hyperparameter_hunter Documentation

Release 3.0.0

Hunter McGushion

Jan 20, 2021

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=True, 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,
    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

4.2 Experimentation

4.3 Hyperparameter Optimization

4.4 Hyperparameter Space

class hyperparameter_hunter.space.dimensions.**Real**(*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()`

Methods

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

`__init__`(*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*)

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

`__init__`(*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: Optional[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()`

Methods

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

continues on next page

Table 3 – continued from previous page

`transform`(data)	Transform samples from the original space into a warped space

`__init__`(*categories: list*, *prior: Optional[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: 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
... ])
```

__init__(*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()
```

(continues on next page)

(continued from previous page)

```
...     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*)

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)
```

(continues on next page)

(continued from previous page)

```
...     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__ (*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)
...     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')
```

(continues on next page)

(continued from previous page)

```
   - ('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
```

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>*, *on_experiment_end=<object object>*, *on_repetition_start=<object object>*, *on_repetition_end=<object object>*)

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,
...     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