hyperparameter_hunter Documentation

Release 3.0.0

Hunter McGushion

Aug 06, 2019

CONTENTS

CHAPTER

ONE

WHY USE HYPERPARAMETERHUNTER?

This section provides an overview of the mission and primary uses of HyperparameterHunter, as well as some of its main features.

1.1 TL;DR

- HyperparameterHunter saves your Experiments to provide:
  1) Enhanced, long-term hyperparameter optimization; and
  2) Improved awareness of what you've done, what works, and what you should try next

1.2 What is HyperparameterHunter?

- Don't think of HyperparameterHunter as a new machine learning tool; its a toolbox
  - There are tons of excellent machine learning libraries. The problem is keeping track of them all
  - Impractical to keep track of which libraries work, which hyperparameters are best for whichever algorithms, and how your experiment was set up
  - Let HyperparameterHunter organize your tools for you, while you focus on using the best tool for the job
  - Stop wasting time debating between a screwdriver and a wrench, when you're staring at a nail
- Not a new thing to try alongside other algorithms. Its a new way of doing the things you already do
  - Keep using the libraries/algorithms you know and love, just tell HyperparameterHunter about them
- Provides a simple wrapper for executing machine learning algorithms
  - Automatically saves the testing conditions/hyperparameters, results, predictions, and more
  - Test and evaluate wide range of algorithms from many different libraries in a unified format

1.3 Features

- Stop worrying about keeping track of hyperparameters, scores, or re-running the same Experiments
- See records of all your Experiments: from birds-eye-view leaderboards, to individual result files
- Supercharge informed hyperparameter optimization by allowing it to use saved Experiments
  - No need to hold HyperparameterHunter's hand while it tries to find the Experiment you ran months ago

  - It automatically reads your Experiment files to find the ones that fit, and it learns from them
- Eliminate boilerplate code for cross-validation loops, predicting, and scoring
- Have predictions ready to go when its time for ensembling, meta-learning, and finalizing your models

CHAPTER

TWO

INSTALLATION

This section explains how to install HyperparameterHunter.

For the latest stable release, execute:

```
pip install hyperparameter_hunter
```

For the bleeding-edge version, execute:

```
pip install git+https://github.com/HunterMcGushion/hyperparameter_hunter.git
```

2.1 Dependencies

- Dill
- NumPy
- Pandas
- SciPy
- Scikit-Learn
- Scikit-Optimize
- SimpleJSON

CHAPTER

THREE

QUICK START

This section provides a jumping-off point for using HyperparameterHunter's main features.

3.1 Set Up an Environment

```
from hyperparameter_hunter import Environment, CVExperiment
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import StratifiedKFold
from xgboost import XGBClassifier

data = load_breast_cancer
df = pd.DataFrame(data=data.data, columns=data.feature_names)
df["target"] = data.target

env = Environment(
        train_dataset=df,
        results_path="path/to/results/directory",
        metrics=["roc_auc_score"],
        cv_type=StratifiedKFold,
        cv_params=dict(n_splits=5, shuffle=2, random_state=32)
)
```

3.1.1 Individual Experimentation

```
experiment = CVExperiment(
    model_initializer=XGBClassifier,
    model_init_params=dict(objective="reg:linear", max_depth=3, subsample=0.5)
)
```

3.1.2 Hyperparameter Optimization

```
from hyperparameter_hunter import BayesianOptPro, Real, Integer, Categorical

optimizer = BayesianOptPro(iterations=10, read_experiments=True)

optimizer.forge_experiment(
    model_initializer=XGBClassifier,
```

(continues on next page)

(continued from previous page)

```
    model_init_params=dict(
        n_estimators=200,
        subsample=0.5,
        max_depth=Integer(2, 20),
        learning_rate=Real(0.0001, 0.5),
        booster=Categorical(["gbtree", "gblinear", "dart"]),
    )
)

optimizer.go()
```

Plenty of examples for different libraries, and algorithms, as well as more advanced HyperparameterHunter features can be found in the examples directory.

CHAPTER

FOUR

HYPERPARAMETERHUNTER API ESSENTIALS

This section exposes the API for all the HyperparameterHunter functionality that will be necessary for most users.

4.1 Environment

class hyperparameter_hunter.environment.**Environment**(*train_dataset*, *environment_params_path=None*, ***, *results_path=None*, *metrics=None*, *holdout_dataset=None*, *test_dataset=None*, *target_column=None*, *id_column=None*, *do_predict_proba=None*, *prediction_formatter=None*, *metrics_params=None*, *cv_type=None*, *runs=None*, *global_random_seed=None*, *random_seeds=None*, *random_seed_bounds=None*, *cv_params=None*, *verbose=None*, *file_blacklist=None*, *reporting_params=None*, *to_csv_params=None*, *do_full_save=None*, *experiment_callbacks=None*, *experiment_recorders=None*, *save_transformed_metrics=None*)

Bases: object

Class to organize the parameters that allow Experiments/OptPros to be fairly compared

Environment is the collective starting point for all of HyperparameterHunter's biggest and best toys: Experiments and OptimizationProtocols. Without an *Environment*, neither of these will work.

The *Environment* is where we declare all the parameters that transcend traditional "hyperparameters". It houses the stuff without which machine learning can't even really start. Specifically, *Environment* cares about 1) The data used for fitting/predicting, 2) The cross-validation scheme used to split the data and fit models; and 3) How to evaluate the predictions made on that data. There are plenty of other goodies documented below, but the absolutely mission-critical parameters concerned with the above tasks are *train_dataset*, *cv_type*, *cv_params*, and *metrics*. Additionally, it's important to provide *results_path*, so Experiment/OptPro results can be saved, which is kind of what HyperparameterHunter is all about

Parameters

train_dataset: Pandas.DataFrame, or str path The training data for the experiment. Will be split into train/holdout data, if applicable, and train/validation data if cross-validation is to be performed. If str, will attempt to read file at path via `pandas.read_csv()`. For more information on which columns will be used during fitting/predicting, see the "Dataset columns" note in the "Notes" section below

environment_params_path: String path, or None, default=None If not None and is valid .json filepath containing an object (dict), the file's contents are treated as the default values for all keys that match any of the below kwargs used to initialize `Environment`

results_path: String path, or None, default=None If valid directory path and the results directory has not yet been created, it will be created here. If this does not end with <ASSETS_DIRNAME>, it will be appended. If <ASSETS_DIRNAME> already exists at this path, new results will also be stored here. If None or invalid, results will not be stored

metrics: Dict, List, or None, default=None Iterable describing the metrics to be recorded, along with a means to compute the value of each metric. Should be of one of the two following forms:

List Form:

- ["<metric name>", "<metric name>", ...]: Where each value is a string that names an attribute in `sklearn.metrics`
- [*Metric*, *Metric*, ...]: Where each value of the list is an instance of `metrics.Metric`
- [(<name>, <metric_function>, [<direction>]), (<*args>), ...]: Where each value of the list is a tuple of arguments that will be used to instantiate a `metrics.Metric`. Arguments given in tuples must be in order expected by `metrics.Metric`: (*name*, *metric_function*, *direction*)

Dict Form:

- {"<metric name>": <metric_function>, ...}: Where each key is a name for the corresponding metric callable, which is used to compute the value of the metric
- {"<metric name>": (<metric_function>, <direction>), ...}: Where each key is a name for the corresponding metric callable and direction, all of which are used to instantiate a `metrics.Metric`
- {"<metric name>": "<sklearn metric name>", ...}: Where each key is a name for the metric, and each value is the name of the attribute in `sklearn.metrics` for which the corresponding key is an alias
- {"<metric name>": None, ...}: Where each key is the name of the attribute in `sklearn.metrics`
- {"<metric name>": *Metric*, ...}: Where each key names an instance of `metrics.Metric`. This is the internally-used format to which all other formats will be converted

Metric callable functions should expect inputs of form (target, prediction), and should return floats. See the documentation of `metrics.Metric` for information regarding expected parameters and types

holdout_dataset: Pandas.DataFrame, callable, str path, or None, default=None If pd.DataFrame, this is the holdout dataset. If callable, expects a function that takes (self.train: DataFrame, self.target_column: str) as input and returns the new (self.train: DataFrame, self.holdout: DataFrame). If str, will attempt to read file at path via `pandas.read_csv()`. Else, there is no holdout set. For more information on which columns will

be used during fitting/predicting, see the "Dataset columns" note in the "Notes" section below

test_dataset: Pandas.DataFrame, str path, or None, default=None The testing data for the experiment. Structure should be identical to that of *train_dataset*, except its *target_column* column can be empty or non-existent, because *test_dataset* predictions will never be evaluated. If str, will attempt to read file at path via `pandas.read_csv()`. For more information on which columns will be used during fitting/predicting, see the "Dataset columns" note in the "Notes" section below

target_column: Str, or list, default='target' If str, denotes the column name in all provided datasets (except test) that contains the target output. If list, should be a list of strs designating multiple target columns. For example, in a multi-class classification dataset like UCI's hand-written digits, *target_column* would be a list containing ten strings. In this example, the *target_column* data would be sparse, with a 1 to signify that a sample is a written example of a digit (0-9). For a working example, see 'hyperparameter_hunter/examples/lib_keras_multi_classification_example.py'

id_column: Str, or None, default=None If not None, str denoting the column name in all provided datasets containing sample IDs

do_predict_proba: Boolean, or int, default=False

- If False, `models.Model.fit()` will call `models.Model.model.predict()`
- If True, it will call `models.Model.model.predict_proba()`, and the values in all columns will be used as the actual prediction values
- If *do_predict_proba* is an int, `models.Model.fit()` will call `models.Model.model.predict_proba()`, as is the case when *do_predict_proba* is True, but the int supplied as *do_predict_proba* declares the column index to use as the actual prediction values
- For example, for a model to call the *predict* method, *do_predict_proba=False* (default). For a model to call the *predict_proba* method, and use all of the class probabilities, *do_predict_proba=True*. To call the *predict_proba* method, and use the class probabilities in the first column, *do_predict_proba=0*. To use the second column (index 1) of the result, *do_predict_proba=1* - This often corresponds to the positive class's probabilities in binary classification problems. To use the third column *do_predict_proba=2*, and so on

prediction_formatter: Callable, or None, default=None If callable, expected to have same signature as `utils.result_utils.format_predictions()`. That is, the callable will receive (raw_predictions: np.array, dataset_df: pd.DataFrame, target_column: str, id_column: str or None) as input and should return a properly formatted prediction DataFrame. The callable uses raw_predictions as the content, dataset_df to provide any id column, and target_column to identify the column in which to place raw_predictions

metrics_params: Dict, or None, default=dict() Dictionary of extra parameters to provide to `metrics.ScoringMixIn.__init__()`. *metrics* must be provided either 1) as an input kwarg to `Environment.__init__()` (see *metrics*), or 2) as a key in *metrics_params*, but not both. An Exception will be raised if both are given, or if neither is given

cv_type: Class or str, default='KFold' The class to define cross-validation splits. If str, it must be an attribute of *sklearn.model_selection._split*, and it must be a cross-validation class that inherits one of the following *sklearn* classes: *BaseCrossValidator*, or *_RepeatedSplits*. Valid str values include 'KFold', and 'RepeatedKFold', although there are many more. It

must implement the following methods: [*__init__*, *split*]. If using a custom class, see the following tested *sklearn* classes for proper implementations: [*KFold*, *StratifiedKFold*, *RepeatedKFold*, *RepeatedStratifiedKFold*]. The arguments provided to `cv_type.__init__()` will be `Environment.cv_params`, which should include the following: ['n_splits' <int>, 'n_repeats' <int> (if applicable)]. `cv_type.split()` will receive the following arguments: [`BaseExperiment.train_input_data`, `BaseExperiment.train_target_data`]

runs: Int, default=1 The number of times to fit a model within each fold to perform multiple-run-averaging with different random seeds

global_random_seed: Int, default=32 The initial random seed used just before generating an Experiment's random_seeds. This ensures consistency for *random_seeds* between Experiments, without having to explicitly provide it here

random_seeds: None, or List, default=None If None, *random_seeds* of the appropriate shape will be created automatically. Else, must be a list of ints of shape (*cv_params['n_repeats']*, *cv_params['n_splits']*, *runs*). If *cv_params* does not have the key *n_repeats* (because standard cross-validation is being used), the value will default to 1. See `experiments.BaseExperiment._random_seed_initializer()` for info on expected shape

random_seed_bounds: List, default=[0, 100000] A list containing two integers: the lower and upper bounds, respectively, for generating an Experiment's random seeds in `experiments.BaseExperiment._random_seed_initializer()`. Generally, leave this kwarg alone

cv_params: dict, or None, default=dict() Parameters provided upon initialization of cv_type. Keys may be any args accepted by `cv_type.__init__()`. Number of fold splits must be provided via "n_splits", and number of repeats (if applicable for *cv_type*) must be provided via "n_repeats"

verbose: Int, boolean, default=3 Verbosity of printing for any experiments performed while this Environment is active

Higher values indicate more frequent logging. Logs are still recorded in the heartbeat file regardless of verbosity level. *verbose* only dictates which logs are visible in the console. The following table illustrates which types of logging messages will be visible with each verbosity level:

Verbosity	Keys/IDs	Final Score	Repetitions*	Folds	Runs*	Run Starts*	Result Files	Other
0								
1	Yes	Yes						
2	Yes	Yes	Yes	Yes				
3	Yes	Yes	Yes	Yes	Yes			
4	Yes	Yes	Yes	Yes	Yes	Yes		

**: If such logging is deemed appropriate with the given cross-validation parameters. In other words, repetition/run logging will only be verbose if Environment was given more than one repetition/run, respectively*

file_blacklist: List of str, or None, or 'ALL', default=None If list of str, the result files named

within are not saved to their respective directory in "<ASSETS_DIRNAME>/Experiments". If None, all result files are saved. If 'ALL', nothing at all will be saved for the Experiments. If the path of the file that initializes an Experiment does not end with a ".py" extension, the Experiment proceeds as if "script_backup" had been added to *file_blacklist*. This means that backup files will not be created for Jupyter notebooks (or any other non-".py" files). For info on acceptable values, see `validate_file_blacklist()`

reporting_params: Dict, default=dict() Parameters passed to initialize `reporting.ReportingHandler`

to_csv_params: Dict, default=dict() Parameters passed to the calls to `pandas.frame.DataFrame.to_csv()` in `recorders`. In particular, this is where an Experiment's final prediction files are saved, so the values here will affect the format of the .csv prediction files. Warning: If *to_csv_params* contains the key "path_or_buf", it will be removed. Otherwise, all items are supplied directly to `to_csv()`, including kwargs it might not be expecting if they are given

do_full_save: None, or callable, default=:func:'utils.result_utils.default_do_full_save'
If callable, expected to take an Experiment's result description dict as input and return a boolean. If None, treated as a callable that returns True. This parameter is used by `recorders.DescriptionRecorder` to determine whether the Experiment result files following the description should also be created. If *do_full_save* returns False, result file-saving is stopped early, and only the description is saved. If *do_full_save* returns True, all files not in *file_blacklist* are saved normally. This allows you to skip creation of an Experiment's predictions, logs, and heartbeats if its score does not meet some threshold you set, for example. *do_full_save* receives the Experiment description dict as input, so for help setting *do_full_save*, just look into one of your Experiment descriptions

experiment_callbacks: 'LambdaCallback', or list of 'LambdaCallback' (optional)
Callbacks injected directly into Experiments, adding new functionality, or customizing existing processes. Should be a `LambdaCallback` or a list of such classes. *LambdaCallback* can be created using `callbacks.bases.lambda_callback()`, which documents the options for creating callbacks. *experiment_callbacks* will be added to the MRO of the executed Experiment class by `experiment_core.ExperimentMeta` at *__call__* time, making *experiment_callbacks* new base classes of the Experiment. See `callbacks.bases.lambda_callback()` for more information. Note that the Experiments conducted by OptPros will still benefit from *experiment_callbacks*. The presence of LambdaCallbacks will affect neither Environment keys, nor Experiment keys. In other words, for the purposes of Experiment matching/recording, all other factors being equal, an Experiment with *experiment_callbacks* is considered identical to an Experiment without, despite whatever custom functionality was added by the LambdaCallbacks

experiment_recorders: List, None, default=None If not None, may be a list whose values are tuples of (<`recorders.BaseRecorder` descendant>, <str result_path>). The result_path str should be a path relative to *results_path* that specifies the directory/file in which the product of the custom recorder should be saved. The contents of *experiment_recorders* will be provided to *recorders.RecorderList* upon completion of an Experiment, and, if the subclassing documentation in *recorders* is followed properly, will create or update a result file for the just-executed Experiment

save_transformed_metrics: Boolean (optional) Declares manner in which a model's predictions should be evaluated through the provided *metrics*, with regard to target data transformations. This setting can be ignored if no transformation of the target variable takes place (either through `FeatureEngineer`, `EngineerStep`, or otherwise).

The default value of *save_transformed_metrics* depends on the dtype of the target data in *train_dataset*. If all target columns are numeric, *save_transformed_metrics'=False, mean-*

ing metric evaluation should use the original/inverted targets and predictions. Else if any target column is non-numeric, 'save_transformed_metrics'=True, meaning evaluation should use the transformed targets and predictions because most metrics require numeric inputs. This is described further in :attr:'save_transformed_metrics. A more descriptive name for this may be "calculate_metrics_using_transformed_predictions", but that's a bit verbose–even by my standards

Other Parameters

cross_validation_type: ...

- Alias for *cv_type* *

cross_validation_params: ...

- Alias for *cv_params* *

metrics_map: ...

- Alias for *metrics* *

reporting_handler_params: ...

- Alias for *reporting_params* *

root_results_path: ...

- Alias for *results_path* *

Notes

Dataset columns: In order to specify the columns to be used by the three dataset kwargs (*train_dataset*, *holdout_dataset*, *test_dataset*) during fitting and predicting, a few attributes can be used. On *Environment* initialization, the columns specified by the following kwargs will be separated from the rest of the dataset during training/predicting: 1) *target_column*, which names the column containing the target output labels for the input data; and 2) *id_column*, which (if given) represents the name of the column that contains identifying information for each data sample, and should otherwise have no relation to the actual data. Additionally, the *feature_selector* kwarg of the descendants of `hyperparameter_hunter.experiments.BaseExperiment` (like `hyperparameter_hunter.experiments.CVExperiment`) is used to filter out columns of the given datasets prior to fitting. See its documentation for more information, but it can effectively be used to remove any columns from the datasets

Overriding default kwargs at *environment_params_path*: If you have any of the above kwargs specified in the .json file at environment_params_path (except environment_params_path, which will be ignored), you can override its value by passing it as a kwarg when initializing `Environment`. The contents at environment_params_path are only used when the matching kwarg supplied at initialization is None. See "/examples/environment_params_path_example.py" for details

The order of precedence for determining the value of each parameter is as follows, with items at the top having the highest priority, and deferring only to the items below if their own value is None:

- 1)kwargs passed directly to `Environment.__init__()` on initialization,
- 2)keys of the file at environment_params_path (if valid .json object),
- 3)keys of `hyperparameter_hunter.environment.Environment.DEFAULT_PARAMS`

do_predict_proba: Because this parameter can be either a boolean or an integer, it is important to explicitly pass booleans rather than truthy or falsey values. Similarly, only pass integers if you intend for the value to be used as a column index. Do not pass *0* to mean *False*, or *1* to mean *True*

Attributes

train_input: DatasetSentinel Sentinel replaced with current train input data during *Model* fitting/predicting. Commonly given in the *model_extra_params* kwargs of `hyperparameter_hunter.experiments.BaseExperiment` or `hyperparameter_hunter.optimization.protocol_core.BaseOptPro.forge_experiment()` for *eval_set*-like hyperparameters. Importantly, the actual value of this Sentinel is determined after performing cross-validation data splitting, and after executing `FeatureEngineer`

train_target: DatasetSentinel Like `train_input`, except for current train target data

validation_input: DatasetSentinel Like `train_input`, except for current validation input data

validation_target: DatasetSentinel Like `train_input`, except for current validation target data

holdout_input: DatasetSentinel Like `train_input`, except for current holdout input data

holdout_target: DatasetSentinel Like `train_input`, except for current holdout target data

Methods

`environment_workflow`(self)	Execute all methods required to validate the environment and run Experiments
`format_result_paths`(self)	Remove paths contained in file_blacklist, and format others to prepare for saving results
`generate_cross_experiment_key`(self)	Generate a key to describe the current Environment's cross-experiment parameters
`initialize_reporting`(self)	Initialize reporting for the Environment and Experiments conducted during its lifetime
`update_custom_environment_params`(self)	Try to update null parameters from environment_params_path, or DEFAULT_PARAMS
`validate_parameters`(self)	Ensure the provided parameters are valid and properly formatted

property save_transformed_metrics

If *save_transformed_metrics* is True, and target transformation does occur, then experiment metrics are calculated using the transformed targets and predictions, which is the form returned directly by a fitted model's *predict* method. For example, if target data is label-encoded, and an `feature_engineering.EngineerStep` is used to one-hot encode the target, then metrics functions will receive the following as input: (one-hot-encoded targets, one-hot-encoded predictions).

Conversely, if *save_transformed_metrics* is False, and target transformation does occur, then experiment metrics are calculated using the inverse of the transformed targets and predictions, which is same form as the original target data. Continuing the example of label-encoded target data, and an `feature_engineering.EngineerStep` to one-hot encode the target, in this case, metrics functions will receive the following as input: (label-encoded targets, label-encoded predictions)

environment_workflow(*self*)

Execute all methods required to validate the environment and run Experiments

validate_parameters(*self*)

Ensure the provided parameters are valid and properly formatted

format_result_paths(*self*)

Remove paths contained in file_blacklist, and format others to prepare for saving results

update_custom_environment_params (*self*)
Try to update null parameters from environment_params_path, or DEFAULT_PARAMS

generate_cross_experiment_key (*self*)
Generate a key to describe the current Environment's cross-experiment parameters

initialize_reporting (*self*)
Initialize reporting for the Environment and Experiments conducted during its lifetime

property train_input
Get a *DatasetSentinel* representing an Experiment's *fold_train_input*

Returns

DatasetSentinel: A *Sentinel* that will be converted to `hyperparameter_hunter.experiments.BaseExperiment.fold_train_input` upon *Model* initialization

property train_target
Get a *DatasetSentinel* representing an Experiment's *fold_train_target*

Returns

DatasetSentinel: A *Sentinel* that will be converted to `hyperparameter_hunter.experiments.BaseExperiment.fold_train_target` upon *Model* initialization

property validation_input
Get a *DatasetSentinel* representing an Experiment's *fold_validation_input*

Returns

DatasetSentinel: A *Sentinel* that will be converted to `hyperparameter_hunter.experiments.BaseExperiment.fold_validation_input` upon *Model* initialization

property validation_target
Get a *DatasetSentinel* representing an Experiment's *fold_validation_target*

Returns

DatasetSentinel: A *Sentinel* that will be converted to `hyperparameter_hunter.experiments.BaseExperiment.fold_validation_target` upon *Model* initialization

property holdout_input
Get a *DatasetSentinel* representing an Experiment's *holdout_input_data*

Returns

DatasetSentinel: A *Sentinel* that will be converted to `hyperparameter_hunter.experiments.BaseExperiment.holdout_input_data` upon *Model* initialization

property holdout_target
Get a *DatasetSentinel* representing an Experiment's *holdout_target_data*

Returns

DatasetSentinel: A *Sentinel* that will be converted to `hyperparameter_hunter.experiments.BaseExperiment.holdout_target_data` upon *Model* initialization

4.2 Experimentation

class hyperparameter_hunter.experiments.**CVExperiment**(*model_initializer*, *model_init_params=None*, *model_extra_params=None*, *feature_engineer=None*, *feature_selector=None*, *notes=None*, *do_raise_repeated=False*, *auto_start=True*, *target_metric=None*, *callbacks=None*)

Bases: hyperparameter_hunter.experiments.BaseCVExperiment

__init__(*self*, *model_initializer*, *model_init_params=None*, *model_extra_params=None*, *feature_engineer=None*, *feature_selector=None*, *notes=None*, *do_raise_repeated=False*, *auto_start=True*, *target_metric=None*, *callbacks=None*)

One-off Experimentation base class

Bare-bones Description: Runs the cross-validation scheme defined by *Environment*, during which 1) Datasets are processed according to *feature_engineer*; 2) Models are built by instantiating *model_initializer* with *model_init_params*; 3) Models are trained on processed data, optionally using parameters from *model_extra_params*; 4) Results are logged and recorded for each fitting period; 5) Descriptions, predictions, results (both averages and individual periods), etc. are saved.

What's the Big Deal? The most important takeaway from the above description is that descriptions/results are THOROUGH and REUSABLE. By thorough, I mean that all of a model's hyperparameters are saved, not just the ones given in *model_init_params*. This may sound odd, but it's important because it makes results reusable during optimization, when you may be using a different set of hyperparameters. It helps with other things like preventing duplicate experiments and ensembling, as well. But the big part is that this transforms hyperparameter optimization from an isolated, throwaway process we can only afford when an ML project is sufficiently "mature" to a process that covers the entire lifespan of a project. No Experiment is forgotten or wasted. Optimization is automatically given the data it needs to succeed by drawing on all your past Experiments and optimization rounds.

The Experiment has three primary missions: 1. Act as scaffold for organizing ML Experimentation and optimization 2. Record Experiment descriptions and results 3. Eliminate lots of repetitive/error-prone boilerplate code

Providing a scaffold for the entire ML process is critical because without a standardized format, everything we do looks different. Without a unified scaffold, development is slower, more confusing, and less adaptable. One of the benefits of standardizing the format of ML Experimentation is that it enables us to exhaustively record all the important characteristics of Experiment, as well as an assortment of customizable result files – all in a way that allows them to be reused in the future.

What About Data/Metrics? Experiments require an active `Environment` in order to function, from which the Experiment collects important cross-experiment parameters, such as datasets, metrics, cross-validation schemes, and even callbacks to inherit, among many other properties documented in `Environment`

Parameters

model_initializer: Class, or functools.partial, or class instance Algorithm class used to initialize a model, such as XGBoost's *XGBRegressor*, or SKLearn's *KNeighborsClassifier*; although, there are hundreds of possibilities across many different ML libraries. *model_initializer* is expected to define at least *fit* and *predict* methods. *model_initializer* will be initialized with *model_init_params*, and its "extra" methods (*fit*, *predict*, etc.) will be invoked with parameters in *model_extra_params*

model_init_params: Dict, or object (optional) Dictionary of arguments given to create an instance of *model_initializer*. Any kwargs that are considered valid by the *__init__* method of *model_initializer* are valid in *model_init_params*.

One of the key features that makes HyperparameterHunter so magical is that **ALL** hyperparameters in the signature of *model_initializer* (and their default values) are discovered – whether or not they are explicitly given in *model_init_params*. Not only does this make Experiment result descriptions incredibly thorough, it also makes optimization smoother, more effective, and far less work for the user. For example, take LightGBM's *LGBMRegressor*, with *model_init_params'='dict(learning_rate=0.2)*. HyperparameterHunter recognizes that this differs from the default of 0.1. It also recognizes that *LGBMRegressor* is actually initialized with more than a dozen other hyperparameters we didn't bother mentioning, and it records their values, too. So if we want to optimize *num_leaves* tomorrow, the OptPro doesn't start from scratch. It knows that we ran an Experiment that didn't explicitly mention *num_leaves*, but its default value was 31, and it uses this information to fuel optimization – all without us having to manually keep track of tons of janky collections of hyperparameters. In fact, we really don't need to go out of our way at all. HyperparameterHunter just acts as our faithful lab assistant, keeping track of all the stuff we'd rather not worry about

model_extra_params: Dict (optional) Dictionary of extra parameters for models' non-initialization methods (like *fit*, *predict*, *predict_proba*, etc.), and for neural networks. To specify parameters for an extra method, place them in a dict named for the extra method to which the parameters should be given. For example, to call *fit* with *early_stopping_rounds'=5, use 'model_extra_params'='dict(fit=dict(early_stopping_rounds=5))*.

For models whose *fit* methods have a kwarg like *eval_set* (such as XGBoost's), one can use the *DatasetSentinel* attributes of the current active `Environment`, documented under its "Attributes" section and under `train_input`. An example using several DatasetSentinels can be found in HyperparameterHunter's [XGBoost Classification Example](https://github.com/HunterMcGushion/hyperparameter_hunter/blob/master/examples/xgboost_examples/classification.py)

feature_engineer: 'FeatureEngineer', or list (optional) Feature engineering/transformation/pre-processing steps to apply to datasets defined in `Environment`. If list, will be used to initialize `FeatureEngineer`, and can contain any of the following values:

1. `EngineerStep` instance
2. Function input to :class:~hyperparameter_hunter.feature_engineering.EngineerStep'

For important information on properly formatting *EngineerStep* functions, please see the documentation of `EngineerStep`. OptPros can perform hyperparameter optimization of *feature_engineer* steps. This capability adds a third allowed value to the above list and is documented in `forge_experiment()`

feature_selector: List of str, callable, or list of booleans (optional) Column names to include as input data for all provided DataFrames. If None, *feature_selector* is set to all columns in `train_dataset`, less `target_column`, and `id_column`. *feature_selector* is provided as the second argument for calls to *pandas.DataFrame.loc* when constructing datasets

notes: String (optional) Additional information about the Experiment that will be saved with the Experiment's description result file. This serves no purpose other than to facilitate saving Experiment details in a more readable format

do_raise_repeated: Boolean, default=False If True and this Experiment locates a previous Experiment's results with matching Environment and Hyperparameter Keys, a RepeatedExperimentError will be raised. Else, a warning will be logged

auto_start: Boolean, default=True If True, after the Experiment is initialized, it will automatically call `BaseExperiment.preparation_workflow()`, followed by `BaseExperiment.experiment_workflow()`, effectively completing all essential tasks without requiring additional method calls

target_metric: Tuple, str, default=('oof', <:attr:'environment.Environment.metrics'[0]>) Path denoting the metric to be used to compare completed Experiments or to use for certain early stopping procedures in some model classes. The first value should be one of ['oof', 'holdout', 'in_fold']. The second value should be the name of a metric being recorded according to the values supplied in `hyperparameter_hunter.environment.Environment.metrics_params`. See the documentation for `hyperparameter_hunter.metrics.get_formatted_target_metric()` for more info. Any values returned by, or used as the *target_metric* input to this function are acceptable values for *target_metric*

callbacks: 'LambdaCallback', or list of 'LambdaCallback' (optional) Callbacks injected directly into concrete Experiment (*CVExperiment*), adding new functionality, or customizing existing processes. Should be a `LambdaCallback` or a list of such classes. *LambdaCallback* can be created using `callbacks.bases.lambda_callback()`, which documents the options for creating callbacks. *callbacks* will be added to the MRO of the Experiment by `experiment_core.ExperimentMeta` at *__call__* time, making *callbacks* new base classes of the Experiment. See `callbacks.bases.lambda_callback()` for more information. The presence of LambdaCallbacks will not affect Experiment keys. In other words, for the purposes of Experiment matching/recording, all other factors being equal, an Experiment with *callbacks* is considered identical to an Experiment without, despite whatever custom functionality was added by the LambdaCallbacks

See also:

`hyperparameter_hunter.optimization.protocol_core.BaseOptPro.forge_experiment()` OptPro method to define hyperparameter search scaffold for building Experiments during optimization. This method follows the same format as Experiment initialization, but it adds the ability to provide hyperparameter values as ranges to search over, via subclasses of `Dimension`. The other notable difference is that *forge_experiment* removes the *auto_start* and *target_metric* kwargs, which is described in the *forge_experiment* docstring Notes

`Environment` Provides critical information on how Experiments should be conducted, as well as the data to be used by Experiments. An *Environment* must be active before executing any Experiment or OptPro

`lambda_callback()` Enables customization of the Experimentation process and access to all Experiment internals through a collection of methods that are invoked at all the important periods over an Experiment's lifespan. These can be provided via the *experiment_callbacks* kwarg of `Environment`, and the callback classes literally get thrown in to the parent classes of the Experiment, so they're kind of a big deal

4.3 Hyperparameter Optimization

class hyperparameter_hunter.optimization.backends.skopt.protocols.**BayesianOptPro**(*target_metric=*
iterations=1,
verbose=1,
read_experime
reporter_param
*warn_on_re_*
base_estimato
n_initial_poin
acquisition_function=
acquisition_optimizer
random_state=32
*acquisition_function_*
acquisition_optimizer
n_random_sta
callbacks=None,
base_estimato

Bases: hyperparameter_hunter.optimization.protocol_core.SKOptPro

Bayesian optimization with Gaussian Processes

Attributes

search_space_size The number of different hyperparameter permutations possible given the current

source_script

Methods

forge_experiment(self, ...])	model_initializer[,	Define hyperparameter search scaffold for building Experiments during optimization

Continued on next page

Table 2 – continued from previous page

get_ready(self)	Prepare for optimization by finalizing hyperparameter space and identifying similar Experiments.
go(self[, force_ready])	Execute hyperparameter optimization, building an Experiment for each iteration
set_dimensions(self)	Locate given hyperparameters that are *space* choice declarations and add them to dimensions
set_experiment_guidelines(self, *args, ...)	Deprecated since version 3.0.0a2.

__init__(*self*, *target_metric=None*, *iterations=1*, *verbose=1*, *read_experiments=True*, *reporter_parameters=None*, *warn_on_re_ask=False*, *base_estimator='GP'*, *n_initial_points=10*, *acquisition_function='gp_hedge'*, *acquisition_optimizer='auto'*, *random_state=32*, *acquisition_function_kwargs=None*, *acquisition_optimizer_kwargs=None*, *n_random_starts='DEPRECATED'*, *callbacks=None*, *base_estimator_kwargs=None*)

Base class for SKOpt-based Optimization Protocols

There are two important methods for all SKOptPro descendants that should be invoked after initialization:

1. forge_experiment()
2. go()

Parameters

target_metric: Tuple, default=("oof", <:attr:'environment.Environment.metrics'[0]>) Rarely necessary to explicitly provide this, as the default is usually sufficient. Path denoting the metric to be used to compare Experiment performance. The first value should be one of ["oof", "holdout", "in_fold"]. The second value should be the name of a metric being recorded according to environment.Environment.metrics_params. See the documentation for metrics.get_formatted_target_metric() for more info. Any values returned by, or given as the *target_metric* input to, get_formatted_target_metric() are acceptable values for BaseOptPro.target_metric

iterations: Int, default=1 Number of Experiments to conduct during optimization upon invoking BaseOptPro.go()

verbose: {0, 1, 2}, default=1 Verbosity mode for console logging. 0: Silent. 1: Show only logs from the Optimization Protocol. 2: In addition to logs shown when verbose=1, also show the logs from individual Experiments

read_experiments: Boolean, default=True If True, all Experiment records that fit in the current space and guidelines, and match algorithm_name, will be read in and used to fit any optimizers

reporter_parameters: Dict, or None, default=None Additional parameters passed to reporting.OptimizationReporter.__init__(). Note: Unless provided explicitly, the key "do_maximize" will be added by default to *reporter_params*, with a value inferred from the *direction* of target_metric in *G.Env.metrics*. In nearly all cases, the "do_maximize" key should be ignored, as there are very few reasons to explicitly include it

warn_on_re_ask: Boolean, default=False If True, and the internal *optimizer* recommends a point that has already been evaluated on invocation of *ask*, a warning is logged before recommending a random point. Either way, a random point is used instead of already-evaluated recommendations. However, logging the fact that this has taken place can be useful to indicate that the optimizer may be stalling, especially if it repeatedly recommends

the same point. In these cases, if the suggested point is not optimal, it can be helpful to switch a different OptPro (especially *DummyOptPro*), which will suggest points using different criteria

Other Parameters

base_estimator: {SKLearn Regressor, "GP", "RF", "ET", "GBRT", "DUMMY"}, default="GP" If not string, should inherit from *sklearn.base.RegressorMixin*. In addition, the *predict* method should have an optional *return_std* argument, which returns *std(Y | x)*, along with *E[Y | x]*.

If *base_estimator* is a string in {"GP", "RF", "ET", "GBRT", "DUMMY"}, a surrogate model corresponding to the relevant *X_minimize* function is created

n_initial_points: Int, default=10 Number of complete evaluation points necessary before allowing Experiments to be approximated with *base_estimator*. Any valid Experiment records found will count as initialization points. If enough Experiment records are not found, additional points will be randomly sampled

acquisition_function:{"LCB", "EI", "PI", "gp_hedge"}, default="gp_hedge" Function to minimize over the posterior distribution. Can be any of the following:

- "LCB": Lower confidence bound
- "EI": Negative expected improvement
- "PI": Negative probability of improvement
- "gp_hedge": Probabilistically choose one of the above three acquisition functions at every iteration
  - The gains *g_i* are initialized to zero
  - At every iteration,
    - Each acquisition function is optimised independently to propose a candidate point *X_i*
    - Out of all these candidate points, the next point *X_best* is chosen by *softmax(eta g_i)*
    - After fitting the surrogate model with *(X_best, y_best)*, the gains are updated such that *g_i -= mu(X_i)*

acquisition_optimizer: {"sampling", "lbfgs", "auto"}, default="auto" Method to minimize the acquisition function. The fit model is updated with the optimal value obtained by optimizing *acq_func* with *acq_optimizer*

- "sampling": *acq_func* is optimized by computing *acq_func* at *n_initial_points* randomly sampled points.
- "lbfgs": *acq_func* is optimized by
  - Randomly sampling *n_restarts_optimizer* (from *acq_optimizer_kwargs*) points
  - "lbfgs" is run for 20 iterations with these initial points to find local minima
  - The optimal of these local minima is used to update the prior
- "auto": *acq_optimizer* is configured on the basis of the *base_estimator* and the search space. If the space is *Categorical* or if the provided estimator is based on tree-models, then this is set to "sampling"

random_state: Int, 'RandomState' instance, or None, default=None Set to something other than None for reproducible results

acquisition_function_kwargs: Dict, or None, default=dict(xi=0.01, kappa=1.96)
Additional arguments passed to the acquisition function

acquisition_optimizer_kwargs: Dict, or None, default=dict(n_points=10000, n_restarts_optimizer=5, n_jobs=
Additional arguments passed to the acquisition optimizer

n_random_starts: ... Deprecated since version 3.0.0: Use *n_initial_points*, instead. Will be removed in 3.2.0

callbacks: Callable, list of callables, or None, default=[] If callable, then *callbacks(self.optimizer_result)* is called after each update to `optimizer`. If list, then each callable is called

base_estimator_kwargs: Dict, or None, default={} Additional arguments passed to *base_estimator* when it is initialized

Notes

To provide initial input points for evaluation, individual Experiments can be executed prior to instantiating an Optimization Protocol. The results of these Experiments will automatically be detected and cherished by the optimizer.

`SKOptPro` and its children in `optimization` rely heavily on the utilities provided by the *Scikit-Optimize* library, so thank you to the creators and contributors for their excellent work.

Methods

forge_experiment	Define constraints on Experiments conducted by OptPro (like hyperparameter search space)
go	Start optimization

`forge_experiment`(*self*, *model_initializer*, *model_init_params=None*, *model_extra_params=None*, *feature_engineer=None*, *feature_selector=None*, *notes=None*, *do_raise_repeated=True*)
Define hyperparameter search scaffold for building Experiments during optimization

OptPros use this method to guide Experiment construction behind the scenes, which is why it looks just like `hyperparameter_hunter.experiments.BaseExperiment.__init__()`. *forge_experiment* offers one major upgrade to standard Experiment initialization: it accepts hyperparameters not only as concrete values, but also as space choices – using `Real`, `Integer`, and `Categorical`. This functionality applies to the *model_init_params*, *model_extra_params* and *feature_engineer* kwargs. Any Dimensions provided to *forge_experiment* are detected by the OptPro and used to define the hyperparameter search space to be optimized

Parameters

model_initializer: Class, or functools.partial, or class instance Algorithm class used to initialize a model, such as XGBoost's *XGBRegressor*, or SKLearn's *KNeighborsClassifier*; although, there are hundreds of possibilities across many different ML libraries. *model_initializer* is expected to define at least *fit* and *predict* methods. *model_initializer* will be initialized with *model_init_params*, and its extra methods (*fit*, *predict*, etc.) will be invoked with parameters in *model_extra_params*

model_init_params: Dict, or object (optional) Dictionary of arguments given to create an instance of *model_initializer*. Any kwargs that are considered valid by the *__init__* method of *model_initializer* are valid in *model_init_params*.

In addition to providing concrete values, hyperparameters can be expressed as choices (dimensions to optimize) by using instances of `Real`, `Integer`, or `Categorical`. Furthermore, hyperparameter choices and concrete values can be used together in *model_init_params*.

Using XGBoost's *XGBClassifier* to illustrate, the *model_init_params* kwarg of `CVExperiment` is limited to using concrete values, such as `dict(max_depth=10, learning_rate=0.1, booster="gbtree")`. This is still valid for `forge_experiment()`. However, `forge_experiment()` also allows *model_init_params* to consist entirely of space choices, such as `dict(max_depth=Integer(2, 20), learning_rate=Real(0.001, 0.5), booster=Categorical(["gbtree", "dart"]))`, or as any combination of concrete values and choices, for instance, `dict(max_depth=10, learning_rate=Real(0.001, 0.5), booster="gbtree")`.

One of the key features that makes HyperparameterHunter so magical is that **ALL** hyperparameters in the signature of *model_initializer* (and their default values) are discovered – whether or not they are explicitly given in *model_init_params*. Not only does this make Experiment result descriptions incredibly thorough, it also makes optimization smoother, more effective, and far less work for the user. For example, take LightGBM's *LGBMRegressor*, with *model_init_params='dict(learning_rate=0.2)*. HyperparameterHunter recognizes that this differs from the default of 0.1. It also recognizes that *LGBMRegressor* is actually initialized with more than a dozen other hyperparameters we didn't bother mentioning, and it records their values, too. So if we want to optimize *num_leaves* tomorrow, the OptPro doesn't start from scratch. It knows that we ran an Experiment that didn't explicitly mention *num_leaves*, but its default value was 31, and it uses this information to fuel optimization – all without us having to manually keep track of tons of janky collections of hyperparameters. In fact, we really don't need to go out of our way at all. HyperparameterHunter just acts as our faithful lab assistant, keeping track of all the stuff we'd rather not worry about

model_extra_params: Dict (optional) Dictionary of extra parameters for models' non-initialization methods (like *fit*, *predict*, *predict_proba*, etc.), and for neural networks. To specify parameters for an extra method, place them in a dict named for the extra method to which the parameters should be given. For example, to call *fit* with *early_stopping_rounds'=5, use 'model_extra_params'='dict(fit=dict(early_stopping_rounds=5))*.

Declaring hyperparameter space choices works identically to *model_init_params*, meaning that in addition to concrete values, extra parameters can be given as instances of `Real`, `Integer`, or `Categorical`. To optimize over a space in which *early_stopping_rounds* is between 3 and 9, use *model_extra_params'='dict(fit=dict(early_stopping_rounds=Real(3, 9)))*.

For models whose *fit* methods have a kwarg like *eval_set* (such as XGBoost's), one can use the *DatasetSentinel* attributes of the current active `Environment`, documented under its "Attributes" section and under `train_input`. An example using several DatasetSentinels can be found in HyperparameterHunter's [XGBoost Classification Example](https://github.com/HunterMcGushion/hyperparameter_hunter/blob/master/examples/xgboost_examples/classification.py)

feature_engineer: 'FeatureEngineer', or list (optional) Feature engineering/transformation/pre-processing steps to apply to datasets defined in `Environment`. If list, will be used to initialize `FeatureEngineer`, and

can contain any of the following values:

1. `EngineerStep` instance
2. Function input to :class:~hyperparameter_hunter.feature_engineering.EngineerStep‘
3. `Categorical`, with *categories* comprising a selection of the previous two values (optimization only)

For important information on properly formatting *EngineerStep* functions, please see the documentation of `EngineerStep`.

To search a space optionally including an *EngineerStep*, use the *optional* kwarg of `Categorical`. This functionality is illustrated in `FeatureEngineer`. If using a *FeatureEngineer* instance to optimize *feature_engineer*, this instance cannot be used with *CVExperiment* because Experiments can't handle space choices

feature_selector: List of str, callable, or list of booleans (optional) Column names to include as input data for all provided DataFrames. If None, *feature_selector* is set to all columns in `train_dataset`, less `target_column`, and `id_column`. *feature_selector* is provided as the second argument for calls to *pandas.DataFrame.loc* when constructing datasets

notes: String (optional) Additional information about the Experiment that will be saved with the Experiment's description result file. This serves no purpose other than to facilitate saving Experiment details in a more readable format

do_raise_repeated: Boolean, default=False If True and this Experiment locates a previous Experiment's results with matching Environment and Hyperparameter Keys, a RepeatedExperimentError will be raised. Else, a warning will be logged

See also:

`hyperparameter_hunter.experiments.BaseExperiment` One-off experimentation counterpart to an OptPro's `forge_experiment()`. Internally, OptPros feed the processed arguments from *forge_experiment* to initialize Experiments. This hand-off to Experiments takes place in `_execute_experiment()`

Notes

The *auto_start* kwarg is not available here because `_execute_experiment()` sets it to False in order to check for duplicated keys before running the whole Experiment. This and *target_metric* being moved to `__init__()` are the most notable differences between calling `forge_experiment()` and instantiating `CVExperiment`

A more accurate name for this method might be something like "build_experiment_forge", since *forge_experiment* itself does not actually execute any Experiments. However, *forge_experiment* sounds cooler and much less clunky

go(*self*, *force_ready=True*)

Execute hyperparameter optimization, building an Experiment for each iteration

This method may only be invoked after invoking `forge_experiment()`, which defines experiment guidelines and search dimensions. *go* performs a few important tasks: 1) Formally setting the hyperparameter space; 2) Locating similar experiments to be used as learning material (for OptPros that suggest incumbent search points by estimating utilities using surrogate models); and 3) Actually setting off the optimization process, via `_optimization_loop()`

Parameters

force_ready: Boolean, default=False If True, `get_ready()` will be invoked even if it has already been called. This will re-initialize the hyperparameter *space* and *similar_experiments*. Standard behavior is for `go()` to invoke `get_ready()`, so *force_ready* is ignored unless `get_ready()` has been manually invoked

---

class hyperparameter_hunter.optimization.backends.skopt.protocols.**GradientBoostedRegressio**

Bases: hyperparameter_hunter.optimization.protocol_core.SKOptPro

Sequential optimization with gradient boosted regression trees

Attributes

search_space_size The number of different hyperparameter permutations possible given the current

source_script

Methods

forge_experiment(self, model_initializer[, ...])	Define hyperparameter search scaffold for building Experiments during optimization
get_ready(self)	Prepare for optimization by finalizing hyperparameter space and identifying similar Experiments.
go(self[, force_ready])	Execute hyperparameter optimization, building an Experiment for each iteration
set_dimensions(self)	Locate given hyperparameters that are *space* choice declarations and add them to dimensions
set_experiment_guidelines(self, *args, ...)	Deprecated since version 3.0.0a2.

__init__(*self*, *target_metric=None*, *iterations=1*, *verbose=1*, *read_experiments=True*, *reporter_parameters=None*, *warn_on_re_ask=False*, *base_estimator='GBRT'*, *n_initial_points=10*, *acquisition_function='EI'*, *acquisition_optimizer='sampling'*, *random_state=32*, *acquisition_function_kwargs=None*, *acquisition_optimizer_kwargs=None*, *n_random_starts='DEPRECATED'*, *callbacks=None*, *base_estimator_kwargs=None*)

Base class for SKOpt-based Optimization Protocols

There are two important methods for all SKOptPro descendants that should be invoked after initialization:

1. forge_experiment()
2. go()

Parameters

target_metric: Tuple, default=("oof", <:attr:'environment.Environment.metrics'[0]>) Rarely necessary to explicitly provide this, as the default is usually sufficient. Path denoting the metric to be used to compare Experiment performance. The first value should be one of ["oof", "holdout", "in_fold"]. The second value should be the name of a metric being recorded according to environment.Environment.metrics_params. See the documentation for metrics.get_formatted_target_metric() for more info. Any values returned by, or given as the *target_metric* input to, get_formatted_target_metric() are acceptable values for BaseOptPro.target_metric

iterations: Int, default=1 Number of Experiments to conduct during optimization upon invoking BaseOptPro.go()

verbose: {0, 1, 2}, default=1 Verbosity mode for console logging. 0: Silent. 1: Show only logs from the Optimization Protocol. 2: In addition to logs shown when verbose=1, also show the logs from individual Experiments

read_experiments: Boolean, default=True If True, all Experiment records that fit in the current space and guidelines, and match algorithm_name, will be read in and used to fit any optimizers

reporter_parameters: Dict, or None, default=None Additional parameters passed to reporting.OptimizationReporter.__init__(). Note: Unless provided explicitly, the key "do_maximize" will be added by default to *reporter_params*, with a value inferred from the *direction* of target_metric in *G.Env.metrics*. In nearly all cases, the "do_maximize" key should be ignored, as there are very few reasons to explicitly include it

warn_on_re_ask: Boolean, default=False If True, and the internal *optimizer* recommends a point that has already been evaluated on invocation of *ask*, a warning is

logged before recommending a random point. Either way, a random point is used instead of already-evaluated recommendations. However, logging the fact that this has taken place can be useful to indicate that the optimizer may be stalling, especially if it repeatedly recommends the same point. In these cases, if the suggested point is not optimal, it can be helpful to switch a different OptPro (especially *DummyOptPro*), which will suggest points using different criteria

Other Parameters

base_estimator: {SKLearn Regressor, "GP", "RF", "ET", "GBRT", "DUMMY"}, default="GP" If not string, should inherit from *sklearn.base.RegressorMixin*. In addition, the *predict* method should have an optional *return_std* argument, which returns *std(Y | x)*, along with *E[Y | x]*.

If *base_estimator* is a string in {"GP", "RF", "ET", "GBRT", "DUMMY"}, a surrogate model corresponding to the relevant *X_minimize* function is created

n_initial_points: Int, default=10 Number of complete evaluation points necessary before allowing Experiments to be approximated with *base_estimator*. Any valid Experiment records found will count as initialization points. If enough Experiment records are not found, additional points will be randomly sampled

acquisition_function:{"LCB", "EI", "PI", "gp_hedge"}, default="gp_hedge" Function to minimize over the posterior distribution. Can be any of the following:

- "LCB": Lower confidence bound
- "EI": Negative expected improvement
- "PI": Negative probability of improvement
- "gp_hedge": Probabilistically choose one of the above three acquisition functions at every iteration
  - The gains *g_i* are initialized to zero
  - At every iteration,
    - Each acquisition function is optimised independently to propose a candidate point *X_i*
    - Out of all these candidate points, the next point *X_best* is chosen by *softmax(eta g_i)*
    - After fitting the surrogate model with *(X_best, y_best)*, the gains are updated such that *g_i -= mu(X_i)*

acquisition_optimizer: {"sampling", "lbfgs", "auto"}, default="auto" Method to minimize the acquisition function. The fit model is updated with the optimal value obtained by optimizing *acq_func* with *acq_optimizer*

- "sampling": *acq_func* is optimized by computing *acq_func* at *n_initial_points* randomly sampled points.
- "lbfgs": *acq_func* is optimized by
  - Randomly sampling *n_restarts_optimizer* (from *acq_optimizer_kwargs*) points
  - "lbfgs" is run for 20 iterations with these initial points to find local minima
  - The optimal of these local minima is used to update the prior

- "auto": *acq_optimizer* is configured on the basis of the *base_estimator* and the search space. If the space is *Categorical* or if the provided estimator is based on tree-models, then this is set to "sampling"

random_state: Int, 'RandomState' instance, or None, default=None Set to something other than None for reproducible results

acquisition_function_kwargs: Dict, or None, default=dict(xi=0.01, kappa=1.96) Additional arguments passed to the acquisition function

acquisition_optimizer_kwargs: Dict, or None, default=dict(n_points=10000, n_restarts_optimizer=5, n_j Additional arguments passed to the acquisition optimizer

n_random_starts: ... Deprecated since version 3.0.0: Use *n_initial_points*, instead. Will be removed in 3.2.0

callbacks: Callable, list of callables, or None, default=[] If callable, then *callbacks(self.optimizer_result)* is called after each update to `optimizer`. If list, then each callable is called

base_estimator_kwargs: Dict, or None, default={} Additional arguments passed to *base_estimator* when it is initialized

Notes

To provide initial input points for evaluation, individual Experiments can be executed prior to instantiating an Optimization Protocol. The results of these Experiments will automatically be detected and cherished by the optimizer.

`SKOptPro` and its children in `optimization` rely heavily on the utilities provided by the *Scikit-Optimize* library, so thank you to the creators and contributors for their excellent work.

Methods

forge_experiment	Define constraints on Experiments conducted by OptPro (like hyperparameter search space)
go	Start optimization

`forge_experiment`(*self*, *model_initializer*, *model_init_params=None*, *model_extra_params=None*, *feature_engineer=None*, *feature_selector=None*, *notes=None*, *do_raise_repeated=True*)

Define hyperparameter search scaffold for building Experiments during optimization

OptPros use this method to guide Experiment construction behind the scenes, which is why it looks just like `hyperparameter_hunter.experiments.BaseExperiment.__init__()`. *forge_experiment* offers one major upgrade to standard Experiment initialization: it accepts hyperparameters not only as concrete values, but also as space choices – using `Real`, `Integer`, and `Categorical`. This functionality applies to the *model_init_params*, *model_extra_params* and *feature_engineer* kwargs. Any Dimensions provided to *forge_experiment* are detected by the OptPro and used to define the hyperparameter search space to be optimized

Parameters

model_initializer: Class, or functools.partial, or class instance Algorithm class used to initialize a model, such as XGBoost's *XGBRegressor*, or SKLearn's *KNeighborsClassifier*; although, there are hundreds of possibilities across many different ML libraries. *model_initializer* is expected to define at least *fit* and

predict methods. *model_initializer* will be initialized with *model_init_params*, and its extra methods (*fit*, *predict*, etc.) will be invoked with parameters in *model_extra_params*

model_init_params: Dict, or object (optional) Dictionary of arguments given to create an instance of *model_initializer*. Any kwargs that are considered valid by the *__init__* method of *model_initializer* are valid in *model_init_params*.

In addition to providing concrete values, hyperparameters can be expressed as choices (dimensions to optimize) by using instances of `Real`, `Integer`, or `Categorical`. Furthermore, hyperparameter choices and concrete values can be used together in *model_init_params*.

Using XGBoost's *XGBClassifier* to illustrate, the *model_init_params* kwarg of `CVExperiment` is limited to using concrete values, such as `dict(max_depth=10, learning_rate=0.1, booster="gbtree")`. This is still valid for `forge_experiment()`. However, `forge_experiment()` also allows *model_init_params* to consist entirely of space choices, such as `dict(max_depth=Integer(2, 20), learning_rate=Real(0.001, 0.5), booster=Categorical(["gbtree", "dart"]))`, or as any combination of concrete values and choices, for instance, `dict(max_depth=10, learning_rate=Real(0.001, 0.5), booster="gbtree")`.

One of the key features that makes HyperparameterHunter so magical is that **ALL** hyperparameters in the signature of *model_initializer* (and their default values) are discovered – whether or not they are explicitly given in *model_init_params*. Not only does this make Experiment result descriptions incredibly thorough, it also makes optimization smoother, more effective, and far less work for the user. For example, take LightGBM's *LGBMRegressor*, with *model_init_params='dict(learning_rate=0.2)*. HyperparameterHunter recognizes that this differs from the default of 0.1. It also recognizes that *LGBMRegressor* is actually initialized with more than a dozen other hyperparameters we didn't bother mentioning, and it records their values, too. So if we want to optimize *num_leaves* tomorrow, the OptPro doesn't start from scratch. It knows that we ran an Experiment that didn't explicitly mention *num_leaves*, but its default value was 31, and it uses this information to fuel optimization – all without us having to manually keep track of tons of janky collections of hyperparameters. In fact, we really don't need to go out of our way at all. HyperparameterHunter just acts as our faithful lab assistant, keeping track of all the stuff we'd rather not worry about

model_extra_params: Dict (optional) Dictionary of extra parameters for models' non-initialization methods (like *fit*, *predict*, *predict_proba*, etc.), and for neural networks. To specify parameters for an extra method, place them in a dict named for the extra method to which the parameters should be given. For example, to call *fit* with *early_stopping_rounds'=5, use 'model_extra_params'='dict(fit=dict(early_stopping_rounds=5))*.

Declaring hyperparameter space choices works identically to *model_init_params*, meaning that in addition to concrete values, extra parameters can be given as instances of `Real`, `Integer`, or `Categorical`. To optimize over a space in which *early_stopping_rounds* is between 3 and 9, use *model_extra_params'='dict(fit=dict(early_stopping_rounds=Real(3, 9)))*.

For models whose *fit* methods have a kwarg like *eval_set* (such as XGBoost's), one can use the *DatasetSentinel* attributes of the current active `Environment`,

documented under its "Attributes" section and under `train_input`. An example using several DatasetSentinels can be found in HyperparameterHunter's [XGBoost Classification Example](https://github.com/HunterMcGushion/hyperparameter_hunter/blob/master/examples/xgboost_examples/classification.py)

feature_engineer: 'FeatureEngineer', or list (optional) Feature engineering/transformation/pre-processing steps to apply to datasets defined in `Environment`. If list, will be used to initialize `FeatureEngineer`, and can contain any of the following values:

1. `EngineerStep` instance
2. Function input to :class:~hyperparameter_hunter.feature_engineering.EngineerStep'
3. `Categorical`, with *categories* comprising a selection of the previous two values (optimization only)

For important information on properly formatting *EngineerStep* functions, please see the documentation of `EngineerStep`.

To search a space optionally including an *EngineerStep*, use the *optional* kwarg of `Categorical`. This functionality is illustrated in `FeatureEngineer`. If using a *FeatureEngineer* instance to optimize *feature_engineer*, this instance cannot be used with *CVExperiment* because Experiments can't handle space choices

feature_selector: List of str, callable, or list of booleans (optional) Column names to include as input data for all provided DataFrames. If None, *feature_selector* is set to all columns in `train_dataset`, less `target_column`, and `id_column`. *feature_selector* is provided as the second argument for calls to *pandas.DataFrame.loc* when constructing datasets

notes: String (optional) Additional information about the Experiment that will be saved with the Experiment's description result file. This serves no purpose other than to facilitate saving Experiment details in a more readable format

do_raise_repeated: Boolean, default=False If True and this Experiment locates a previous Experiment's results with matching Environment and Hyperparameter Keys, a RepeatedExperimentError will be raised. Else, a warning will be logged

See also:

`hyperparameter_hunter.experiments.BaseExperiment` One-off experimentation counterpart to an OptPro's `forge_experiment()`. Internally, OptPros feed the processed arguments from *forge_experiment* to initialize Experiments. This hand-off to Experiments takes place in `_execute_experiment()`

Notes

The *auto_start* kwarg is not available here because `_execute_experiment()` sets it to False in order to check for duplicated keys before running the whole Experiment. This and *target_metric* being moved to `__init__()` are the most notable differences between calling `forge_experiment()` and instantiating `CVExperiment`

A more accurate name for this method might be something like "build_experiment_forge", since *forge_experiment* itself does not actually execute any Experiments. However, *forge_experiment* sounds cooler and much less clunky

go(*self*, *force_ready=True*)

Execute hyperparameter optimization, building an Experiment for each iteration

This method may only be invoked after invoking `forge_experiment()`, which defines experiment guidelines and search dimensions. *go* performs a few important tasks: 1) Formally setting the hyperparameter space; 2) Locating similar experiments to be used as learning material (for OptPros that suggest incumbent search points by estimating utilities using surrogate models); and 3) Actually setting off the optimization process, via `_optimization_loop()`

Parameters

force_ready: Boolean, default=False If True, `get_ready()` will be invoked even if it has already been called. This will re-initialize the hyperparameter *space* and *similar_experiments*. Standard behavior is for `go()` to invoke `get_ready()`, so *force_ready* is ignored unless `get_ready()` has been manually invoked

---

class hyperparameter_hunter.optimization.backends.skopt.protocols.**RandomForestOptPro**(*target_metric=('oof', 'roc_auc'), iterations=1, verbose=1, read_experiments=True, reporter_parameters=None, warn_on_re_gets=True, base_estimator='RF', n_initial_points=10, acq_func='gp_hedge', acquisition_optimizer_kwargs=None, random_state=None, acq_func_kwargs=None, acquisition_optimizer='sampling', n_random_starts=None, callbacks=None, base_estimator_kwargs=None*)

Bases: `hyperparameter_hunter.optimization.protocol_core.SKOptPro`

Sequential optimization with random forest regressor decision trees

Attributes

search_space_size The number of different hyperparameter permutations possible given the current

source_script

Methods

`forge_experiment`(self, model_initializer[, ...])	Define hyperparameter search scaffold for building Experiments during optimization
`get_ready`(self)	Prepare for optimization by finalizing hyperparameter space and identifying similar Experiments.
`go`(self[, force_ready])	Execute hyperparameter optimization, building an Experiment for each iteration
`set_dimensions`(self)	Locate given hyperparameters that are *space* choice declarations and add them to `dimensions`
`set_experiment_guidelines`(self, *args, ...)	Deprecated since version 3.0.0a2.

__init__(*self*, *target_metric=None*, *iterations=1*, *verbose=1*, *read_experiments=True*, *reporter_parameters=None*, *warn_on_re_ask=False*, *base_estimator='RF'*, *n_initial_points=10*, *acquisition_function='EI'*, *acquisition_optimizer='sampling'*, *random_state=32*, *acquisition_function_kwargs=None*, *acquisition_optimizer_kwargs=None*, *n_random_starts='DEPRECATED'*, *callbacks=None*, *base_estimator_kwargs=None*)

Base class for SKOpt-based Optimization Protocols

There are two important methods for all `SKOptPro` descendants that should be invoked after initialization:

1. `forge_experiment()`
2. `go()`

Parameters

target_metric: Tuple, default=("oof", <:attr:'environment.Environment.metrics'[0]>) Rarely necessary to explicitly provide this, as the default is usually sufficient. Path denoting the metric to be used to compare Experiment performance. The first value should be one of ["oof", "holdout", "in_fold"]. The second value should be the name of a metric being recorded according to `environment.Environment.metrics_params`. See the documentation for `metrics.get_formatted_target_metric()` for more info. Any values returned by, or given as the *target_metric* input to, `get_formatted_target_metric()` are acceptable values for `BaseOptPro.target_metric`

iterations: Int, default=1 Number of Experiments to conduct during optimization upon invoking `BaseOptPro.go()`

verbose: {0, 1, 2}, default=1 Verbosity mode for console logging. 0: Silent. 1: Show only logs from the Optimization Protocol. 2: In addition to logs shown when verbose=1, also show the logs from individual Experiments

read_experiments: Boolean, default=True If True, all Experiment records that fit in the current `space` and guidelines, and match `algorithm_name`, will be read in and used to fit any optimizers

reporter_parameters: Dict, or None, default=None Additional parameters passed to `reporting.OptimizationReporter.__init__()`. Note: Unless provided explicitly, the key "do_maximize" will be added by default to *reporter_params*, with a value inferred from the *direction* of `target_metric` in

G.Env.metrics. In nearly all cases, the "do_maximize" key should be ignored, as there are very few reasons to explicitly include it

warn_on_re_ask: Boolean, default=False If True, and the internal *optimizer* recommends a point that has already been evaluated on invocation of *ask*, a warning is logged before recommending a random point. Either way, a random point is used instead of already-evaluated recommendations. However, logging the fact that this has taken place can be useful to indicate that the optimizer may be stalling, especially if it repeatedly recommends the same point. In these cases, if the suggested point is not optimal, it can be helpful to switch a different OptPro (especially *DummyOptPro*), which will suggest points using different criteria

Other Parameters

base_estimator: {SKLearn Regressor, "GP", "RF", "ET", "GBRT", "DUMMY"}, default="GP" If not string, should inherit from *sklearn.base.RegressorMixin*. In addition, the *predict* method should have an optional *return_std* argument, which returns *std(Y | x)*, along with *E[Y | x]*.

If *base_estimator* is a string in {"GP", "RF", "ET", "GBRT", "DUMMY"}, a surrogate model corresponding to the relevant *X_minimize* function is created

n_initial_points: Int, default=10 Number of complete evaluation points necessary before allowing Experiments to be approximated with *base_estimator*. Any valid Experiment records found will count as initialization points. If enough Experiment records are not found, additional points will be randomly sampled

acquisition_function:{"LCB", "EI", "PI", "gp_hedge"}, default="gp_hedge" Function to minimize over the posterior distribution. Can be any of the following:

- "LCB": Lower confidence bound
- "EI": Negative expected improvement
- "PI": Negative probability of improvement
- "gp_hedge": Probabilistically choose one of the above three acquisition functions at every iteration
  - The gains *g_i* are initialized to zero
  - At every iteration,
    - Each acquisition function is optimised independently to propose a candidate point *X_i*
    - Out of all these candidate points, the next point *X_best* is chosen by *softmax(eta g_i)*
    - After fitting the surrogate model with *(X_best, y_best)*, the gains are updated such that *g_i -= mu(X_i)*

acquisition_optimizer: {"sampling", "lbfgs", "auto"}, default="auto" Method to minimize the acquisition function. The fit model is updated with the optimal value obtained by optimizing *acq_func* with *acq_optimizer*

- "sampling": *acq_func* is optimized by computing *acq_func* at *n_initial_points* randomly sampled points.
- "lbfgs": *acq_func* is optimized by
  - Randomly sampling *n_restarts_optimizer* (from *acq_optimizer_kwargs*) points

- "lbfgs" is run for 20 iterations with these initial points to find local minima
- The optimal of these local minima is used to update the prior

- "auto": *acq_optimizer* is configured on the basis of the *base_estimator* and the search space. If the space is *Categorical* or if the provided estimator is based on tree-models, then this is set to "sampling"

random_state: Int, 'RandomState' instance, or None, default=None Set to something other than None for reproducible results

acquisition_function_kwargs: Dict, or None, default=dict(xi=0.01, kappa=1.96) Additional arguments passed to the acquisition function

acquisition_optimizer_kwargs: Dict, or None, default=dict(n_points=10000, n_restarts_optimizer=5, n_j Additional arguments passed to the acquisition optimizer

n_random_starts: ... Deprecated since version 3.0.0: Use *n_initial_points*, instead. Will be removed in 3.2.0

callbacks: Callable, list of callables, or None, default=[] If callable, then *callbacks(self.optimizer_result)* is called after each update to `optimizer`. If list, then each callable is called

base_estimator_kwargs: Dict, or None, default={} Additional arguments passed to *base_estimator* when it is initialized

Notes

To provide initial input points for evaluation, individual Experiments can be executed prior to instantiating an Optimization Protocol. The results of these Experiments will automatically be detected and cherished by the optimizer.

`SKOptPro` and its children in `optimization` rely heavily on the utilities provided by the *Scikit-Optimize* library, so thank you to the creators and contributors for their excellent work.

Methods

forge_experiment	Define constraints on Experiments conducted by OptPro (like hyperparameter search space)
go	Start optimization

forge_experiment(*self*, *model_initializer*, *model_init_params=None*, *model_extra_params=None*, *feature_engineer=None*, *feature_selector=None*, *notes=None*, *do_raise_repeated=True*)

Define hyperparameter search scaffold for building Experiments during optimization

OptPros use this method to guide Experiment construction behind the scenes, which is why it looks just like `hyperparameter_hunter.experiments.BaseExperiment.__init__()`. *forge_experiment* offers one major upgrade to standard Experiment initialization: it accepts hyperparameters not only as concrete values, but also as space choices – using `Real`, `Integer`, and `Categorical`. This functionality applies to the *model_init_params*, *model_extra_params* and *feature_engineer* kwargs. Any Dimensions provided to *forge_experiment* are detected by the OptPro and used to define the hyperparameter search space to be optimized

Parameters

model_initializer: Class, or functools.partial, or class instance Algorithm class used to initialize a model, such as XGBoost's *XGBRegressor*, or SKLearn's *KNeighborsClassifier*; although, there are hundreds of possibilities across many different ML libraries. *model_initializer* is expected to define at least *fit* and *predict* methods. *model_initializer* will be initialized with *model_init_params*, and its extra methods (*fit*, *predict*, etc.) will be invoked with parameters in *model_extra_params*

model_init_params: Dict, or object (optional) Dictionary of arguments given to create an instance of *model_initializer*. Any kwargs that are considered valid by the *__init__* method of *model_initializer* are valid in *model_init_params*.

In addition to providing concrete values, hyperparameters can be expressed as choices (dimensions to optimize) by using instances of `Real`, `Integer`, or `Categorical`. Furthermore, hyperparameter choices and concrete values can be used together in *model_init_params*.

Using XGBoost's *XGBClassifier* to illustrate, the *model_init_params* kwarg of `CVExperiment` is limited to using concrete values, such as `dict(max_depth=10, learning_rate=0.1, booster="gbtree")`. This is still valid for `forge_experiment()`. However, `forge_experiment()` also allows *model_init_params* to consist entirely of space choices, such as `dict(max_depth=Integer(2, 20), learning_rate=Real(0.001, 0.5), booster=Categorical(["gbtree", "dart"]))`, or as any combination of concrete values and choices, for instance, `dict(max_depth=10, learning_rate=Real(0.001, 0.5), booster="gbtree")`.

One of the key features that makes HyperparameterHunter so magical is that **ALL** hyperparameters in the signature of *model_initializer* (and their default values) are discovered – whether or not they are explicitly given in *model_init_params*. Not only does this make Experiment result descriptions incredibly thorough, it also makes optimization smoother, more effective, and far less work for the user. For example, take LightGBM's *LGBMRegressor*, with *model_init_params'='dict(learning_rate=0.2)*. HyperparameterHunter recognizes that this differs from the default of 0.1. It also recognizes that *LGBMRegressor* is actually initialized with more than a dozen other hyperparameters we didn't bother mentioning, and it records their values, too. So if we want to optimize *num_leaves* tomorrow, the OptPro doesn't start from scratch. It knows that we ran an Experiment that didn't explicitly mention *num_leaves*, but its default value was 31, and it uses this information to fuel optimization – all without us having to manually keep track of tons of janky collections of hyperparameters. In fact, we really don't need to go out of our way at all. HyperparameterHunter just acts as our faithful lab assistant, keeping track of all the stuff we'd rather not worry about

model_extra_params: Dict (optional) Dictionary of extra parameters for models' non-initialization methods (like *fit*, *predict*, *predict_proba*, etc.), and for neural networks. To specify parameters for an extra method, place them in a dict named for the extra method to which the parameters should be given. For example, to call *fit* with *early_stopping_rounds'=5, use 'model_extra_params'='dict(fit=dict(early_stopping_rounds=5))*.

Declaring hyperparameter space choices works identically to *model_init_params*, meaning that in addition to concrete values, extra parameters can be given as instances of `Real`, `Integer`, or `Categorical`. To optimize over a space in which *early_stopping_rounds* is between 3 and 9, use

model_extra_params='*dict(fit=dict(early_stopping_rounds=Real(3, 9)))*.

For models whose *fit* methods have a kwarg like *eval_set* (such as XGBoost's), one can use the *DatasetSentinel* attributes of the current active `Environment`, documented under its "Attributes" section and under `train_input`. An example using several DatasetSentinels can be found in HyperparameterHunter's [XGBoost Classification Example](https://github.com/HunterMcGushion/hyperparameter_hunter/blob/master/examples/xgboost_examples/classification.py)

feature_engineer: 'FeatureEngineer', or list (optional) Feature engineering/transformation/pre-processing steps to apply to datasets defined in `Environment`. If list, will be used to initialize `FeatureEngineer`, and can contain any of the following values:

1. `EngineerStep` instance
2. Function input to :class:~hyperparameter_hunter.feature_engineering.EngineerStep'
3. `Categorical`, with *categories* comprising a selection of the previous two values (optimization only)

For important information on properly formatting *EngineerStep* functions, please see the documentation of `EngineerStep`.

To search a space optionally including an *EngineerStep*, use the *optional* kwarg of `Categorical`. This functionality is illustrated in `FeatureEngineer`. If using a *FeatureEngineer* instance to optimize *feature_engineer*, this instance cannot be used with *CVExperiment* because Experiments can't handle space choices

feature_selector: List of str, callable, or list of booleans (optional) Column names to include as input data for all provided DataFrames. If None, *feature_selector* is set to all columns in `train_dataset`, less `target_column`, and `id_column`. *feature_selector* is provided as the second argument for calls to *pandas.DataFrame.loc* when constructing datasets

notes: String (optional) Additional information about the Experiment that will be saved with the Experiment's description result file. This serves no purpose other than to facilitate saving Experiment details in a more readable format

do_raise_repeated: Boolean, default=False If True and this Experiment locates a previous Experiment's results with matching Environment and Hyperparameter Keys, a RepeatedExperimentError will be raised. Else, a warning will be logged

See also:

`hyperparameter_hunter.experiments.BaseExperiment` One-off experimentation counterpart to an OptPro's `forge_experiment()`. Internally, OptPros feed the processed arguments from *forge_experiment* to initialize Experiments. This hand-off to Experiments takes place in `_execute_experiment()`

Notes

The *auto_start* kwarg is not available here because `_execute_experiment()` sets it to False in order to check for duplicated keys before running the whole Experiment. This and *target_metric* being moved to `__init__()` are the most notable differences between calling `forge_experiment()` and instantiating `CVExperiment`

A more accurate name for this method might be something like "build_experiment_forge", since *forge_experiment* itself does not actually execute any Experiments. However, *forge_experiment* sounds cooler and much less clunky

go(*self*, *force_ready=True*)

Execute hyperparameter optimization, building an Experiment for each iteration

This method may only be invoked after invoking `forge_experiment()`, which defines experiment guidelines and search dimensions. *go* performs a few important tasks: 1) Formally setting the hyperparameter space; 2) Locating similar experiments to be used as learning material (for OptPros that suggest incumbent search points by estimating utilities using surrogate models); and 3) Actually setting off the optimization process, via `_optimization_loop()`

Parameters

force_ready: Boolean, default=False If True, `get_ready()` will be invoked even if it has already been called. This will re-initialize the hyperparameter *space* and *similar_experiments*. Standard behavior is for `go()` to invoke `get_ready()`, so *force_ready* is ignored unless `get_ready()` has been manually invoked

---

class hyperparameter_hunter.optimization.backends.skopt.protocols.**ExtraTreesOptPro**(*target_met*
it-
er-
a-
tions=1,
ver-
bose=1,
read_exper
re-
porter_par
warn_on_r
base_estim
n_initial_p
ac-
qui-
si-
tion_functi
ac-
qui-
si-
tion_optim
ran-
dom_state=
ac-
qui-
si-
tion_functi
ac-
qui-
si-
tion_optim
*n_random_*
call-
backs=Non
base_estim

Bases: `hyperparameter_hunter.optimization.protocol_core.SKOptPro`

Sequential optimization with extra trees regressor decision trees

Attributes

search_space_size The number of different hyperparameter permutations possible given the current

source_script

Methods

`forge_experiment`(self, model_initializer[, ...])	Define hyperparameter search scaffold for building Experiments during optimization
`get_ready`(self)	Prepare for optimization by finalizing hyperparameter space and identifying similar Experiments.
`go`(self[, force_ready])	Execute hyperparameter optimization, building an Experiment for each iteration
`set_dimensions`(self)	Locate given hyperparameters that are *space* choice declarations and add them to `dimensions`
`set_experiment_guidelines`(self, *args, ...)	Deprecated since version 3.0.0a2.

__init__(*self*, *target_metric=None*, *iterations=1*, *verbose=1*, *read_experiments=True*, *reporter_parameters=None*, *warn_on_re_ask=False*, *base_estimator='ET'*, *n_initial_points=10*, *acquisition_function='EI'*, *acquisition_optimizer='sampling'*, *random_state=32*, *acquisition_function_kwargs=None*, *acquisition_optimizer_kwargs=None*, *n_random_starts='DEPRECATED'*, *callbacks=None*, *base_estimator_kwargs=None*)

Base class for SKOpt-based Optimization Protocols

There are two important methods for all `SKOptPro` descendants that should be invoked after initialization:

1. `forge_experiment()`
2. `go()`

Parameters

target_metric: Tuple, default=("oof", <:attr:'environment.Environment.metrics'[0]>) Rarely necessary to explicitly provide this, as the default is usually sufficient. Path denoting the metric to be used to compare Experiment performance. The first value should be one of ["oof", "holdout", "in_fold"]. The second value should be the name of a metric being recorded according to `environment.Environment.metrics_params`. See the documentation for `metrics.get_formatted_target_metric()` for more info. Any values returned by, or given as the *target_metric* input to, `get_formatted_target_metric()` are acceptable values for `BaseOptPro.target_metric`

iterations: Int, default=1 Number of Experiments to conduct during optimization upon invoking `BaseOptPro.go()`

verbose: {0, 1, 2}, default=1 Verbosity mode for console logging. 0: Silent. 1: Show only logs from the Optimization Protocol. 2: In addition to logs shown when verbose=1, also show the logs from individual Experiments

read_experiments: Boolean, default=True If True, all Experiment records that fit in the current `space` and guidelines, and match `algorithm_name`, will be read in and used to fit any optimizers

reporter_parameters: Dict, or None, default=None Additional parameters passed to `reporting.OptimizationReporter.__init__()`. Note: Unless provided explicitly, the key "do_maximize" will be added by default to *reporter_params*, with a value inferred from the *direction* of `target_metric` in *G.Env.metrics*. In nearly all cases, the "do_maximize" key should be ignored, as there are very few reasons to explicitly include it

warn_on_re_ask: Boolean, default=False If True, and the internal *optimizer* recommends a point that has already been evaluated on invocation of *ask*, a warning is logged before recommending a random point. Either way, a random point is used instead of already-evaluated recommendations. However, logging the fact that this has taken place can be useful to indicate that the optimizer may be stalling, especially if it repeatedly recommends the same point. In these cases, if the suggested point is not optimal, it can be helpful to switch a different OptPro (especially *DummyOptPro*), which will suggest points using different criteria

Other Parameters

base_estimator: {SKLearn Regressor, "GP", "RF", "ET", "GBRT", "DUMMY"}, default="GP" If not string, should inherit from *sklearn.base.RegressorMixin*. In addition, the *predict* method should have an optional *return_std* argument, which returns *std(Y | x)*, along with *E[Y | x]*.

If *base_estimator* is a string in {"GP", "RF", "ET", "GBRT", "DUMMY"}, a surrogate model corresponding to the relevant *X_minimize* function is created

n_initial_points: Int, default=10 Number of complete evaluation points necessary before allowing Experiments to be approximated with *base_estimator*. Any valid Experiment records found will count as initialization points. If enough Experiment records are not found, additional points will be randomly sampled

acquisition_function:{"LCB", "EI", "PI", "gp_hedge"}, default="gp_hedge" Function to minimize over the posterior distribution. Can be any of the following:

- "LCB": Lower confidence bound
- "EI": Negative expected improvement
- "PI": Negative probability of improvement
- "gp_hedge": Probabilistically choose one of the above three acquisition functions at every iteration
  - The gains *g_i* are initialized to zero
  - At every iteration,
    - Each acquisition function is optimised independently to propose a candidate point *X_i*
    - Out of all these candidate points, the next point *X_best* is chosen by *softmax(eta g_i)*
    - After fitting the surrogate model with *(X_best, y_best)*, the gains are updated such that *g_i -= mu(X_i)*

acquisition_optimizer: {"sampling", "lbfgs", "auto"}, default="auto" Method to minimize the acquisition function. The fit model is updated with the optimal value obtained by optimizing *acq_func* with *acq_optimizer*

- "sampling": *acq_func* is optimized by computing *acq_func* at *n_initial_points* randomly sampled points.
- "lbfgs": *acq_func* is optimized by
  - Randomly sampling *n_restarts_optimizer* (from *acq_optimizer_kwargs*) points
  - "lbfgs" is run for 20 iterations with these initial points to find local minima
  - The optimal of these local minima is used to update the prior
- "auto": *acq_optimizer* is configured on the basis of the *base_estimator* and the search space. If the space is *Categorical* or if the provided estimator is based on tree-models, then this is set to "sampling"

random_state: Int, 'RandomState' instance, or None, default=None Set to something other than None for reproducible results

acquisition_function_kwargs: Dict, or None, default=dict(xi=0.01, kappa=1.96) Additional arguments passed to the acquisition function

acquisition_optimizer_kwargs: Dict, or None, default=dict(n_points=10000, n_restarts_optimizer=5, n_j Additional arguments passed to the acquisition optimizer

n_random_starts: ... Deprecated since version 3.0.0: Use *n_initial_points*, instead. Will be removed in 3.2.0

callbacks: Callable, list of callables, or None, default=[] If callable, then *callbacks(self.optimizer_result)* is called after each update to `optimizer`. If list, then each callable is called

base_estimator_kwargs: Dict, or None, default={} Additional arguments passed to *base_estimator* when it is initialized

Notes

To provide initial input points for evaluation, individual Experiments can be executed prior to instantiating an Optimization Protocol. The results of these Experiments will automatically be detected and cherished by the optimizer.

`SKOptPro` and its children in `optimization` rely heavily on the utilities provided by the *Scikit-Optimize* library, so thank you to the creators and contributors for their excellent work.

Methods

forge_experiment	Define constraints on Experiments conducted by OptPro (like hyperparameter search space)
go	Start optimization

forge_experiment(*self*, *model_initializer*, *model_init_params=None*, *model_extra_params=None*, *feature_engineer=None*, *feature_selector=None*, *notes=None*, *do_raise_repeated=True*)

Define hyperparameter search scaffold for building Experiments during optimization

OptPros use this method to guide Experiment construction behind the scenes, which is why it looks just like `hyperparameter_hunter.experiments.BaseExperiment.__init__()`. *forge_experiment* offers one major upgrade to standard Experiment initialization: it accepts hyperparameters not only as concrete values, but also as space choices – using `Real`, `Integer`, and `Categorical`. This functionality applies to the *model_init_params*, *model_extra_params* and *feature_engineer* kwargs. Any Dimensions provided to *forge_experiment* are detected by the OptPro and used to define the hyperparameter search space to be optimized

Parameters

model_initializer: Class, or functools.partial, or class instance Algorithm class used to initialize a model, such as XGBoost's *XGBRegressor*, or SKLearn's *KNeighborsClassifier*; although, there are hundreds of possibilities across many different ML libraries. *model_initializer* is expected to define at least *fit* and *predict* methods. *model_initializer* will be initialized with *model_init_params*, and its extra methods (*fit*, *predict*, etc.) will be invoked with parameters in *model_extra_params*

model_init_params: Dict, or object (optional) Dictionary of arguments given to create an instance of *model_initializer*. Any kwargs that are considered valid by the *__init__* method of *model_initializer* are valid in *model_init_params*.

In addition to providing concrete values, hyperparameters can be expressed as choices (dimensions to optimize) by using instances of `Real`, `Integer`, or `Categorical`. Furthermore, hyperparameter choices and concrete values can be used together in *model_init_params*.

Using XGBoost's *XGBClassifier* to illustrate, the *model_init_params* kwarg of `CVExperiment` is limited to using concrete values, such as `dict(max_depth=10, learning_rate=0.1, booster="gbtree")`. This is still valid for `forge_experiment()`. However, `forge_experiment()` also allows *model_init_params* to consist entirely of space choices, such as `dict(max_depth=Integer(2, 20), learning_rate=Real(0.001, 0.5), booster=Categorical(["gbtree", "dart"]))`, or as any combination of concrete values and choices, for instance, `dict(max_depth=10, learning_rate=Real(0.001, 0.5), booster="gbtree")`.

One of the key features that makes HyperparameterHunter so magical is that **ALL** hyperparameters in the signature of *model_initializer* (and their default values) are discovered – whether or not they are explicitly given in *model_init_params*. Not only does this make Experiment result descriptions incredibly thorough, it also makes optimization smoother, more effective, and far less work for the user. For example, take LightGBM's *LGBMRegressor*, with *model_init_params*‘=‘*dict(learning_rate=0.2)*. HyperparameterHunter recognizes that this differs from the default of 0.1. It also recognizes that *LGBMRegressor* is actually initialized with more than a dozen other hyperparameters we didn't bother mentioning, and it records their values, too. So if we want to optimize *num_leaves* tomorrow, the OptPro doesn't start from scratch. It knows that we ran an Experiment that didn't explicitly mention *num_leaves*, but its default value was 31, and it uses this information to fuel optimization – all without us having to manually keep track of tons of janky collections of hyperparameters. In fact, we really don't need to go out of our way at all. HyperparameterHunter just acts as our faithful lab assistant, keeping track of all the stuff we'd rather not worry about

model_extra_params: Dict (optional) Dictionary of extra parameters for models'

non-initialization methods (like *fit*, *predict*, *predict_proba*, etc.), and for neural networks. To specify parameters for an extra method, place them in a dict named for the extra method to which the parameters should be given. For example, to call *fit* with *early_stopping_rounds'=5, use 'model_extra_params'='dict(fit=dict(early_stopping_rounds=5))*.

Declaring hyperparameter space choices works identically to *model_init_params*, meaning that in addition to concrete values, extra parameters can be given as instances of `Real`, `Integer`, or `Categorical`. To optimize over a space in which *early_stopping_rounds* is between 3 and 9, use *model_extra_params'='dict(fit=dict(early_stopping_rounds=Real(3, 9)))*.

For models whose *fit* methods have a kwarg like *eval_set* (such as XGBoost's), one can use the *DatasetSentinel* attributes of the current active `Environment`, documented under its "Attributes" section and under `train_input`. An example using several DatasetSentinels can be found in HyperparameterHunter's [XGBoost Classification Example](https://github.com/HunterMcGushion/hyperparameter_hunter/blob/master/examples/xgboost_examples/classification.py)

feature_engineer: 'FeatureEngineer', or list (optional) Feature engineering/transformation/pre-processing steps to apply to datasets defined in `Environment`. If list, will be used to initialize `FeatureEngineer`, and can contain any of the following values:

1. `EngineerStep` instance
2. Function input to :class:~hyperparameter_hunter.feature_engineering.EngineerStep'
3. `Categorical`, with *categories* comprising a selection of the previous two values (optimization only)

For important information on properly formatting *EngineerStep* functions, please see the documentation of `EngineerStep`.

To search a space optionally including an *EngineerStep*, use the *optional* kwarg of `Categorical`. This functionality is illustrated in `FeatureEngineer`. If using a *FeatureEngineer* instance to optimize *feature_engineer*, this instance cannot be used with *CVExperiment* because Experiments can't handle space choices

feature_selector: List of str, callable, or list of booleans (optional) Column names to include as input data for all provided DataFrames. If None, *feature_selector* is set to all columns in `train_dataset`, less `target_column`, and `id_column`. *feature_selector* is provided as the second argument for calls to *pandas.DataFrame.loc* when constructing datasets

notes: String (optional) Additional information about the Experiment that will be saved with the Experiment's description result file. This serves no purpose other than to facilitate saving Experiment details in a more readable format

do_raise_repeated: Boolean, default=False If True and this Experiment locates a previous Experiment's results with matching Environment and Hyperparameter Keys, a RepeatedExperimentError will be raised. Else, a warning will be logged

See also:

`hyperparameter_hunter.experiments.BaseExperiment` One-off experimentation counterpart to an OptPro's `forge_experiment()`. Internally, OptPros feed the processed arguments from *forge_experiment* to initialize Experiments. This hand-off to Experiments takes place in `_execute_experiment()`

Notes

The *auto_start* kwarg is not available here because `_execute_experiment()` sets it to False in order to check for duplicated keys before running the whole Experiment. This and *target_metric* being moved to `__init__()` are the most notable differences between calling `forge_experiment()` and instantiating `CVExperiment`

A more accurate name for this method might be something like "build_experiment_forge", since *forge_experiment* itself does not actually execute any Experiments. However, *forge_experiment* sounds cooler and much less clunky

go(*self*, *force_ready=True*)

Execute hyperparameter optimization, building an Experiment for each iteration

This method may only be invoked after invoking `forge_experiment()`, which defines experiment guidelines and search dimensions. *go* performs a few important tasks: 1) Formally setting the hyperparameter space; 2) Locating similar experiments to be used as learning material (for OptPros that suggest incumbent search points by estimating utilities using surrogate models); and 3) Actually setting off the optimization process, via `_optimization_loop()`

Parameters

force_ready: Boolean, default=False If True, `get_ready()` will be invoked even if it has already been called. This will re-initialize the hyperparameter *space* and *similar_experiments*. Standard behavior is for `go()` to invoke `get_ready()`, so *force_ready* is ignored unless `get_ready()` has been manually invoked

class hyperparameter_hunter.optimization.backends.skopt.protocols.**DummyOptPro**(*target_metric=No*
it-
er-
a-
tions=1,
ver-
bose=1,
read_experiments=
re-
porter_parameters
warn_on_re_ask=
base_estimator='I
n_initial_points=1
ac-
qui-
si-
tion_function='EI
ac-
qui-
si-
tion_optimizer='s
ran-
dom_state=32,
ac-
qui-
si-
tion_function_kwa
ac-
qui-
si-
tion_optimizer_kw
n_random_starts=
call-
backs=None,
base_estimator_kw

Bases: hyperparameter_hunter.optimization.protocol_core.SKOptPro

Random search by uniform sampling

Attributes

search_space_size The number of different hyperparameter permutations possible given the current

source_script

Methods

forge_experiment(self, model_initializer[, ...])	Define hyperparameter search scaffold for building Experiments during optimization
get_ready(self)	Prepare for optimization by finalizing hyperparameter space and identifying similar Experiments.
go(self[, force_ready])	Execute hyperparameter optimization, building an Experiment for each iteration

Continued on next page

Table 6 – continued from previous page

set_dimensions(self)	Locate given hyperparameters that are *space* choice declarations and add them to dimensions
set_experiment_guidelines(self, *args, ...)	Deprecated since version 3.0.0a2.

__init__(*self*, *target_metric=None*, *iterations=1*, *verbose=1*, *read_experiments=True*, *reporter_parameters=None*, *warn_on_re_ask=False*, *base_estimator='DUMMY'*, *n_initial_points=10*, *acquisition_function='EI'*, *acquisition_optimizer='sampling'*, *random_state=32*, *acquisition_function_kwargs=None*, *acquisition_optimizer_kwargs=None*, *n_random_starts='DEPRECATED'*, *callbacks=None*, *base_estimator_kwargs=None*)

Base class for SKOpt-based Optimization Protocols

There are two important methods for all SKOptPro descendants that should be invoked after initialization:

1. forge_experiment()
2. go()

Parameters

target_metric: Tuple, default=("oof", <:attr:'environment.Environment.metrics'[0]>) Rarely necessary to explicitly provide this, as the default is usually sufficient. Path denoting the metric to be used to compare Experiment performance. The first value should be one of ["oof", "holdout", "in_fold"]. The second value should be the name of a metric being recorded according to environment.Environment.metrics_params. See the documentation for metrics.get_formatted_target_metric() for more info. Any values returned by, or given as the *target_metric* input to, get_formatted_target_metric() are acceptable values for BaseOptPro.target_metric

iterations: Int, default=1 Number of Experiments to conduct during optimization upon invoking BaseOptPro.go()

verbose: {0, 1, 2}, default=1 Verbosity mode for console logging. 0: Silent. 1: Show only logs from the Optimization Protocol. 2: In addition to logs shown when verbose=1, also show the logs from individual Experiments

read_experiments: Boolean, default=True If True, all Experiment records that fit in the current space and guidelines, and match algorithm_name, will be read in and used to fit any optimizers

reporter_parameters: Dict, or None, default=None Additional parameters passed to reporting.OptimizationReporter.__init__(). Note: Unless provided explicitly, the key "do_maximize" will be added by default to *reporter_params*, with a value inferred from the *direction* of target_metric in *G.Env.metrics*. In nearly all cases, the "do_maximize" key should be ignored, as there are very few reasons to explicitly include it

warn_on_re_ask: Boolean, default=False If True, and the internal *optimizer* recommends a point that has already been evaluated on invocation of *ask*, a warning is logged before recommending a random point. Either way, a random point is used instead of already-evaluated recommendations. However, logging the fact that this has taken place can be useful to indicate that the optimizer may be stalling, especially if it repeatedly recommends the same point. In these cases, if the suggested point is not optimal, it can be helpful to switch a different OptPro (especially *DummyOptPro*), which will suggest points using different criteria

Other Parameters

base_estimator: {SKLearn Regressor, "GP", "RF", "ET", "GBRT", "DUMMY"}, default="GP" If not string, should inherit from *sklearn.base.RegressorMixin*. In addition, the *predict* method should have an optional *return_std* argument, which returns *std(Y | x)*, along with *E[Y | x]*.

If *base_estimator* is a string in {"GP", "RF", "ET", "GBRT", "DUMMY"}, a surrogate model corresponding to the relevant *X_minimize* function is created

n_initial_points: Int, default=10 Number of complete evaluation points necessary before allowing Experiments to be approximated with *base_estimator*. Any valid Experiment records found will count as initialization points. If enough Experiment records are not found, additional points will be randomly sampled

acquisition_function:{"LCB", "EI", "PI", "gp_hedge"}, default="gp_hedge" Function to minimize over the posterior distribution. Can be any of the following:

- "LCB": Lower confidence bound
- "EI": Negative expected improvement
- "PI": Negative probability of improvement
- "gp_hedge": Probabilistically choose one of the above three acquisition functions at every iteration
  - The gains *g_i* are initialized to zero
  - At every iteration,
    - Each acquisition function is optimised independently to propose a candidate point *X_i*
    - Out of all these candidate points, the next point *X_best* is chosen by *softmax(eta g_i)*
    - After fitting the surrogate model with *(X_best, y_best)*, the gains are updated such that *g_i -= mu(X_i)*

acquisition_optimizer: {"sampling", "lbfgs", "auto"}, default="auto" Method to minimize the acquisition function. The fit model is updated with the optimal value obtained by optimizing *acq_func* with *acq_optimizer*

- "sampling": *acq_func* is optimized by computing *acq_func* at *n_initial_points* randomly sampled points.
- "lbfgs": *acq_func* is optimized by
  - Randomly sampling *n_restarts_optimizer* (from *acq_optimizer_kwargs*) points
  - "lbfgs" is run for 20 iterations with these initial points to find local minima
  - The optimal of these local minima is used to update the prior
- "auto": *acq_optimizer* is configured on the basis of the *base_estimator* and the search space. If the space is *Categorical* or if the provided estimator is based on tree-models, then this is set to "sampling"

random_state: Int, 'RandomState' instance, or None, default=None Set to something other than None for reproducible results

acquisition_function_kwargs: Dict, or None, default=dict(xi=0.01, kappa=1.96) Additional arguments passed to the acquisition function

acquisition_optimizer_kwargs: Dict, or None, default=dict(n_points=10000, n_restarts_optimizer=5, n_j
Additional arguments passed to the acquisition optimizer

n_random_starts: ... Deprecated since version 3.0.0: Use *n_initial_points*, instead. Will be removed in 3.2.0

callbacks: Callable, list of callables, or None, default=[] If callable, then *callbacks(self.optimizer_result)* is called after each update to `optimizer`. If list, then each callable is called

base_estimator_kwargs: Dict, or None, default={} Additional arguments passed to *base_estimator* when it is initialized

Notes

To provide initial input points for evaluation, individual Experiments can be executed prior to instantiating an Optimization Protocol. The results of these Experiments will automatically be detected and cherished by the optimizer.

`SKOptPro` and its children in `optimization` rely heavily on the utilities provided by the *Scikit-Optimize* library, so thank you to the creators and contributors for their excellent work.

Methods

forge_experiment	Define constraints on Experiments conducted by OptPro (like hyperparameter search space)
go	Start optimization

forge_experiment(*self*, *model_initializer*, *model_init_params=None*, *model_extra_params=None*, *feature_engineer=None*, *feature_selector=None*, *notes=None*, *do_raise_repeated=True*)
Define hyperparameter search scaffold for building Experiments during optimization

OptPros use this method to guide Experiment construction behind the scenes, which is why it looks just like `hyperparameter_hunter.experiments.BaseExperiment.__init__()`. *forge_experiment* offers one major upgrade to standard Experiment initialization: it accepts hyperparameters not only as concrete values, but also as space choices – using `Real`, `Integer`, and `Categorical`. This functionality applies to the *model_init_params*, *model_extra_params* and *feature_engineer* kwargs. Any Dimensions provided to *forge_experiment* are detected by the OptPro and used to define the hyperparameter search space to be optimized

Parameters

model_initializer: Class, or functools.partial, or class instance Algorithm class used to initialize a model, such as XGBoost's *XGBRegressor*, or SKLearn's *KNeighborsClassifier*; although, there are hundreds of possibilities across many different ML libraries. *model_initializer* is expected to define at least *fit* and *predict* methods. *model_initializer* will be initialized with *model_init_params*, and its extra methods (*fit*, *predict*, etc.) will be invoked with parameters in *model_extra_params*

model_init_params: Dict, or object (optional) Dictionary of arguments given to create an instance of *model_initializer*. Any kwargs that are considered valid by the *__init__* method of *model_initializer* are valid in *model_init_params*.

In addition to providing concrete values, hyperparameters can be expressed as choices (dimensions to optimize) by using instances of `Real`, `Integer`, or

`Categorical`. Furthermore, hyperparameter choices and concrete values can be used together in *model_init_params*.

Using XGBoost's *XGBClassifier* to illustrate, the *model_init_params* kwarg of `CVExperiment` is limited to using concrete values, such as `dict(max_depth=10, learning_rate=0.1, booster="gbtree")`. This is still valid for `forge_experiment()`. However, `forge_experiment()` also allows *model_init_params* to consist entirely of space choices, such as `dict(max_depth=Integer(2, 20), learning_rate=Real(0.001, 0.5), booster=Categorical(["gbtree", "dart"]))`, or as any combination of concrete values and choices, for instance, `dict(max_depth=10, learning_rate=Real(0.001, 0.5), booster="gbtree")`.

One of the key features that makes HyperparameterHunter so magical is that **ALL** hyperparameters in the signature of *model_initializer* (and their default values) are discovered – whether or not they are explicitly given in *model_init_params*. Not only does this make Experiment result descriptions incredibly thorough, it also makes optimization smoother, more effective, and far less work for the user. For example, take LightGBM's *LGBMRegressor*, with *model_init_params*'='*dict(learning_rate=0.2)*. HyperparameterHunter recognizes that this differs from the default of 0.1. It also recognizes that *LGBMRegressor* is actually initialized with more than a dozen other hyperparameters we didn't bother mentioning, and it records their values, too. So if we want to optimize *num_leaves* tomorrow, the OptPro doesn't start from scratch. It knows that we ran an Experiment that didn't explicitly mention *num_leaves*, but its default value was 31, and it uses this information to fuel optimization – all without us having to manually keep track of tons of janky collections of hyperparameters. In fact, we really don't need to go out of our way at all. HyperparameterHunter just acts as our faithful lab assistant, keeping track of all the stuff we'd rather not worry about

model_extra_params: Dict (optional) Dictionary of extra parameters for models' non-initialization methods (like *fit*, *predict*, *predict_proba*, etc.), and for neural networks. To specify parameters for an extra method, place them in a dict named for the extra method to which the parameters should be given. For example, to call *fit* with *early_stopping_rounds*'='*5, use* *'model_extra_params*'='*dict(fit=dict(early_stopping_rounds=5))*.

Declaring hyperparameter space choices works identically to *model_init_params*, meaning that in addition to concrete values, extra parameters can be given as instances of `Real`, `Integer`, or `Categorical`. To optimize over a space in which *early_stopping_rounds* is between 3 and 9, use *model_extra_params*'='*dict(fit=dict(early_stopping_rounds=Real(3, 9)))*.

For models whose *fit* methods have a kwarg like *eval_set* (such as XGBoost's), one can use the *DatasetSentinel* attributes of the current active `Environment`, documented under its "Attributes" section and under `train_input`. An example using several DatasetSentinels can be found in HyperparameterHunter's [XGBoost Classification Example](https://github.com/HunterMcGushion/hyperparameter_hunter/blob/master/examples/xgboost_examples/classification.py)

feature_engineer: 'FeatureEngineer', or list (optional) Feature engineering/transformation/pre-processing steps to apply to datasets defined in `Environment`. If list, will be used to initialize `FeatureEngineer`, and can contain any of the following values:

1. `EngineerStep` instance
2. Function input to :class:~hyperparameter_hunter.feature_engineering.EngineerStep‘
3. `Categorical`, with *categories* comprising a selection of the previous two values (optimization only)

For important information on properly formatting *EngineerStep* functions, please see the documentation of `EngineerStep`.

To search a space optionally including an *EngineerStep*, use the *optional* kwarg of `Categorical`. This functionality is illustrated in `FeatureEngineer`. If using a *FeatureEngineer* instance to optimize *feature_engineer*, this instance cannot be used with *CVExperiment* because Experiments can't handle space choices

feature_selector: List of str, callable, or list of booleans (optional) Column names to include as input data for all provided DataFrames. If None, *feature_selector* is set to all columns in `train_dataset`, less `target_column`, and `id_column`. *feature_selector* is provided as the second argument for calls to *pandas.DataFrame.loc* when constructing datasets

notes: String (optional) Additional information about the Experiment that will be saved with the Experiment's description result file. This serves no purpose other than to facilitate saving Experiment details in a more readable format

do_raise_repeated: Boolean, default=False If True and this Experiment locates a previous Experiment's results with matching Environment and Hyperparameter Keys, a RepeatedExperimentError will be raised. Else, a warning will be logged

See also:

`hyperparameter_hunter.experiments.BaseExperiment` One-off experimentation counterpart to an OptPro's `forge_experiment()`. Internally, OptPros feed the processed arguments from *forge_experiment* to initialize Experiments. This hand-off to Experiments takes place in `_execute_experiment()`

Notes

The *auto_start* kwarg is not available here because `_execute_experiment()` sets it to False in order to check for duplicated keys before running the whole Experiment. This and *target_metric* being moved to `__init__()` are the most notable differences between calling `forge_experiment()` and instantiating `CVExperiment`

A more accurate name for this method might be something like "build_experiment_forge", since *forge_experiment* itself does not actually execute any Experiments. However, *forge_experiment* sounds cooler and much less clunky

go(*self*, *force_ready=True*)

Execute hyperparameter optimization, building an Experiment for each iteration

This method may only be invoked after invoking `forge_experiment()`, which defines experiment guidelines and search dimensions. *go* performs a few important tasks: 1) Formally setting the hyperparameter space; 2) Locating similar experiments to be used as learning material (for OptPros that suggest incumbent search points by estimating utilities using surrogate models); and 3) Actually setting off the optimization process, via `_optimization_loop()`

Parameters

force_ready: Boolean, default=False If True, `get_ready()` will be invoked even if it has already been called. This will re-initialize the hyperparameter *space* and

similar_experiments. Standard behavior is for `go()` to invoke `get_ready()`, so *force_ready* is ignored unless `get_ready()` has been manually invoked

4.4 Hyperparameter Space

class hyperparameter_hunter.space.dimensions.**Real**(*low*, *high*, *prior='uniform'*, *transform='identity'*, *name=None*)

Bases: hyperparameter_hunter.space.dimensions.NumericalDimension

Search space dimension that can assume any real value in a given range

Parameters

low: Float Lower bound (inclusive)

high: Float Upper bound (inclusive)

prior: {"uniform", "log-uniform"}, default="uniform" Distribution to use when sampling random points for this dimension. If "uniform", points are sampled uniformly between the lower and upper bounds. If "log-uniform", points are sampled uniformly between *log10(lower)* and *log10(upper)*

transform: {"identity", "normalize"}, default="identity" Transformation to apply to the original space. If "identity", the transformed space is the same as the original space. If "normalize", the transformed space is scaled between 0 and 1

name: String, tuple, or None, default=None A name associated with the dimension

Attributes

distribution: rv_generic See documentation of `_make_distribution()` or `distribution()`

transform_: String Original value passed through the *transform* kwarg - Because `transform()` exists

transformer: Transformer See documentation of `_make_transformer()` or `transformer()`

Methods

`distance`(self, a, b)	Calculate distance between two points in the dimension's bounds
`get_params`(self)	Get dict of parameters used to initialize the *Real*, or their defaults
`inverse_transform`(self, data_t)	Inverse transform samples from the warped space back to the original space
`rvs`(self[, n_samples, random_state])	Draw random samples.
`transform`(self, data)	Transform samples from the original space into a warped space

__init__(*self*, *low*, *high*, *prior='uniform'*, *transform='identity'*, *name=None*)

Search space dimension that can assume any real value in a given range

Parameters

low: Float Lower bound (inclusive)

high: Float Upper bound (inclusive)

prior: {"uniform", "log-uniform"}, default="uniform" Distribution to use when sampling random points for this dimension. If "uniform", points are sampled uniformly between the lower and upper bounds. If "log-uniform", points are sampled uniformly between *log10(lower)* and *log10(upper)*

transform: {"identity", "normalize"}, default="identity" Transformation to apply to the original space. If "identity", the transformed space is the same as the original space. If "normalize", the transformed space is scaled between 0 and 1

name: String, tuple, or None, default=None A name associated with the dimension

Attributes

distribution: rv_generic See documentation of `_make_distribution()` or `distribution()`

transform_: String Original value passed through the *transform* kwarg - Because `transform()` exists

transformer: Transformer See documentation of `_make_transformer()` or `transformer()`

---

class hyperparameter_hunter.space.dimensions.**Integer**(*low*, *high*, *transform='identity'*, *name=None*)

Bases: hyperparameter_hunter.space.dimensions.NumericalDimension

Search space dimension that can assume any integer value in a given range

Parameters

low: Int Lower bound (inclusive)

high: Int Upper bound (inclusive)

transform: {"identity", "normalize"}, default="identity" Transformation to apply to the original space. If "identity", the transformed space is the same as the original space. If "normalize", the transformed space is scaled between 0 and 1

name: String, tuple, or None, default=None A name associated with the dimension

Attributes

distribution: rv_generic See documentation of `_make_distribution()` or `distribution()`

transform_: String Original value passed through the *transform* kwarg - Because `transform()` exists

transformer: Transformer See documentation of `_make_transformer()` or `transformer()`

Methods

`distance`(self, a, b)	Calculate distance between two points in the dimension's bounds
`get_params`(self)	Get dict of parameters used to initialize the *Integer*, or their defaults

Continued on next page

Table 8 – continued from previous page

`inverse_transform`(self, data_t)	Inverse transform samples from the warped space back to the original space
`rvs`(self[, n_samples, random_state])	Draw random samples.
`transform`(self, data)	Transform samples from the original space into a warped space

`__init__` (*self*, *low*, *high*, *transform='identity'*, *name=None*)

Search space dimension that can assume any integer value in a given range

Parameters

low: Int Lower bound (inclusive)

high: Int Upper bound (inclusive)

transform: {"identity", "normalize"}, default="identity" Transformation to apply to the original space. If "identity", the transformed space is the same as the original space. If "normalize", the transformed space is scaled between 0 and 1

name: String, tuple, or None, default=None A name associated with the dimension

Attributes

distribution: rv_generic See documentation of `_make_distribution()` or `distribution()`

transform_: String Original value passed through the *transform* kwarg - Because `transform()` exists

transformer: Transformer See documentation of `_make_transformer()` or `transformer()`

---

class hyperparameter_hunter.space.dimensions.**Categorical**(*categories: list*, *prior: list = None*, *transform='onehot'*, *optional=False*, *name=None*)

Bases: `hyperparameter_hunter.space.dimensions.Dimension`

Search space dimension that can assume any categorical value in a given list

Parameters

categories: List Sequence of possible categories of shape (n_categories,)

prior: List, or None, default=None If list, prior probabilities for each category of shape (categories,). By default all categories are equally likely

transform: {"onehot", "identity"}, default="onehot" Transformation to apply to the original space. If "identity", the transformed space is the same as the original space. If "onehot", the transformed space is a one-hot encoded representation of the original space

optional: Boolean, default=False Intended for use by `FeatureEngineer` when optimizing an `EngineerStep`. Specifically, this enables searching through a space in which an *EngineerStep* either may or may not be used. This is contrary to *Categorical*'s usual function of creating a space comprising multiple *categories*. When *optional* = True, the space created will represent any of the values in *categories* either being included in the entire *FeatureEngineer* process, or being skipped entirely. Internally, a value excluded by *optional* is represented by a sentinel value that signals it should

be removed from the containing list, so *optional* will not work for choosing between a single value and None, for example

name: String, tuple, or None, default=None A name associated with the dimension

Attributes

categories: Tuple Original value passed through the *categories* kwarg, cast to a tuple. If *optional* is True, then an instance of `RejectedOptional` will be appended to *categories*

distribution: rv_generic See documentation of `_make_distribution()` or `distribution()`

optional: Boolean Original value passed through the *optional* kwarg

prior: List, or None Original value passed through the *prior* kwarg

prior_actual: List Calculated prior value, initially equivalent to `prior`, but then set to a default array if None

transform_: String Original value passed through the *transform* kwarg - Because `transform()` exists

transformer: Transformer See documentation of `_make_transformer()` or `transformer()`

Methods

`distance`(self, a, b)	Calculate distance between two points in the dimension's bounds
`get_params`(self)	Get dict of parameters used to initialize the *Categorical*, or their defaults
`inverse_transform`(self, data_t)	Inverse transform samples from the warped space back to the original space
`rvs`(self[, n_samples, random_state])	Draw random samples.
`transform`(self, data)	Transform samples from the original space into a warped space

`__init__`(*self*, *categories:list*, *prior:list=None*, *transform='onehot'*, *optional=False*, *name=None*)
Search space dimension that can assume any categorical value in a given list

Parameters

categories: List Sequence of possible categories of shape (n_categories,)

prior: List, or None, default=None If list, prior probabilities for each category of shape (categories,). By default all categories are equally likely

transform: {"onehot", "identity"}, default="onehot" Transformation to apply to the original space. If "identity", the transformed space is the same as the original space. If "onehot", the transformed space is a one-hot encoded representation of the original space

optional: Boolean, default=False Intended for use by `FeatureEngineer` when optimizing an `EngineerStep`. Specifically, this enables searching through a space in which an *EngineerStep* either may or may not be used. This is contrary to *Categorical*'s usual function of creating a space comprising multiple *categories*. When *optional* = True, the space created will represent any of the values

in *categories* either being included in the entire *FeatureEngineer* process, or being skipped entirely. Internally, a value excluded by *optional* is represented by a sentinel value that signals it should be removed from the containing list, so *optional* will not work for choosing between a single value and None, for example

name: String, tuple, or None, default=None A name associated with the dimension

Attributes

categories: Tuple Original value passed through the *categories* kwarg, cast to a tuple. If *optional* is True, then an instance of `RejectedOptional` will be appended to *categories*

distribution: rv_generic See documentation of `_make_distribution()` or `distribution()`

optional: Boolean Original value passed through the *optional* kwarg

prior: List, or None Original value passed through the *prior* kwarg

prior_actual: List Calculated prior value, initially equivalent to `prior`, but then set to a default array if None

transform_: String Original value passed through the *transform* kwarg - Because `transform()` exists

transformer: Transformer See documentation of `_make_transformer()` or `transformer()`

4.5 Feature Engineering

class hyperparameter_hunter.feature_engineering.**FeatureEngineer**(*steps=None*, *do_validate=False*, ***datasets*)

Bases: `object`

Class to organize feature engineering step callables *steps* (`EngineerStep` instances) and the datasets that the steps request and return.

Parameters

steps: List, or None, default=None List of arbitrary length, containing any of the following values:

1. `EngineerStep` instance,
2. Function to provide as input to `EngineerStep`, or
3. `Categorical`, with *categories* comprising a selection of the previous two *steps* values (optimization only)

The third value can only be used during optimization. The *feature_engineer* provided to `CVExperiment`, for example, may only contain the first two values. To search a space optionally including an *EngineerStep*, use the *optional* kwarg of `Categorical`.

See `EngineerStep` for information on properly formatted *EngineerStep* functions. Additional engineering steps may be added via `add_step()`

do_validate: Boolean, or "strict", default=False ... Experimental... Whether to validate the datasets resulting from feature engineering steps. If True, hashes of the new datasets will be compared to those of the originals to ensure they were actually modified. Results will be logged. If *do_validate* = "strict", an exception will be raised if any anomalies

are found, rather than logging a message. If *do_validate* = False, no validation will be performed

****datasets: DFDict** This is not expected to be provided on initialization and is offered primarily for debugging/testing. Mapping of datasets necessary to perform feature engineering steps

See also:

`EngineerStep` For proper formatting of non-*Categorical* values of *steps*

Notes

If *steps* does include any instances of `hyperparameter_hunter.space.dimensions.Categorical`, this *FeatureEngineer* instance will not be usable by Experiments. It can only be used by Optimization Protocols. Furthermore, the *FeatureEngineer* that the Optimization Protocol actually ends up using will not pass identity checks against the original *FeatureEngineer* that contained *Categorical* steps

Examples

```
>>> from sklearn.preprocessing import StandardScaler, MinMaxScaler,␣
↪QuantileTransformer
>>> # Define some engineer step functions to play with
>>> def s_scale(train_inputs, non_train_inputs):
...     s = StandardScaler()
...     train_inputs[train_inputs.columns] = s.fit_transform(train_inputs.values)
...     non_train_inputs[train_inputs.columns] = s.transform(non_train_inputs.
↪values)
...     return train_inputs, non_train_inputs
>>> def mm_scale(train_inputs, non_train_inputs):
...     s = MinMaxScaler()
...     train_inputs[train_inputs.columns] = s.fit_transform(train_inputs.values)
...     non_train_inputs[train_inputs.columns] = s.transform(non_train_inputs.
↪values)
...     return train_inputs, non_train_inputs
>>> def q_transform(train_targets, non_train_targets):
...     t = QuantileTransformer(output_distribution="normal")
...     train_targets[train_targets.columns] = t.fit_transform(train_targets.
↪values)
...     non_train_targets[train_targets.columns] = t.transform(non_train_targets.
↪values)
...     return train_targets, non_train_targets, t
>>> def sqr_sum(all_inputs):
...     all_inputs["square_sum"] = all_inputs.agg(
...         lambda row: np.sqrt(np.sum([np.square(_) for _ in row])), axis=
↪"columns"
...     )
...     return all_inputs
```

FeatureEngineer steps wrapped by 'EngineerStep' == raw function steps - as long as the 'EngineerStep' is using the default parameters

```
>>> # FeatureEngineer steps wrapped by `EngineerStep` == raw function steps
>>> #   ... As long as the `EngineerStep` is using the default parameters
>>> fe_0 = FeatureEngineer([sqr_sum, s_scale])
>>> fe_1 = FeatureEngineer([EngineerStep(sqr_sum), EngineerStep(s_scale)])
```

(continues on next page)

(continued from previous page)

```
>>> fe_0.steps == fe_1.steps
True
>>> fe_2 = FeatureEngineer([sqr_sum, EngineerStep(s_scale), q_transform])
```

'Categorical' can be used during optimization and placed anywhere in 'steps'. 'Categorical' can also handle either 'EngineerStep' categories or raw functions. Use the 'optional' kwarg of 'Categorical' to test some questionable steps

```
>>> fe_3 = FeatureEngineer([sqr_sum, Categorical([s_scale, mm_scale]), q_
↪transform])
>>> fe_4 = FeatureEngineer([Categorical([sqr_sum], optional=True), s_scale, q_
↪transform])
>>> fe_5 = FeatureEngineer([
...     Categorical([sqr_sum], optional=True),
...     Categorical([EngineerStep(s_scale), mm_scale]),
...     q_transform
... ])
```

`__init__`(*self, steps=None, do_validate=False, **datasets:Dict[str, pandas.core.frame.DataFrame]*)

Class to organize feature engineering step callables *steps* (`EngineerStep` instances) and the datasets that the steps request and return.

Parameters

steps: List, or None, default=None List of arbitrary length, containing any of the following values:

1. `EngineerStep` instance,
2. Function to provide as input to `EngineerStep`, or
3. `Categorical`, with *categories* comprising a selection of the previous two *steps* values (optimization only)

The third value can only be used during optimization. The *feature_engineer* provided to `CVExperiment`, for example, may only contain the first two values. To search a space optionally including an *EngineerStep*, use the *optional* kwarg of `Categorical`.

See `EngineerStep` for information on properly formatted *EngineerStep* functions. Additional engineering steps may be added via `add_step()`

do_validate: Boolean, or "strict", default=False ... Experimental... Whether to validate the datasets resulting from feature engineering steps. If True, hashes of the new datasets will be compared to those of the originals to ensure they were actually modified. Results will be logged. If *do_validate* = "strict", an exception will be raised if any anomalies are found, rather than logging a message. If *do_validate* = False, no validation will be performed

****datasets: DFDict** This is not expected to be provided on initialization and is offered primarily for debugging/testing. Mapping of datasets necessary to perform feature engineering steps

See also:

`EngineerStep` For proper formatting of non-*Categorical* values of *steps*

Notes

If *steps* does include any instances of `hyperparameter_hunter.space.dimensions.Categorical`, this *FeatureEngineer* instance will not be usable by Experiments. It can only be used by Optimization Protocols. Furthermore, the *FeatureEngineer* that the Optimization Protocol actually ends up using will not pass identity checks against the original *FeatureEngineer* that contained *Categorical* steps

Examples

```
>>> from sklearn.preprocessing import StandardScaler, MinMaxScaler,␣
↪QuantileTransformer
>>> # Define some engineer step functions to play with
>>> def s_scale(train_inputs, non_train_inputs):
...     s = StandardScaler()
...     train_inputs[train_inputs.columns] = s.fit_transform(train_inputs.
↪values)
...     non_train_inputs[train_inputs.columns] = s.transform(non_train_
↪inputs.values)
...     return train_inputs, non_train_inputs
>>> def mm_scale(train_inputs, non_train_inputs):
...     s = MinMaxScaler()
...     train_inputs[train_inputs.columns] = s.fit_transform(train_inputs.
↪values)
...     non_train_inputs[train_inputs.columns] = s.transform(non_train_
↪inputs.values)
...     return train_inputs, non_train_inputs
>>> def q_transform(train_targets, non_train_targets):
...     t = QuantileTransformer(output_distribution="normal")
...     train_targets[train_targets.columns] = t.fit_transform(train_targets.
↪values)
...     non_train_targets[train_targets.columns] = t.transform(non_train_
↪targets.values)
...     return train_targets, non_train_targets, t
>>> def sqr_sum(all_inputs):
...     all_inputs["square_sum"] = all_inputs.agg(
...         lambda row: np.sqrt(np.sum([np.square(_) for _ in row])), axis=
↪"columns"
...     )
...     return all_inputs
```

FeatureEngineer steps wrapped by 'EngineerStep' == raw function steps - as long as the 'EngineerStep' is using the default parameters

```
>>> # FeatureEngineer steps wrapped by `EngineerStep` == raw function steps
>>> #   ... As long as the `EngineerStep` is using the default parameters
>>> fe_0 = FeatureEngineer([sqr_sum, s_scale])
>>> fe_1 = FeatureEngineer([EngineerStep(sqr_sum), EngineerStep(s_scale)])
>>> fe_0.steps == fe_1.steps
True
>>> fe_2 = FeatureEngineer([sqr_sum, EngineerStep(s_scale), q_transform])
```

'Categorical' can be used during optimization and placed anywhere in 'steps'. 'Categorical' can also handle either 'EngineerStep' categories or raw functions. Use the 'optional' kwarg of 'Categorical' to test some questionable steps

```
>>> fe_3 = FeatureEngineer([sqr_sum, Categorical([s_scale, mm_scale]), q_
↪transform])
>>> fe_4 = FeatureEngineer([Categorical([sqr_sum], optional=True), s_scale,␣
↪q_transform])
>>> fe_5 = FeatureEngineer([
...     Categorical([sqr_sum], optional=True),
...     Categorical([EngineerStep(s_scale), mm_scale]),
...     q_transform
... ])
```

---

class hyperparameter_hunter.feature_engineering.**EngineerStep**(*f: Callable*, *stage=None*, *name=None*, *params=None*, *do_validate=False*)

Bases: `object`

Container for individual `FeatureEngineer` step functions

Compartmentalizes functions of singular engineer steps and allows for greater customization than a raw engineer step function

Parameters

f: Callable Feature engineering step function that requests, modifies, and returns datasets *params*

Step functions should follow these guidelines:

1. Request as input a subset of the 11 data strings listed in *params*
2. Do whatever you want to the DataFrames given as input
3. Return new DataFrame values of the input parameters in same order as requested

If performing a task like target transformation, causing predictions to be transformed, it is often desirable to inverse-transform the predictions to be of the expected form. This can easily be done by returning an extra value from *f* (after the datasets) that is either a callable, or a transformer class that was fitted during the execution of *f* and implements an *inverse_transform* method. This is the only instance in which it is acceptable for *f* to return values that don't mimic its input parameters. See the engineer function definition using SKLearn's *QuantileTransformer* in the Examples section below for an actual inverse-transformation-compatible implementation

stage: String in {"pre_cv", "intra_cv"}, or None, default=None Feature engineering stage during which the callable *f* will be given the datasets *params* to modify and return. If None, will be inferred based on *params*.

- "pre_cv" functions are applied only once in the experiment: when it starts
- "intra_cv" functions are reapplied for each fold in the cross-validation splits

If *stage* is left to be inferred, "pre_cv" will *usually* be selected. However, if any *params* (or parameters in the signature of *f*) are prefixed with "validation..." or "non_train...", then *stage* will inferred as "intra_cv". See the Notes section below for suggestions on the *stage* to use for different functions

name: String, or None, default=None Identifier for the transformation applied by this engineering step. If None, *f.__name__* will be used

params: Tuple[str], or None, default=None Dataset names requested by feature engineering step callable *f*. If None, will be inferred by parsing the signature of *f*. Must be a subset of the following 11 strings:

Input Data

1. "train_inputs"
2. "validation_inputs"
3. "holdout_inputs"
4. "test_inputs"
5. **"all_inputs"** `("train_inputs" + ["validation_inputs"] + "holdout_inputs" + "test_inputs")`
6. **"non_train_inputs"** `(["validation_inputs"] + "holdout_inputs" + "test_inputs")`

Target Data

7. "train_targets"
8. "validation_targets"
9. "holdout_targets"
10. "all_targets" `("train_targets" + ["validation_targets"] + "holdout_targets")`
11. "non_train_targets" `(["validation_targets"] + "holdout_targets")`

As an alternative to the above list, just remember that the first half of all parameter names should be one of {"train", "validation", "holdout", "test", "all", "non_train"}, and the second half should be either "inputs" or "targets". The only exception to this rule is "test_targets", which doesn't exist.

Inference of "validation" *params* is affected by *stage*. During the "pre_cv" stage, the validation dataset has not yet been created and is still a part of the train dataset. During the "intra_cv" stage, the validation dataset is created by removing a portion of the train dataset, and their values passed to *f* reflect this fact. This also means that the values of the merged ("all"/"non_train"-prefixed) datasets may or may not contain "validation" data depending on the *stage*; however, this is all handled internally, so you probably don't need to worry about it.

params may not include multiple references to the same dataset, either directly or indirectly. This means *("train_inputs", "train_inputs")* is invalid due to duplicate direct references. Less obviously, *("train_inputs", "all_inputs")* is invalid because "all_inputs" includes "train_inputs"

do_validate: Boolean, or "strict", default=False ... Experimental... Whether to validate the datasets resulting from feature engineering steps. If True, hashes of the new datasets will be compared to those of the originals to ensure they were actually modified. Results will be logged. If *do_validate* = "strict", an exception will be raised if any anomalies are found, rather than logging a message. If *do_validate* = False, no validation will be performed

See also:

`FeatureEngineer` The container for *EngineerStep* instances - *EngineerStep's should always be provided to HyperparameterHunter through a 'FeatureEngineer*

Categorical Can be used during optimization to search through a group of *EngineerStep's given as 'categories*. The *optional* kwarg of *Categorical* designates a *FeatureEngineer* step that may be one of the *EngineerStep's in 'categories*, or may be omitted entirely

get_engineering_step_stage() More information on *stage* inference and situations where overriding it may be prudent

Notes

stage: Generally, feature engineering conducted in the "pre_cv" stage should regard each sample/row as independent entities. For example, steps like converting a string day of the week to one-hot encoded columns, or imputing missing values by replacement with -1 might be conducted "pre_cv", since they are unlikely to introduce an information leakage. Conversely, steps like scaling/normalization, whose results for the data in one row are affected by the data in other rows should be performed "intra_cv" in order to recalculate the final values of the datasets for each cross validation split and avoid information leakage.

params: In the list of the 11 valid *params* strings, "test_inputs" is notably missing the "..._targets" counterpart accompanying the other datasets. The "targets" suffix is missing because test data targets are never given. Note that although "test_inputs" is still included in both "all_inputs" and "non_train_inputs", its lack of a target column means that "all_targets" and "non_train_targets" may have different lengths than their "inputs"-suffixed counterparts

Examples

```
>>> from sklearn.preprocessing import StandardScaler, QuantileTransformer
>>> def s_scale(train_inputs, non_train_inputs):
...     s = StandardScaler()
...     train_inputs[train_inputs.columns] = s.fit_transform(train_inputs.values)
...     non_train_inputs[train_inputs.columns] = s.transform(non_train_inputs.
↪values)
...     return train_inputs, non_train_inputs
>>> # Sensible parameter defaults inferred based on `f`
>>> es_0 = EngineerStep(s_scale)
>>> es_0.stage
'intra_cv'
>>> es_0.name
's_scale'
>>> es_0.params
('train_inputs', 'non_train_inputs')
>>> # Override `stage` if you want to fit your scaler on OOF data like a crazy␣
↪person
>>> es_1 = EngineerStep(s_scale, stage="pre_cv")
>>> es_1.stage
'pre_cv'
```

Watch out for multiple requests to the same data

```
>>> es_2 = EngineerStep(s_scale, params=("train_inputs", "all_inputs"))
Traceback (most recent call last):
    File "feature_engineering.py", line ? in validate_dataset_names
ValueError: Requested params include duplicate references to `train_inputs` by␣
↪way of:
   - ('all_inputs', 'train_inputs')
   - ('train_inputs',)
Each dataset may only be requested by a single param for each function
```

Error is the same if '(train_inputs, all_inputs)' is in the actual function signature

EngineerStep functions aren't just limited to transformations. Make your own features!

```
>>> def sqr_sum(all_inputs):
...     all_inputs["square_sum"] = all_inputs.agg(
...         lambda row: np.sqrt(np.sum([np.square(_) for _ in row])), axis=
↪"columns"
...     )
...     return all_inputs
>>> es_3 = EngineerStep(sqr_sum)
>>> es_3.stage
'pre_cv'
>>> es_3.name
'sqr_sum'
>>> es_3.params
('all_inputs',)
```

Inverse-transformation Implementation:

```
>>> def q_transform(train_targets, non_train_targets):
...     t = QuantileTransformer(output_distribution="normal")
...     train_targets[train_targets.columns] = t.fit_transform(train_targets.
↪values)
...     non_train_targets[train_targets.columns] = t.transform(non_train_targets.
↪values)
...     return train_targets, non_train_targets, t
>>> # Note that `train_targets` and `non_train_targets` must still be returned in␣
↪order,
>>> #   but they are followed by `t`, an instance of `QuantileTransformer` we␣
↪just fitted,
>>> #   whose `inverse_transform` method will be called on predictions
>>> es_4 = EngineerStep(q_transform)
>>> es_4.stage
'intra_cv'
>>> es_4.name
'q_transform'
>>> es_4.params
('train_targets', 'non_train_targets')
>>> # `params` does not include any returned transformers - Only data requested␣
↪as input
```

__init__(*self*, *f:Callable*, *stage=None*, *name=None*, *params=None*, *do_validate=False*)

Container for individual `FeatureEngineer` step functions

Compartmentalizes functions of singular engineer steps and allows for greater customization than a raw engineer step function

Parameters

f: Callable Feature engineering step function that requests, modifies, and returns datasets *params*

Step functions should follow these guidelines:

1. Request as input a subset of the 11 data strings listed in *params*
2. Do whatever you want to the DataFrames given as input
3. Return new DataFrame values of the input parameters in same order as requested

If performing a task like target transformation, causing predictions to be transformed, it is often desirable to inverse-transform the predictions to be of the expected form. This can easily be done by returning an extra value from *f* (after the datasets) that is either a callable, or a transformer class that was fitted during the execution of *f* and implements an *inverse_transform* method. This is the only instance in which it is acceptable for *f* to return values that don't mimic its input parameters. See the engineer function definition using SKLearn's *QuantileTransformer* in the Examples section below for an actual inverse-transformation-compatible implementation

stage: String in {"pre_cv", "intra_cv"}, or None, default=None Feature engineering stage during which the callable *f* will be given the datasets *params* to modify and return. If None, will be inferred based on *params*.

- "pre_cv" functions are applied only once in the experiment: when it starts
- "intra_cv" functions are reapplied for each fold in the cross-validation splits

If *stage* is left to be inferred, "pre_cv" will *usually* be selected. However, if any *params* (or parameters in the signature of *f*) are prefixed with "validation…" or "non_train…", then *stage* will inferred as "intra_cv". See the Notes section below for suggestions on the *stage* to use for different functions

name: String, or None, default=None Identifier for the transformation applied by this engineering step. If None, *f.__name__* will be used

params: Tuple[str], or None, default=None Dataset names requested by feature engineering step callable *f*. If None, will be inferred by parsing the signature of *f*. Must be a subset of the following 11 strings:

Input Data

1. "train_inputs"
2. "validation_inputs"
3. "holdout_inputs"
4. "test_inputs"
5. **"all_inputs"** `("train_inputs" + ["validation_inputs"] + "holdout_inputs" + "test_inputs")`
6. **"non_train_inputs"** `(["validation_inputs"] + "holdout_inputs" + "test_inputs")`

Target Data

7. "train_targets"
8. "validation_targets"
9. "holdout_targets"
10. "all_targets" `("train_targets" + ["validation_targets"] + "holdout_targets")`
11. "non_train_targets" `(["validation_targets"] + "holdout_targets")`

As an alternative to the above list, just remember that the first half of all parameter names should be one of {"train", "validation", "holdout", "test", "all", "non_train"}, and the second half should be either "inputs" or "targets". The only exception to this rule is "test_targets", which doesn't exist.

Inference of "validation" *params* is affected by *stage*. During the "pre_cv" stage, the validation dataset has not yet been created and is still a part of the train dataset. During the "intra_cv" stage, the validation dataset is created by removing a portion of the train dataset, and their values passed to *f* reflect this fact. This also means that the values of the merged ("all"/"non_train"-prefixed) datasets may or may not contain "validation" data depending on the *stage*; however, this is all handled internally, so you probably don't need to worry about it.

params may not include multiple references to the same dataset, either directly or indirectly. This means *("train_inputs", "train_inputs")* is invalid due to duplicate direct references. Less obviously, *("train_inputs", "all_inputs")* is invalid because "all_inputs" includes "train_inputs"

do_validate: Boolean, or "strict", default=False ... Experimental... Whether to validate the datasets resulting from feature engineering steps. If True, hashes of the new datasets will be compared to those of the originals to ensure they were actually modified. Results will be logged. If *do_validate* = "strict", an exception will be raised if any anomalies are found, rather than logging a message. If *do_validate* = False, no validation will be performed

See also:

`FeatureEngineer` The container for *EngineerStep* instances - *EngineerStep's should always be provided to HyperparameterHunter through a 'FeatureEngineer*

`Categorical` Can be used during optimization to search through a group of *EngineerStep's given as 'categories*. The *optional* kwarg of *Categorical* designates a *FeatureEngineer* step that may be one of the *EngineerStep's in 'categories*, or may be omitted entirely

`get_engineering_step_stage()` More information on *stage* inference and situations where overriding it may be prudent

Notes

stage: Generally, feature engineering conducted in the "pre_cv" stage should regard each sample/row as independent entities. For example, steps like converting a string day of the week to one-hot encoded columns, or imputing missing values by replacement with -1 might be conducted "pre_cv", since they are unlikely to introduce an information leakage. Conversely, steps like scaling/normalization, whose results for the data in one row are affected by the data in other rows should be performed "intra_cv" in order to recalculate the final values of the datasets for each cross validation split and avoid information leakage.

params: In the list of the 11 valid *params* strings, "test_inputs" is notably missing the "..._targets" counterpart accompanying the other datasets. The "targets" suffix is missing because test data targets are never given. Note that although "test_inputs" is still included in both "all_inputs" and "non_train_inputs", its lack of a target column means that "all_targets" and "non_train_targets" may have different lengths than their "inputs"-suffixed counterparts

Examples

```
>>> from sklearn.preprocessing import StandardScaler, QuantileTransformer
>>> def s_scale(train_inputs, non_train_inputs):
...     s = StandardScaler()
...     train_inputs[train_inputs.columns] = s.fit_transform(train_inputs.
↪values)
...     non_train_inputs[train_inputs.columns] = s.transform(non_train_
↪inputs.values)
```

(continues on next page)

(continued from previous page)

```
...     return train_inputs, non_train_inputs
>>> # Sensible parameter defaults inferred based on `f`
>>> es_0 = EngineerStep(s_scale)
>>> es_0.stage
'intra_cv'
>>> es_0.name
's_scale'
>>> es_0.params
('train_inputs', 'non_train_inputs')
>>> # Override `stage` if you want to fit your scaler on OOF data like a␣
↪crazy person
>>> es_1 = EngineerStep(s_scale, stage="pre_cv")
>>> es_1.stage
'pre_cv'
```

Watch out for multiple requests to the same data

```
>>> es_2 = EngineerStep(s_scale, params=("train_inputs", "all_inputs"))
Traceback (most recent call last):
    File "feature_engineering.py", line ? in validate_dataset_names
ValueError: Requested params include duplicate references to `train_inputs`␣
↪by way of:
   - ('all_inputs', 'train_inputs')
   - ('train_inputs',)
Each dataset may only be requested by a single param for each function
```

Error is the same if '(train_inputs, all_inputs)' is in the actual function signature

EngineerStep functions aren't just limited to transformations. Make your own features!

```
>>> def sqr_sum(all_inputs):
...     all_inputs["square_sum"] = all_inputs.agg(
...         lambda row: np.sqrt(np.sum([np.square(_) for _ in row])), axis=
↪"columns"
...     )
...     return all_inputs
>>> es_3 = EngineerStep(sqr_sum)
>>> es_3.stage
'pre_cv'
>>> es_3.name
'sqr_sum'
>>> es_3.params
('all_inputs',)
```

Inverse-transformation Implementation:

```
>>> def q_transform(train_targets, non_train_targets):
...     t = QuantileTransformer(output_distribution="normal")
...     train_targets[train_targets.columns] = t.fit_transform(train_targets.
↪values)
...     non_train_targets[train_targets.columns] = t.transform(non_train_
↪targets.values)
...     return train_targets, non_train_targets, t
>>> # Note that `train_targets` and `non_train_targets` must still be␣
↪returned in order,
>>> #   but they are followed by `t`, an instance of `QuantileTransformer`␣
↪we just fitted,
```

(continues on next page)

(continued from previous page)

```
>>> #   whose `inverse_transform` method will be called on predictions
>>> es_4 = EngineerStep(q_transform)
>>> es_4.stage
'intra_cv'
>>> es_4.name
'q_transform'
>>> es_4.params
('train_targets', 'non_train_targets')
>>> # `params` does not include any returned transformers - Only data␣
↪requested as input
```

4.6 Extras

hyperparameter_hunter.callbacks.bases.**lambda_callback**(*on_exp_start=None*, *on_exp_end=None*, *on_rep_start=None*, *on_rep_end=None*, *on_fold_start=None*, *on_fold_end=None*, *on_run_start=None*, *on_run_end=None*, *agg_name=None*, *do_reshape_aggs=True*, *method_agg_keys=False*, *on_experiment_start=<object object at 0x7fb828ce0be0>*, *on_experiment_end=<object object at 0x7fb828ce0be0>*, *on_repetition_start=<object object at 0x7fb828ce0be0>*, *on_repetition_end=<object object at 0x7fb828ce0be0>*)

Utility for creating custom callbacks to be declared by `Environment` and used by Experiments. The callable "on_<...>_<start/end>" parameters provided will receive as input whichever attributes of the Experiment are included in the signature of the given callable. If ***kwargs* is given in the callable's signature, a dict of all of the Experiment's attributes will be provided. This can be helpful for trying to figure out how to build a custom callback, but should not be used unless absolutely necessary. If the Experiment does not have an attribute specified in the callable's signature, the following placeholder will be given: "INVALID KWARG"

Parameters

on_exp_start: Callable, or None, default=None Callable that receives Experiment's values for parameters in the signature at Experiment start

on_exp_end: Callable, or None, default=None Callable that receives Experiment's values for parameters in the signature at Experiment end

on_rep_start: Callable, or None, default=None Callable that receives Experiment's values for parameters in the signature at repetition start

on_rep_end: Callable, or None, default=None Callable that receives Experiment's values for parameters in the signature at repetition end

on_fold_start: Callable, or None, default=None Callable that receives Experiment's values for parameters in the signature at fold start

on_fold_end: Callable, or None, default=None Callable that receives Experiment's values for parameters in the signature at fold end

on_run_start: Callable, or None, default=None Callable that receives Experiment's values for parameters in the signature at run start

on_run_end: Callable, or None, default=None Callable that receives Experiment's values for parameters in the signature at run end

agg_name: Str, default=uuid.uuid4 This parameter is only used if the callables are behaving like AggregatorCallbacks by returning values (see the "Notes" section below for details on this). If the callables do return values, they will be stored under a key named ("_" + *agg_name*) in a dict in `hyperparameter_hunter.experiments.BaseExperiment.stat_aggregates`. The purpose of this parameter is to make it easier to understand an Experiment's description file, as *agg_name* will default to a UUID if it is not given

do_reshape_aggs: Boolean, default=True Whether to reshape the aggregated values to reflect the nested repetitions/folds/runs structure used for other aggregated values. If False, lists of aggregated values are left in their original shapes. This parameter is only used if the callables are behaving like AggregatorCallbacks (see the "Notes" section below and *agg_name* for details on this)

method_agg_keys: Boolean, default=False If True, the aggregate keys for the items added to the dict at *agg_name* are equivalent to the names of the "on_<...>_<start/end>" pseudo-methods whose values are being aggregated. In other words, the pool of all possible aggregate keys goes from ["runs", "folds", "reps", "final"] to the names of the eight "on_<...>_<start/end>" kwargs of `lambda_callback()`. See the "Notes" section below for further details and a rough outline

on_experiment_start: ... Deprecated since version 3.0.0: Renamed to *on_exp_start*. Will be removed in 3.2.0

on_experiment_end: ... Deprecated since version 3.0.0: Renamed to *on_exp_end*. Will be removed in 3.2.0

on_repetition_start: ... Deprecated since version 3.0.0: Renamed to *on_rep_start*. Will be removed in 3.2.0

on_repetition_end: ... Deprecated since version 3.0.0: Renamed to *on_rep_end*. Will be removed in 3.2.0

Returns

LambdaCallback: `LambdaCallback` Uninitialized class, whose methods are the callables of the corresponding "on..." kwarg

Notes

For all of the "on_<...>_<start/end>" callables provided as input to *lambda_callback*, consider the following guidelines (for example function "f", which can represent any of the callables):

- All input parameters in the signature of "f" are attributes of the Experiment being executed
  - If "**kwargs" is a parameter, a dict of all the Experiment's attributes will be provided
- "f" will be treated as a method of a parent class of the Experiment
  - Take care when modifying attributes, as changes are reflected in the Experiment itself
- If "f" returns something, it will automatically behave like an AggregatorCallback (see `hyperparameter_hunter.callbacks.aggregators`). Specifically, the following will occur:

  - A new key (named by *agg_name* if given, else a UUID) with a dict value is added to `hyperparameter_hunter.experiments.BaseExperiment.stat_aggregates`
    - This new dict can have up to four keys: "runs" (list), "folds" (list), "reps" (list), and "final" (object)
  - If "f" is an "on_run..." function, the returned value is appended to the "runs" list in the new dict
  - Similarly, if "f" is an "on_fold..." or "on_rep..." function, the returned value is appended to the "folds", or "reps" list, respectively
  - If "f" is an "on_exp..." function, the "final" key in the new dict is set to the returned value
  - If values were aggregated in the aforementioned manner, the lists of collected values will be reshaped according to runs/folds/reps on Experiment end
  - The aggregated values will be saved in the Experiment's description file
    - This is because `hyperparameter_hunter.experiments.BaseExperiment.stat_aggregates` is saved in its entirety

What follows is a rough outline of the structure produced when using an aggregator-like callback that automatically populates `experiments.BaseExperiment.stat_aggregates` with results of the functions used as arguments to `lambda_callback()`:

```
BaseExperiment.stat_aggregates = dict(
    ...,
    <`agg_name`>=dict(
        <agg_key "runs">  = [...],
        <agg_key "folds"> = [...],
        <agg_key "reps">  = [...],
        <agg_key "final"> = object(),
        ...
    ),
    ...
)
```

In the above outline, the actual *agg_key's included in the dict at 'agg_name* depend on which "on_<...>_<start/end>" callables are behaving like aggregators. For example, if neither *on_run_start* nor *on_run_end* explicitly returns something, then the "runs" *agg_key* is not included in the *agg_name* dict. Similarly, if, for example, neither *on_exp_start* nor *on_exp_end* is provided, then the "final" *agg_key* is not included. If *method_agg_keys=True*, then the agg keys used in the dict are modified to be named after the method called. For example, if *method_agg_keys=True* and *on_fold_start* and *on_fold_end* are both callables returning values to be aggregated, then the *agg_key's used for each will be "on_fold_start" and "on_fold_end", respectively. In this example, if 'method_agg_keys=False* (default) and *do_reshape_aggs=False*, then the single "folds" *agg_key* would contain the combined contents returned by both methods in the order in which they were returned

For examples using *lambda_callback* to create custom callbacks, see `hyperparameter_hunter.callbacks.recipes`

Examples

```
>>> from hyperparameter_hunter.environment import Environment
>>> def printer_helper(_rep, _fold, _run, last_evaluation_results):
...     print(f"{_rep}.{_fold}.{_run}   {last_evaluation_results}")
>>> my_lambda_callback = lambda_callback(
...     on_exp_end=printer_helper,
...     on_rep_end=printer_helper,
...     on_fold_end=printer_helper,
```

(continues on next page)

(continued from previous page)

```
...     on_run_end=printer_helper,
... )
... # env = Environment(
... #     train_dataset="i am a dataset",
... #     results_path="path/to/HyperparameterHunterAssets",
... #     metrics=["roc_auc_score"],
... #     experiment_callbacks=[my_lambda_callback]
... # )
... # ... Now execute an Experiment, or an Optimization Protocol...
```

See `hyperparameter_hunter.examples.lambda_callback_example` for more information

4.7 Indices and tables

- genindex
- modindex
- search

CHAPTER

FIVE

COMPLETE HYPERPARAMETERHUNTER API

This section exposes the complete HyperparameterHunter API.

- genindex
- modindex
- search

CHAPTER

SIX

FILE STRUCTURE OVERVIEW

This section is an overview of the result file structure created and updated when `Experiment`s are completed.

6.1 HyperparameterHunterAssets/

- Contains one file (**'Heartbeat.log'**), and four subdirectories (**'Experiments/'**, **'KeyAttributeLookup/'**, **'Leaderboards/'**, and **'TestedKeys/'**).
- **'Heartbeat.log'** is the log file for the current/most recently executed `Experiment`. It will look very much like the printed output of `CVExperiment`, with some additional debug messages thrown in. When the `Experiment` is completed, a copy of this file is saved as the `Experiment`'s own Heartbeat file, which will be discussed below.

6.1.1 /Experiments/

Contains up to six different subdirectories. The files contained in each of the subdirectories all follow the same naming convention: they are named after the `Experiment`'s randomly-generated UUID. The subdirectories are as follows:

---

1) /Descriptions/

Contains a .json file for each completed `Experiment`, describing all critical (and some extra) information about the `Experiment`'s results. Such information includes, but is certainly not limited to: keys, algorithm/library name, final scores, model_initializer hash, hyperparameters, cross experiment parameters, breakdown of times elapsed, start/end datetimes, breakdown of evaluations over runs/folds/reps, source script name, platform, and additional notes. This file is meant to give you all the details you need regarding an `Experiment`'s results and the conditions that led to those results.

2) /Heartbeats/

Contains a .log file for each completed `Experiment` that is created by copying the aforementioned **'HyperparameterHunterAssets/Heartbeat.log'** file. This file is meant to give you a record of what exactly the `Experiment` was experiencing along the course of its existence. This can be useful if you need to verify questionable results, or check for error/warning/debug messages that might not have been noticed before.

3) /PredictionsOOF/

Contains a .csv file for each completed `Experiment`, containing out-of-fold predictions for the `train_dataset` provided to `Environment`. If `Environment` is given a `runs` value > 1, or if a repeated cross-validation scheme is provided (like sklearn's `RepeatedKFold` or `RepeatedStratifiedKFold`), then OOF predictions will be averaged according to the number of runs and repetitions. An extended discussion of this file's uses probably isn't necessary, but just some of the things you might want it for include: testing the performance of ensembled models via their prediction files, or calculating other metric values, if, for example, we wanted an F1 score, or simple accuracy after the `Experiment` had finished, instead of the ROC-AUC score we told the `Environment` we wanted. Note that if we knew ahead of time we wanted all three of these metrics, we could have easily given the `Environment` all three (or any other number of metrics) at its initialization. See the 'custom_metrics_example.py' example script for more details on advanced metrics specifications.

4) /PredictionsHoldout/

This subdirectory's file structure is pretty much identical to **'PredictionsOOF/'** and is populated when we use `Environment`'s `holdout_dataset` kwarg to provide a holdout DataFrame, a filepath to one, or a callable to extract a `holdout_dataset` from our `train_dataset`. Additionally, if a `holdout_dataset` is provided, the provided metrics will be calculated for it as well (unless you tell it otherwise).

5) /PredictionsTest/

This subdirectory is much like **'PredictionsOOF/'** and **'PredictionsHoldout/'**. It is populated when we use `Environment`'s `test_dataset` kwarg to provide a test DataFrame, or a filepath to one. It may be worth noting that the major difference between `test_dataset` and its counterparts (`train_dataset`, and `holdout_dataset`) is that test predictions are not evaluated because it is the nature of the `test_dataset` to have unknown targets.

6) /ScriptBackups/

Contains a .py file for each completed `Experiment` that is an exact copy of the script executed that led to the instantiation of the `Experiment`. These files exist primarily to assist in "oh shit" moments where you have no idea how to recreate an `Experiment`. 'script_backup' is blacklisted by default when executing a hyperparameter `OptimizationProtocol`, as all experiments would be created by the same file.

6.1.2 /KeyAttributeLookup/

- This directory stores any complex-typed `Environment` parameters and hyperparameters, as well as the hashes with which those complex objects are associated.
- Specifically, this directory is concerned with any python classes, or callables, or DataFrames you may provide, and will create a the appropriate file or directory to properly store the object.
  - If a class is provided (as is the case with `cv_type`, and `model_initializer`), the Shelve and Dill libraries are used to pickle a copy of the class, linked to the class's hash as its key.
  - If a defined function, or a lambda is provided (as is the case with `prediction_formatter`, which is an optional `Environment` kwarg), a .json file entry is created linking the callable's hash to its source code saved as a string, which can be recreated using Python's exec function.

  - If a Pandas DataFrame is provided (as is the case with `train_dataset`, and its holdout and test counterparts), the process is slightly different. Rather than naming a file after the complex-typed attribute (as in the first two types), a directory is named after the attribute, hence the **'HyperparameterHunterAssets/KeyAttributeLookup/train_dataset/'** directory. Then, .csv files are added to the corresponding directory, which are named after the DataFrame's hash, and which contain the DataFrame itself.

- Entries in the **'KeyAttributeLookup/'** directory are created on an as-needed basis.
  - This means that you may see entries named after attributes other than those shown in this example along the course of your own project.
  - They are created whenever `Environment`s or `Experiment`s are provided arguments too complex to neatly display in the `Experiment`'s **'Descriptions/'** entry file.
  - Some other complex attributes you may come across that are given **'KeyAttributeLookup/'** entries include: custom metrics provided via `Environment`'s `metrics` and `metrics_params` kwargs, and Keras Neural Network `callbacks` and `build_fn`s.

6.1.3 /Leaderboards/

- At the time of this documentation's writing, this directory contains only one file: **'GlobalLeaderboard.csv'**; although, more are on the way to assist you in comparing the performance of different `Experiment`s, and they should be similar in structure to this one.
- **'GlobalLeaderboard.csv'** is a DataFrame containing one row for every completed `Experiment`
- It has a column for every final metric evaluation performed, as well as the following columns: 'experiment_id', 'hyperparameter_key', 'cross_experiment_key', and 'algorithm_name'
- Rows are sorted in descending order according to the first metric provided, and will prioritize OOF evaluations before holdout evaluations if both are given.
- If an `Experiment` does not have a particular evaluation, the `Experiment` row's value for that column will be null.
  - This can happen if new metrics are specified, which were not recorded for earlier experiments, or if a `holdout_dataset` is provided to later `Experiment`s that earlier ones did not have.

6.1.4 /TestedKeys/

- This directory contains a .json file named for every unique `cross_experiment_key` encountered.
- Each .json file contains a dictionary, whose keys are the `hyperparameter_key`s that have been tested in conjunction with the `cross_experiment_key` for which the containing file is named.
- The value of each of these keys is a list of strings, in which each string is an `experiment_id`, denoting an `Experiment` that was conducted with the hyperparameters symbolized by that list's key, and an `Environment`, whose cross-experiment parameters are symbolized by the name of the containing file.
  - The values are lists in order to accommodate `Experiment`s that are intentionally duplicated.

CHAPTER

SEVEN

HYPERPARAMETERHUNTER EXAMPLES

This section provides links to example scripts that may be helpful to better understand how HyperparameterHunter works with some libraries, as well as some of HyperparameterHunter's more advanced features.

7.1 Getting Started

- Simple Experiment
- Simple Hyperparameter Optimization

7.2 Different Libraries

- CatBoost
- Keras
- LightGBM
- Scikit-Learn
- XGBoost
- rgf_python

7.3 Advanced Features

- Holdout/Test Datasets
- do_full_save
- environment_params_path
- lambda_callback

CHAPTER

EIGHT

HYPERPARAMETERHUNTER LIBRARY COMPATIBILITY

This section lists libraries that have been tested with HyperparameterHunter and briefly outlines some works in progress.

8.1 Tested and Compatible

- CatBoost
- Keras
- LightGBM
- Scikit-Learn
- XGBoost
- rgf_python

8.2 Support On the Way

- PyTorch/Skorch
- TensorFlow
- Boruta
- Imbalanced-Learn

8.3 Not Yet Compatible

- TPOT
  - After admittedly minimal testing, problems arose due to the fact that TPOT implements its own cross-validation scheme
  - This resulted in (probably unexpected) nested cross validation, and extremely long execution times

8.4 Notes

- If you don't see the one of your favorite libraries listed above, and you want to do something about that, let us know!
- See HyperparameterHunter's **'examples/'** directory for help on getting started with compatible libraries
- Improved support for hyperparameter tuning with Keras is on the way!

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search