GeneticPy

Brandon Schabell

Mar 28, 2022

CONTENTS:

Get started by *installing* GeneticPy.

CHAPTER

ONE

INSTALLATION

Installing GeneticPy can be done by simply using pip:

```
pip install geneticpy
```

GeneticPy requires Python 3.7+ and only a few common Python packages:

- `numpy`
- `scikit-learn`
- `tqdm`

CHAPTER

TWO

USAGE

GeneticPy can be used through an `optimize` function or using a `GeneticSearchCV` class, which behaves similarly to a `scikit-learn` `GridSearchCV` class.

geneticpy.optimize(fn, param_space[, size, ...])	The `optimize` function is used to run the genetic algorithm over the specified parameter space in an effort to minimize (or maximize if `maximize_fn=True`) the specified loss[reward] function, `fn(params)`.
geneticpy.GeneticSearchCV(estimator, ...[, ...])	The GeneticSearchCV class can be used as a drop-in replacement for Scikit-Learn's GridSearchCV.

2.1 geneticpy.optimize

geneticpy.**optimize**(*fn: callable*, *param_space: Dict[str, geneticpy.distributions.distribution_base.DistributionBase]*, *size: int = 100*, *generation_count: int = 10*, *percentage_to_randomly_spawn: float = 0.1*, *mutate_chance: float = 0.35*, *retain_percentage: float = 0.5*, *maximize_fn: bool = False*, *target: Optional[float] = None*, *verbose: bool = False*, *seed: Optional[int] = None*) → Dict[str, Union[float, Dict[str, Any]]]

The `optimize` function is used to run the genetic algorithm over the specified parameter space in an effort to minimize (or maximize if `maximize_fn=True`) the specified loss[reward] function, `fn(params)`.

Parameters

fn: callable A callable function that can be either synchronous or asynchronous. This function is expected to take a dictionary of parameters as input and return a float. (e.g. `def fn(params: dict) -> float`)

param_space: Dict[str, DistributionBase] A dictionary of parameters to tune. Keys should be a string representing the name of the variable, and values should be geneticpy distributions.

size: int, default = 100 The number of iterations to attempt with every generation.

generation_count: int, default = 10 The number of generations to use during the optimization.

percentage_to_randomly_spawn: float, default = 0.1 The percentage of iterations within each generation that will be created with random initial values.

mutate_chance: float, default = 0.35 The percentage of iterations within each generation that will be filled with parameters mutated from top performing iterations in the previous generation.

retain_percentage: float, default = 0.5 The percentage of iterations that will be kept at the end of each generation. The best performing iterations, as determined by the `fn` function will be kept.

maximize_fn: bool, default = False If `True`, the specified `fn` function will be treated as a reward function, otherwise the `fn` function will be treated as a loss function.

target: Optional[float], default = None If specified, the algorithm will stop searching once a parameter set resulting in a loss/reward better than or equal to the specified value has been found.

verbose: bool, default = False If True, a progress bar will be displayed.

seed: Optional[int], default = None If specified, the random number generators used to generate new parameter sets will be seeded, resulting in a deterministic and repeatable result.

Returns

Dict[str, Union[float, Dict[str, Any]]]: A dictionary containing `top_params`, `top_score`, and `total_time` keys:

`top_params`: A dictionary containing the top parameters from the optimization.

`top_score`: The score of the `top_params` parameter set as determined by the specified `fn` function.

`total_time`: The total time in seconds that it took to run the optimization.

Examples

```
import geneticpy

def loss_function(params):
    if params['type'] == 'add':
        return params['x'] + params['y']
    elif params['type'] == 'multiply':
        return params['x'] * params['y']

param_space = {'type': geneticpy.ChoiceDistribution(choice_list=['add', 'multiply
↪']),
               'x': geneticpy.UniformDistribution(low=5, high=10, q=1),
               'y': geneticpy.GaussianDistribution(mean=0, standard_deviation=1,␣
↪low=-1, high=1)}

results = geneticpy.optimize(loss_function, param_space)
print(results)

{'top_params': {'type': 'multiply', 'x': 10, 'y': -1},
 'top_score': -10,
 'total_time': 0.1290111541748047}
```

2.2 geneticpy.GeneticSearchCV

class geneticpy.**GeneticSearchCV**(*estimator: sklearn.base.BaseEstimator*, *param_distributions: Dict[str, geneticpy.distributions.distribution_base.DistributionBase]*, ***, *scoring: Optional[Union[str, callable]] = None*, *refit: bool = True*, *cv: Optional[Union[int, Generator, Iterable]] = None*, *verbose: bool = False*, *random_state: Optional[int] = None*, *population_size: int = 50*, *generation_count: int = 10*)

The GeneticSearchCV class can be used as a drop-in replacement for Scikit-Learn's GridSearchCV. This allows for faster and more complete optimization of your hyperparameters when using Scikit-Learn estimators and/or pipelines.

__init__(*estimator: sklearn.base.BaseEstimator*, *param_distributions: Dict[str, geneticpy.distributions.distribution_base.DistributionBase]*, ***, *scoring: Optional[Union[str, callable]] = None*, *refit: bool = True*, *cv: Optional[Union[int, Generator, Iterable]] = None*, *verbose: bool = False*, *random_state: Optional[int] = None*, *population_size: int = 50*, *generation_count: int = 10*)

Parameters

estimator: BaseEstimator The estimator that will be used for fitting and predicting subsequently supplied data.

param_distributions: Dict[str, DistributionBase] A dictionary of parameters to tune. Keys should be a string representing the name of the variable, and values should be geneticpy distributions.

scoring: Optional[Union[str, callable]], default = None Strategy to evaluate the performance of the cross-validated model on the test set.

If *scoring* represents a single score, one can use:

- a single string;
- a callable that returns a single value.

refit: bool, default = True If True, the model will be refit with the best parameters following the hyperparameter tuning.

cv: Optional[Union[int, Generator, Iterable]], default=None Determines the cross-validation splitting strategy. Possible inputs for cv are: - None, to use the default 5-fold cross validation, - integer, to specify the number of folds. - *CV splitter* - An iterable yielding (train, test) splits as arrays of indices.

verbose: bool, default = False If True, a progress bar will be displayed.

random_state: Optional[int], default = None If specified, the random number generators used to generate new parameter sets will be seeded, resulting in a deterministic and repeatable result.

population_size: int, default = 50 The number of iterations to attempt with every generation.

generation_count: int, default = 10 The number of generations to use during the optimization.

Examples

```
from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

from geneticpy import GeneticSearchCV, ChoiceDistribution,␣
↪LogNormalDistribution, UniformDistribution

# Define a pipeline to search for the best combination of PCA truncation
# and classifier regularization.
pca = PCA()
# set the tolerance to a large value to make the example faster
logistic = LogisticRegression(max_iter=10000, tol=0.1, solver='saga')
pipe = Pipeline(steps=[('pca', pca), ('logistic', logistic)])

X_digits, y_digits = datasets.load_digits(return_X_y=True)

# Parameters of pipelines can be set using '__' separated parameter names:
param_grid = {
    'pca__n_components': UniformDistribution(low=5, high=64, q=1),
    'logistic__C': LogNormalDistribution(mean=1, sigma=0.5, low=0.001, high=2),
    'logistic__penalty': ChoiceDistribution(choice_list=['l1', 'l2'])
}
search = GeneticSearchCV(pipe, param_grid)
search.fit(X_digits, y_digits)
print(f"Best parameter (CV score={search.best_score_}):")
print(search.best_params_)
```

Methods

__init__(estimator, param_distributions, *)	**Parameters**
decision_function(X)	Call decision_function on the estimator with the best found parameters.
fit(X[, y, groups])	Run fit with all sets of parameters.
inverse_transform(Xt)	Call inverse_transform on the estimator with the best found params.
predict(X)	Call predict on the estimator with the best found parameters.
predict_log_proba(X)	Call predict_log_proba on the estimator with the best found parameters.
predict_proba(X)	Call predict_proba on the estimator with the best found parameters.
score(X[, y])	Returns the score on the given data, if the estimator has been refit.
transform(X)	Call transform on the estimator with the best found parameters.

Attributes

classes_
n_features_in_

INDEX

Symbols

G

O