_max`: numeric range (if provided),
- `const_range`, `const_about`: alternative range controls,
- `const_p_int`: bias toward integer ERCs,
- `const_num_dec_places`: decimal precision for ERCs.

Listing 10.4: Tuning ERC behaviour.

```
params.update({
    "const_p_ERC": 0.10,
    "const_min": -5.0, "const_max": 5.0,
    "const_p_int": 0.20,
    "const_num_dec_places": 3
})
```

Feature terminals (input variables) are wired automatically; ERCs supplement them to improve expressivity.

10.4 Tree & Gene Constraints

Use `gene_*` and `tree_*` to bound representation size and depth:

Listing 10.5: Constraining structure complexity.

```
params.update({
    "gene_max_genes": [5], # per-pop gene cap
    "tree_max_depth": [10], # per-pop tree depth cap
    # "gene_min_genes": [...], # if exposed in your build
    # "tree_min_depth": [...],
})
```

Smaller caps typically improve runtime and generalisation; increase gradually if underfitting.

10.5 Selection & Ensemble Mixing

Key controls reflected in your examples and sources:

- `selection_elite_fraction`: per–population elitism rate.
- `selection_elite_fraction_ensemble`: ensemble elitism rate.
- `selection_p_ensemble`: probability of participating in the ensemble.

The linear mixing head is trained with softmax–style settings (check your build for the available knobs; typical names include `softmax_*`):

Listing 10.6: Ensemble participation and softmax training.

```
params.update({
    "selection_elite_fraction": [0.05],
    "selection_elite_fraction_ensemble": [0.05],
    "selection_p_ensemble": [0.50],

    # Examples (adjust to your build's exact names/defaults):
    "softmax_epochs": [50],
    "softmax_lr": [1e-2],
    "softmax_batch_size": [64]
})
```

Use `gp.report(ensemble_row=...)` to write an HTML snapshot (trees, mixing weights, metrics) for a chosen ensemble member.

10.6 Parallel Fitness Evaluation

GenForge supports CPU-side parallel evaluation. Typical flags (see your examples):

Listing 10.7: Parallel settings.

```
params.update({
    "runcontrol_useparallel": True,
    "runcontrol_n_jobs": 8, # number of worker processes/threads
    "runcontrol_batch_job": 32 # granularity per worker (if available)
})
```

Tips.

- Balance `runcontrol_batch_job` to reduce overhead on small trees.
- Fix seeds for reproducibility when running parallel (see below).

10.7 Saving, Logging, & Reproducibility

- Plots: enable `runcontrol_plotfitness`, `runcontrol_plotrankbest`, `runcontrol_plotrankall` and set `runcontrol_plotsavefig`.

- Plot destination: `runcontrol_plotfolder` (if exposed) controls output directory; otherwise files are saved under the run's folder.
- Snapshots: `runcontrol_savefreq` (when present) controls periodic checkpointing of run state.
- Naming: `userdata_name` tags outputs and reports.

For reproducibility, fix `random_state` (SPFP), your GP initialisation seeds (if exposed), and pin dependency versions.

10.8 Performance Tuning Checklist

(a) Start small: reduce `runcontrol_generations`, `runcontrol_pop_size`, `tree_max_depth` to sanity–check the pipeline.

(b) Use `SPFPPartitioner` to reduce the effective input dimensionality per population; tune `n_groups` and `perfs`.

(c) Keep the primitive set lean (`functions_name`); add operators only when they demonstrably help.

(d) Enable parallel evaluation once correctness is verified.

(e) Inspect HTML reports to prune unhelpful primitives or raise depth caps.

10.9 Minimal End-to-End Template

Listing 10.8: Skeleton script for advanced runs.

```
import numpy as np
from genforge.spfp.spfp_partition import SPFPPartitioner
from genforge.gpregressor import gpregressor

# 0) Load data
X_train = np.load("X_train.npy")
y_train = np.load("y_train.npy")
X_val = np.load("X_val.npy")
y_val = np.load("y_val.npy")
X_test = np.load("X_test.npy")
```

```
y_test = np.load("y_test.npy")

# 1) Optional: SPFP partitioning
spfp = SPFPPartitioner(n_groups=4, n_bins=100, objective="mi",
                    perfs=0.10, remp=0.60, random_state=19, verbose=1)
spfp.fit(X_train, y_train)
groups = spfp.get_partitions()

# 2) GP configuration
params = {
    "userdata_name": "adv_reg",
    "userdata_xtrain": X_train, "userdata_ytrain": y_train,
    "userdata_xval": X_val, "userdata_yval": y_val,
    "userdata_xtest": X_test, "userdata_ytest": y_test,
    "userdata_pop_idx": groups,

    "runcontrol_generations": 50,
    "runcontrol_pop_size": 64,
    "runcontrol_useparallel": True,
    "runcontrol_n_jobs": 8,
    "runcontrol_plotfitness": True,
    "runcontrol_plotrankbest": True,
    "runcontrol_plotsavefig": True,

    "functions_name": [["times","minus","plus","divide"]],
    "gene_max_genes": [5],
    "tree_max_depth": [10],

    "selection_elite_fraction": [0.05],
    "selection_elite_fraction_ensemble": [0.05],
    "selection_p_ensemble": [0.50],

    # softmax_* if present in your build
}

gp = gpregressor.initialize(**params)
gp.evolve()
gp.report(ensemble_row=0)
```

10.10 Troubleshooting Extensions

- **Custom op returns NaN/inf.** Add guards (e.g., `safe_div`, `safe_log`) and validate on random arrays before registering.
- **Very slow runs.** Reduce `tree_max_depth`, prune `functions_name`, and raise `runcontrol_batch_job`.
- **Ensemble not improving.** Increase `selection_p_ensemble` slightly, tune `softmax_*` epochs/lr.
- **Partitions too small/large.** Adjust SPFPPartitioner `perfs` and `n_groups`; consider `objective="micor"` to balance MI and correlation.