AutoGL

Release v0.1.0

THUMNLab/aglteam

Dec 23, 2020

TUTORIAL

1 AutoGL **1**

2 Installation **3**
- 2.1 Requirements . . . 3
- 2.2 Installation . . . 3

3 Modules **5**
- 3.1 Quick Start . . . 5
- 3.2 AutoGL Dataset . . . 7
- 3.3 AutoGL Feature Engineering . . . 9
- 3.4 AutoGL Model . . . 11
- 3.5 AutoGL Trainer . . . 12
- 3.6 Hyper Parameter Optimization . . . 13
- 3.7 Ensemble . . . 16
- 3.8 AutoGL Solver . . . 17
- 3.9 data . . . 22
- 3.10 dataset . . . 26
- 3.11 module . . . 27
- 3.12 solver . . . 29

4 Indices and tables **31**

Python Module Index **33**

Index **35**

CHAPTER

ONE

AUTOGL

Actively under development by @THUMNLab

AutoGL is developed for researchers and developers to quickly conduct autoML on the graph datasets & tasks. See our documentation for detailed information!

The workflow below shows the overall framework of AutoGL.

AutoGL uses `AutoGL Dataset` to maintain datasets for graph-based machine learning, which is based on the dataset in PyTorch Geometric with some support added to corporate with the auto solver framework.

Different graph-based machine learning tasks are solved by different `AutoGL Solvers` , which make use of four main modules to automatically solve given tasks, namely `Auto Feature Engineer`, `Auto Model`, `HyperParameter Optimization`, and `Auto Ensemble`.

CHAPTER

TWO

INSTALLATION

2.1 Requirements

Please make sure you meet the following requirements before installing AutoGL.

1. Python >= 3.6.0
2. PyTorch (>=1.5.1)

   see PyTorch for installation.

3. PyTorch Geometric

   see PyTorch Geometric for installation.

2.2 Installation

2.2.1 Install from pip & conda

Run the following command to install this package through pip.

```
pip install auto-graph-learning
```

2.2.2 Install from source

Run the following command to install this package from the source.

```
git clone https://github.com/THUMNLab/AutoGL.git
cd AutoGL
python setup.py install
```

2.2.3 Install for development

If you are a developer of the AutoGL project, please use the following command to create a soft link, then you can modify the local package without installation again.

```
pip install -e .
```

CHAPTER

THREE

MODULES

In AutoGL, the tasks are solved by corresponding learners, which in general do the following things:

1. Preprocess and feature engineer the given datasets. This is done by the module named **auto feature engineer**, which can automatically add/delete useful/useless attributes in the given datasets. Some topological features may also be extracted & combined to form stronger features for current tasks.
2. Automatically train and tune popular models specified by users. This is done by modules named **auto model** and **hyperparameter optimization**. In the auto model, several commonly used graph deep models are provided, together with their hyperparameter spaces. These kinds of models can be tuned using **hyperparameter optimization** module to find the best hyperparameter for the current task.
3. Find the best way to ensemble models found and trained in the last step. This is done by the module named **auto ensemble**. The suitable models available are ensembled here to form a more powerful learner.

3.1 Quick Start

This tutorial will help you quickly go through the concepts and usages of important classes in AutoGL. In this tutorial, you will conduct a quick auto graph learning on dataset Cora.

3.1.1 AutoGL Learning

Based on the concept of autoML, auto graph learning aims at **automatically** solve tasks with data represented by graphs. Unlike conventional learning frameworks, auto graph learning, like autoML, does not need humans inside the experiment loop. You only need to provide the datasets and tasks to the AutoGL solver. This framework will automatically find suitable methods and hyperparameters for you.

The diagram below describes the workflow of AutoGL framework.

To reach the aim of autoML, our proposed auto graph learning framework is organized as follows. We have `dataset` to maintain the graph datasets given by users. A `solver` object needs to be built for specifying the target tasks. Inside `solver`, there are four submodules to help complete the auto graph tasks, namely `auto feature engineer`, `auto model`, `hyperparameter optimization` and `auto ensemble`, which will automatically preprocess/enhance your data, choose and optimize deep models and ensemble them in the best way for you.

Let's say you want to conduct an auto graph learning on dataset `Cora`. First, you can easily get the `Cora` dataset using the `dataset` module:

```
from autogl.datasets import build_dataset_from_name
cora_dataset = build_dataset_from_name('cora')
```

The dataset will be automatically downloaded for you. Please refer to *AutoGL Dataset* or *dataset* for more details of dataset constructions, available datasets, add local datasets, etc.

After deriving the dataset, you can build a `node classification solver` to handle auto training process:

```
import torch
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
from autogl.solver import AutoNodeClassifier
solver = AutoNodeClassifier(
    feature_module='deepgl',
    graph_models=['gcn', 'gat'],
    hpo_module='anneal',
    ensemble_module='voting',
    device=device
)
```

In this way, we build a `node classification solver`, which will use `deepgl` as its feature engineer, and use `anneal` hyperparameter optimizer to optimize the given three models `['gcn','gat']`. The derived models will then be ensembled using `voting` ensembler. Please refer to the corresponding tutorials or documentation to see the definition and usages of available submodules.

Then, you can fit the solver and then check the leaderboard:

```
solver.fit(cora_dataset, time_limit=3600)
solver.get_leaderboard().show()
```

The `time_limit` is set to 3600 so that the whole auto graph process will not exceed 1 hour. `solver.show()` will present the models maintained by `solver`, with their performances on the validation dataset.

Then, you can make the predictions and evaluate the results using the evaluation functions provided:

```
from autogl.module.train import Acc
predicted = solver.predict_proba()
print('Test accuracy: ', Acc.evaluate(predicted,
    cora_dataset.data.y[cora_dataset.data.test_mask].cpu().numpy()))
```

Note: You don't need to pass the `cora_dataset` again when predicting, since the dataset is **remembered** by the `solver` and will be reused when no dataset is passed at predicting. However, you can also pass a new dataset when predicting, and the new dataset will be used instead of the remembered one. Please refer to *AutoGL Solver* or *solver* for more details.

3.2 AutoGL Dataset

We import the module of datasets from *CogDL* and *PyTorch Geometric* and add support for datasets from *OGB*. One can refer to the usage of creating and building datasets via the tutorial of CogDL, PyTorch Geometric, and OGB.

3.2.1 Supporting datasets

AutoGL now supports the following benchmarks for different tasks:

Semi-supervised node classification: Cora, Citeseer, Pubmed, Amazon Computers*, Amazon Photo*, Coauthor CS*, Coauthor Physics*, Reddit *: using *utils.random_splits_mask_class* for splitting dataset is recommended.). For detailed information for supporting datasets, please kindly refer to PyTorch Geometric Dataset.

Dataset	PyG	CogDL	x	y	edge_index	edge_attr | train/val/test node		train/val/test mask
Cora	✓		✓	✓	✓	✓		✓
Citeseer	✓		✓	✓	✓	✓		✓
Pubmed	✓		✓	✓	✓	✓		✓
Amazon Computers	✓		✓	✓	✓	✓		
Amazon Photo	✓		✓	✓	✓	✓		
Coauthor CS	✓		✓	✓	✓	✓		
Coauthor Physics	✓		✓	✓	✓	✓		
Reddit	✓		✓	✓	✓	✓		✓

Graph classification: MUTAG, IMDB-B, IMDB-M, PROTEINS, COLLAB

Dataset	PyG	CogDL	x	y	edge_index	edge_attr
MUTAG	✓		✓	✓	✓	✓
IMDB-B	✓			✓	✓	
IMDB-M	✓			✓	✓	
PROTEINS	✓		✓	✓	✓	
COLLAB	✓			✓	✓	

TODO: Supporting all datasets from *PyTorch Geometric*.

3.2.2 OGB datasets

AutoGL also supports the popular benchmark on *OGB* for node classification and graph classification tasks. For the summary of *OGB* datasets, please kindly refer to the their docs.

Since the loss and evaluation metric used for *OGB* datasets vary among different tasks, we also add *string* properties of datasets for identification:

Dataset	dataset.metric	datasets.loss
ogbn-products	Accuracy	nll_loss
ogbn-proteins	ROC-AUC	BCEWithLogitsLoss
ogbn-arxiv	Accuracy	nll_loss
ogbn-papers100M	Accuracy	nll_loss
ogbn-mag	Accuracy	nll_loss
ogbg-molhiv	ROC-AUC	BCEWithLogitsLoss
ogbg-molpcba	AP	BCEWithLogitsLoss
ogbg-ppa	Accuracy	CrossEntropyLoss
ogbg-code	F1 score	CrossEntropyLoss

3.2.3 Create a dataset via URL

If your dataset is the same as the 'ppi' dataset, which contains two matrices: 'network' and 'group', you can register your dataset directly use the above code. The default root for downloading dataset is *~/.cache-autogl*, you can also specify the root by passing the string to the *path* in *build_dataset(args, path)* or *build_dataset_from_name(dataset, path)*.

```
# following code-snippet is from autogl/datasets/matlab_matrix.py

@register_dataset("ppi")
class PPIDataset(MatlabMatrix):
    def __init__(self, path):
        dataset, filename = "ppi", "Homo_sapiens"
        url = "http://snap.stanford.edu/node2vec/"
        super(PPIDataset, self).__init__(path, filename, url)
```

You should declare the name of the dataset, the name of the file, and the URL, where our script can download the resource. Then you can use either *build_dataset(args, path)* or *build_dataset_from_name(dataset, path)* in your task to build a dataset with corresponding parameters.

3.2.4 Create a dataset locally

If you want to test your local dataset, we recommend you to refer to the docs on creating PyTorch Geometric dataset.

You can simply inherit from *torch_geometric.data.InMemoryDataset* to create an empty *dataset*, then create some *torch_geometric.data.Data* objects for your data and pass a regular python list holding them, then pass them to *torch_geometric.data.Dataset* or *torch_geometric.data.DataLoader*. Let's see this process in a simplified example:

```
from typing import Iterable
from torch_geometric.data.data import Data
from autogl.datasets import build_dataset_from_name
from torch_geometric.data import InMemoryDataset

class MyDataset(InMemoryDataset):
    def __init__(self, datalist) -> None:
        super().__init__()
        self.data, self.slices = self.collate(datalist)

# Create your own Data objects

# for example, if you have edge_index, features and labels
# you can create a Data as follows
```

(continues on next page)

(continued from previous page)

```
# See pytorch geometric more info of Data
data = Data()
data.edge_index = edge_index
data.x = features
data.y = labels

# create a list of Data object
data_list = [data, Data(...), ..., Data(...)]

# Initialize AutoGL Dataset with your own data
myData = MyDataset(data_list)
```

3.3 AutoGL Feature Engineering

We provide a series of node and subgraph feature engineers for you to compose within a feature engineering pipeline. An automatic feature engineering algorithm is also provided.

3.3.1 Quick Start

```
# 1. Choose a dataset.
from autogl.datasets import build_dataset_from_name
data = build_dataset_from_name('cora')

# 2. Compose a feature engineering pipeline
from autogl.module.feature import BaseFeatureAtom,AutoFeatureEngineer
from autogl.module.feature.generators import GeEigen
from autogl.module.feature.selectors import SeGBDT
from autogl.module.feature.subgraph import SgNetLSD
# you may compose feature engineering atoms through BaseFeatureAtom.compose
fe = BaseFeatureAtom.compose([
GeEigen(size=32) ,
SeGBDT(fixlen=100),
SgNetLSD()
])
# or just through '&' operator
fe = fe & AutoFeatureEngineer(fixlen=200,max_epoch=3)

# 3. Fit and transform the data
fe.fit(data)
data1=fe.transform(data,inplace=False)
```

3.3.2 List of FE atom names

Now three kinds of feature engineering atoms are supported,namely `generators`, `selectors` , `subgraph`.You can import atoms from according module as is mentioned in the `Quick Start` part. Or you may want to just list names of atoms in configurations or as arguments of the autogl solver.

1. `generators`

Atom	Description
graphlet	concatenate local graphlet numbers as features.
eigen	concatenate Eigen features.
pagerank	concatenate Pagerank scores.
PYGLocalDegreeProfile	concatenate Local Degree Profile features.
PYGNormalizeFeatures	Normalize all node features
PYGOneHotDegree	concatenate degree one-hot encoding.
onehot	concatenate node id one-hot encoding.

2. selectors

Atom	Description
SeFilterConstant	delete all constant and one-hot encoding node features.
gbdt	select top-k important node features ranked by Gradient Descent Decision Tree.

3. subgraph

netlsd is a subgraph feature generation method. please refer to the according document.

A set of subgraph feature extractors implemented in NetworkX are wrapped, please refer to NetworkX for details. (NxLargeCliqueSize, NxAverageClusteringApproximate, NxDegreeAssortativityCoefficient, NxDegreePearsonCorrelationCoefficient, NxHasBridge ,``NxGraphCliqueNumber``, NxGraphNumberOfCliques, NxTransitivity, NxAverageClustering, NxIsConnected, NxNumberConnectedComponents, NxIsDistanceRegular, NxLocalEfficiency, NxGlobalEfficiency, NxIsEulerian)

The taxonomy of atom types is based on the way of transforming features. generators concatenate the original features with ones newly generated or just overwrite the original ones. Instead of generating new features , selectors try to select useful features and keep learned selecting methods in the atom itself. The former two types of atoms can be exploited in node or edge level (modification upon each node or edge feature) ,while subgraph focuses on feature engineering in graph level (modification upon each graph feature). For your convenience in further development,you may want to design a new item by inheriting one of them. Of course, you can directly inherit the BaseFeatureAtom as well.

3.3.3 Create Your Own FE

You can create your own feature engineering object by simply inheriting one of feature engineering atom types ,namely generators, selectors , subgraph, and overloading methods _fit and _transform.

```
# for example : create a node one-hot feature.
from autogl.module.feature.generators.base import BaseGenerator
import numpy as np
class GeOnehot(BaseGenerator):
    def __init__(self):
        super(GeOnehot,self).__init__(data_t='np',multigraph=True,subgraph=False)
        # data type in mid is 'numpy',
        # and it can be used for multigraph,
        # but not suitable for subgraph feature extraction.

    def _fit(self):
        pass # nothing to train or memorize

    def _transform(self, data):
        fe=np.eye(data.x.shape[0])
```

(continues on next page)

(continued from previous page)

```
        data.x=np.concatenate([data.x,fe],axis=1)
        return data
```

3.4 AutoGL Model

AutoGL project uses `model` to define the common graph nerual networks and `automodel` to denote the relative class that includes some auto functions. Currently, we support the following models and automodels:

- `GCN` and `AutoGCN` : graph convolutional network from https://arxiv.org/abs/1609.02907
- `GAT` and `AutoGAT` : graph attentional network from https://arxiv.org/abs/1710.10903
- `GraphSAGE` and `AutoGraphSAGE` : from the "Inductive Representation Learning on Large Graphs" https://arxiv.org/abs/1706.02216

And we also support the following models and automodels for graph classification tasks: * `GIN` and `AutoGIN` : graph isomorphism network from https://arxiv.org/abs/1810.00826 * `Topkpool` and `AutoTopkpool` : graph U-Net from https://arxiv.org/abs/1905.05178, https://arxiv.org/abs/1905.02850

3.4.1 Define your own model and automodel

If you want to add your own model and automodel for some task, the only thing you should do is add a new model where the forward function should be fulfilled and a new automodel inherited from the basemodel.

Firstly, you should define your model if it does not belong to the models above.

Secondly, you should define your corresponding automodel.

```
# 1. define your search space to self.space of your automodel instance
[
    {'parameterName': 'num_layers', 'type': 'DISCRETE', 'feasiblePoints': '2,3,4'},
    {"parameterName": 'hidden', "type": "NUMERICAL_LIST", "numericalType": "INTEGER",
↪"length": 3, "minValue": [8, 8, 8], "maxValue": [64, 64, 64], "scalingType": "LOG"},
    {'parameterName': 'dropout', 'type': 'DOUBLE', 'maxValue': 0.9, 'minValue': 0.1,
↪'scalingType': 'LINEAR'},
    {'parameterName': 'act', 'type': 'CATEGORICAL_LIST', "feasiblePoints": ['leaky_
↪relu', 'relu', 'elu', 'tanh']},
]
# 2. define the default point to self.hyperparams of your automodel instance
{
    'num_layers': 2,
    'hidden': [16],
    'dropout': 0.2,
    'act': 'leaky_relu'
}
```

Where `self.space` is a list of dictionary indicating the name, type, feasible point, min/max value and some properties of the parameter. `self.hyperparams` is a dictionary indicating the hyper-parameters used in this model.

Finally, you can use the defined model and automodel for the specific need.

```
# for example
import torch
from .base import BaseModel
class YourGNN(torch.nn.Module):
```

(continues on next page)

(continued from previous page)

```
    def forward(self, data):
        pass  # Your forward function

class YourAutoGNN(BaseModel):
    def __init__(self, num_features=None, num_classes=None, device=None, init=True,␣
↪**args):
        """
        num_features: the number of features
        num_classes: the number of classes
        device: your device to run code
        init: if True, the model will be initialize
        """
        self.space = XXX  # Define your search space
        self.hyperparams = XXX  # Define your hyper-parameters
        self.initialized = False
        if init is True:
            self.initialize()
```

3.5 AutoGL Trainer

AutoGL project use `trainer` to handle the auto-training of tasks. Currently, we support the following tasks:

- `NodeClassificationTrainer` for semi-supervised node classification
- `GraphClassificationTrainer` for supervised graph classification

3.5.1 Initialization

A trainer can either be initialized from its `__init__()`. If you want to build a trainer by `__init__()`, you need to pass the following parameters to it, namely as `model`, `num_features`, and `num_classes` and `auto ensemble`. You can also define some parameters alternatively, including `optimizer`, `lr`, `max_epoch`, `early_stopping_round`, `weight_decay` and etc.

In the `__init__()`, you need to define the space and hyperparameter of your trainer:

```
# 1. define your search space of trainer
self.space = [
    {'parameterName': 'max_epoch', 'type': 'INTEGER', 'maxValue': 300, 'minValue': 10,
↪ 'scalingType': 'LINEAR'},
    {'parameterName': 'early_stopping_round', 'type': 'INTEGER', 'maxValue': 30,
↪'minValue': 10,
         'scalingType': 'LINEAR'},
    {'parameterName': 'lr', 'type': 'DOUBLE', 'maxValue': 1e-3, 'minValue': 1e-4,
↪'scalingType': 'LOG'},
    {'parameterName': 'weight_decay', 'type': 'DOUBLE', 'maxValue': 5e-3, 'minValue':␣
↪5e-4,
         'scalingType': 'LOG'},
]

# 2. define the initial point of hyperparameter search of your trainer
self.hyperparams = {
    'max_epoch': self.max_epoch,
    'early_stopping_round': self.early_stopping_round,
    'lr': self.lr,
```

(continues on next page)

(continued from previous page)

```
    'weight_decay': self.weight_decay
}
```

Where `self.space` is a list of dictionary indicating the name, type, and some properties of the parameter. `self.hyperparams` is a dictionary indicating the hyper-parameters used in this trainer.

3.5.2 Train and Predict

After initializing a trainer, you can train it on the given datasets.

We have given the training and testing functions for the tasks of node classification and graph classification up to now. You can also create your tasks following the similar patterns with ours. For training, you need to define `train_only()` and use it in `train()`. For testing, you need to define `predict_proba()` and use it in `predict()`.

The evaluation funtion is defined in `evaluate()`, you can use your our evaluation metrics and methods.

3.6 Hyper Parameter Optimization

We support black box hyper parameter optimization in variant search space.

3.6.1 Search Space

Three types of search space are supported, use `dict` in python to define your search space. For numerical list search space. You can either assign a fixed length for the list, if so, you need not provide `cutPara` and `cutFunc`. Or you can let HPO cut the list to a certain length which is dependent on other parameters. You should provide those parameters' names in `curPara` and the function to calculate the cut length in "cutFunc".

```
# numerical search space:
{
    "parameterName": "xxx",
    "type": "DOUBLE" / "INTEGER",
    "minValue": xx,
    "maxValue": xx,
    "scalingType": "LINEAR" / "LOG"
}

# numerical list search space:
{
    "parameterName": "xxx",
    "type": "NUMERICAL_LIST",
    "numericalType": "DOUBLE" / "INTEGER",
    "length": 3,
    "cutPara": ("para_a", "para_b"),
    "cutFunc": lambda x: x[0] - 1,
    "minValue": [xx,xx,xx],
    "maxValue": [xx,xx,xx],
    "scalingType": "LINEAR" / "LOG"
}

# categorical search space:
{
```

(continues on next page)

(continued from previous page)

```
    "parameterName": xxx,
    "type": "CATEGORICAL"
    "feasiblePoints": [a,b,c]
}

# fixed parameter as search space:
{
    "parameterName": xxx,
    "type": "FIXED",
    "value": xxx
}
```

How given HPO algorithms support search space is listed as follows:

Algorithm	numerical	numerical list	categorical	fixed
Grid			✓	✓
Random	✓	✓	✓	✓
Anneal	✓	✓	✓	✓
Bayes	✓	✓	✓	✓
TPE	✓	✓	✓	✓
CMAES	✓	✓	✓	✓
MOCMAES	✓	✓	✓	✓
Quasi random	✓	✓	✓	✓
AutoNE	✓	✓	✓	✓

Here, TPE is from [1], CMAES is from [2], MOCMAES is from [3], quasi random is from [4], AutoNE is from [5].

[1] Bergstra, James S., et al. "Algorithms for hyper-parameter optimization." Advances in neural information processing systems. 2011. [2] Arnold, Dirk V., and Nikolaus Hansen. "Active covariance matrix adaptation for the (1+1)-CMA-ES." Proceedings of the 12th annual conference on Genetic and evolutionary computation. 2010. [3] Voß, Thomas, Nikolaus Hansen, and Christian Igel. "Improved step size adaptation for the MO-CMA-ES." Proceedings of the 12th annual conference on Genetic and evolutionary computation. 2010. [4] Bratley, Paul, Bennett L. Fox, and Harald Niederreiter. "Programs to generate Niederreiter's low-discrepancy sequences." ACM Transactions on Mathematical Software (TOMS) 20.4 (1994): 494-495. [5] Tu, Ke, et al. "Autone: Hyperparameter optimization for massive network embedding." Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2019.

3.6.2 Add Your HPOptimizer

If you want to add your own HPOptimizer, the only thing you should do is finishing `optimize` function in you HPOptimizer:

```
# For example, create a random HPO by yourself
import random
from autogl.module.hpo.base import BaseHPOptimizer
class RandomOptimizer(BaseHPOptimizer):
    # Get essential parameters at initialization
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.max_evals = kwargs.get("max_evals", 2)

    # The most important thing you should do is completing optimization function
    def optimize(self, trainer, dataset, time_limit=None, memory_limit=None):
```

(continues on next page)

(continued from previous page)

```
        # 1. Get the search space from trainer.
        space = trainer.hyper_parameter_space + trainer.model.hyper_parameter_space
        # optional: use self._encode_para (in BaseOptimizer) to pretreat the space
        # If you use _encode_para, the NUMERICAL_LIST will be spread to DOUBLE or␣
↪INTEGER, LOG scaling type will be changed to LINEAR, feasible points in CATEGORICAL␣
↪will be changed to discrete numbers.
        # You should also use _decode_para to transform the types of parameters back.
        current_space = self._encode_para(space)

        # 2. Define your function to get the performance.
        def fn(dset, para):
            current_trainer = trainer.duplicate_from_hyper_parameter(para)
            current_trainer.train(dset)
            loss, self.is_higher_better = current_trainer.get_valid_score(dset)
            # For convenience, we change the score which is higher better to negative,
↪ then we should only minimize the score.
            if self.is_higher_better:
                loss = -loss
            return current_trainer, loss

        # 3. Define the how to get HP suggestions, it should return a parameter dict.␣
↪You can use history trials to give new suggestions
        def get_random(history_trials):
            hps = {}
            for para in current_space:
                # Because we use _encode_para function before, we should only deal␣
↪with DOUBLE, INTEGER and DISCRETE
                if para["type"] == "DOUBLE" or para["type"] == "INTEGER":
                    hp = random.random() * (para["maxValue"] - para["minValue"]) +␣
↪para["minValue"]
                    if para["type"] == "INTEGER":
                        hp = round(hp)
                    hps[para["parameterName"]] = hp
                elif para["type"] == "DISCRETE":
                    feasible_points = para["feasiblePoints"].split(",")
                    hps[para["parameterName"]] = random.choice(feasible_points)
            return hps

        # 4. Run your algorithm. For each turn, get a set of parameters according to␣
↪history information and evaluate it.
        best_trainer, best_para, best_perf = None, None, None
        self.trials = []
        for i in range(self.max_evals):
            # in this example, we don't need history trails. Since we pass None to␣
↪history_trails
            new_hp = get_random(None)
            # optional: if you use _encode_para, use _decode_para as well. para_for_
↪trainer undos all transformation in _encode_para, and turns double parameter to␣
↪interger if needed. para_for_hpo only turns double parameter to interger.
            para_for_trainer, para_for_hpo = self._decode_para(new_hp)
            current_trainer, perf = fn(dataset, para_for_trainer)
            self.trials.append((para_for_hpo, perf))
            if not best_perf or perf < best_perf:
                best_perf = perf
                best_trainer = current_trainer
                best_para = para_for_trainer
```

(continues on next page)

(continued from previous page)

```
        # 5. Return the best trainer and parameter.
        return best_trainer, best_para
```

3.7 Ensemble

We currently support voting and stacking methods.

3.7.1 Voting

A voter essentially constructs a weighted sum of the predictions of base learners. Given an evaluation metric, the weights of base learners are specified in some way to maximize the validation score.

We adopt Rich Caruana's method for weight specification. This method first finds a collection of (possibly redundant) base learners with equal weights via a greedy search, then specifies the weights in the voter by the number of occurrence in the collection.

You can customize your own weight specification method by overwriting the `_specify_weights` method.

```
# An example : use equal weights for all base learners.
class EqualWeightVoting(Voting):
    def _specify_weights(self, predictions, label, feval):
        return np.ones(self.n_models)/self.n_models
        # just allocate the same weight for each base learner
```

3.7.2 Stacking

A stacker trains a meta-model with the predictions of base learners as input to find an optimal combination of these base learners.

Currently we support generalized linear model (GLM) and gradient boosting model (GBM) as the meta-model.

3.7.3 Create a New Ensembler

You can create your own ensembler by inheriting the base ensembler, and overloading methods `fit` and `ensemble`.

```
# An example : use the currently available best model.
from autogl.module.ensemble.base import BaseEnsembler
import numpy as np
class BestModel(BaseEnsembler):
    def fit(self, predictions, label, identifiers, feval):
        if not isinstance(feval, list):
            feval = [feval]
        scores = np.array([feval[0].evaluate(pred, label) for pred in predictions]) *␣
↪(1 if feval[0].is_higher_better else -1)
        self.scores = dict(zip(identifiers, scores)) # record validation score of␣
↪base learners
        ensemble_pred = predictions[np.argmax(scores)]
        return [fx.evaluate(ensemble_pred, label) for fx in feval]

    def ensemble(self, predictions, identifiers):
```

(continues on next page)

(continued from previous page)

```
        best_idx = np.argmax([self.scores[model_name] for model_name in identifiers])
↪# choose the currently best model in the identifiers
        return predictions[best_idx]
```

3.8 AutoGL Solver

Our AutoGL project use `solver` to handle the auto-solvation of tasks. Currently, we support the following tasks:

- `AutoNodeClassifier` for semi-supervised node classification
- `AutoGraphClassifier` for supervised graph classification

3.8.1 Initialization

A solver can either be initialized from its `__init__()` or from a config dictionary or file.

Initialize from `__init__()`

If you want to build a solver by `__init__()`, you need to pass the four key modules to it, namely as `auto feature engineer`, `auto model list`, `hyperparameter optimizer` and `auto ensemble`. You can either pass the keywords of corresponding modules or the initialized instances:

```
from autogl.solver import AutoNodeClassifier

# 1. initialize from keywords
solver = AutoNodeClassifier(
    feature_module='deepgl',
    graph_models=['gat','gcn'],
    hpo_module='anneal',
    ensemble_module='voting',
    device='auto'
)

# 2. initialize using instances
from autogl.module import AutoFeatureEngineer, AutoGCN, AutoGAT, AnnealAdvisorHPO,␣
↪Voting
solver = AutoNodeClassifier(
    feature_module=AutoFeatureEngineer(),
    graph_models=[AutoGCN(device='cuda'), AutoGAT(device='cuda')],
    hpo_module=AnnealAdvisorHPO(max_evals=10),
    ensemble_module=Voting(size=2),
    device='cuda'
)
```

Where, the argument `device` means where to perform the training and searching, by setting to `auto`, the `cuda` is used when it is available.

If you want to disable one module (except graphModuleList), you can set it to `None`:

```
solver = AutoNodeClassifier(feature_module=None, hpo_module=None, ensemble_
↪module=None)
```

You can also pass some important arguments of modules directly to solver, which will automatically set them for you:

```
solver = AutoNodeClassifier(hpo_module='anneal', max_evals=10)
```

Refer to *solver* for more details of argument default value or important argument lists.

Initialize from config dictionary or file

You can also initialize a solver directly from a config dictionary or file. Currently, the AutoGL solver supports config file type of `yaml` or `json`. You need to use `from_config()` when you want to initialize in this way:

```
# initialize from config file
path_to_config = 'your/path/to/config'
solver = AutoNodeClassifier.from_config(path_to_config)

# initialize from a dictionary
config = {
    'models':{'gcn': None, 'gat': None},
    'hpo': {'name': 'tpe', 'max_evals': 10},
    'ensemble': {'name': 'voting', 'size': 2}
}
solver = AutoNodeClassifier.from_config(config)
```

Refer to the config dictionary description *Config structure* for more details.

3.8.2 Optimization

After initializing a solver, you can optimize it on the given datasets (please refer to *AutoGL Dataset* and *dataset* for creating datasets).

You can use `fit()` or `fit_predict()` to perform optimization, which shares similar argument lists:

```
# load your dataset here
dataset = some_dataset()
solver.fit(dataset, inplace=True, time_limit=3600)
```

The inplace argument is used for saving memory if set to `True`. It will modify your dataset in an inplace manner during feature engineering. You can also set `time_limit` to limit the time cost of the whole auto process.

You can also specify the `train_split` and `val_split` arguments to let solver auto-split the given dataset. If these arguments are given, the split dataset will be used instead of the default split specified by the dataset provided. All the models will be trained on `train dataset`. Their hyperparameters will be optimized based on the performance of `valid dataset`, as well as the final ensemble method. For example:

```
# split 0.2 of total nodes/graphs for train and 0.4 of nodes/graphs for validation,
# the rest 0.4 is left for test.
solver.fit(dataset, train_split=0.2, val_split=0.4)

# split 600 nodes/graphs for train and 400 nodes/graphs for validation,
# the rest nodes are left for test.
solver.fit(dataset, train_split=600, val_split=400)
```

For the node classification problem, we also support balanced sampling of train and valid: force the number of sampled nodes in different classes to be the same. The balanced mode can be turned on by setting `balanced=True` in `fit()`, which is by default set to `True`.

For the graph classification problem, we also provide a way to conduct cross-validation. You can enable cross-validation by specifying `cross_validation=True`. `cv_fold` is also provided to determine the number of folds.

Then, the `train dataset` will be further split into `cv_fold` folds for each model to be trained and optimized hyperparameters on. The auto ensemble will base on the model performance of `valid dataset`.

Note: If you want to use cross validation, please make sure the `dataset` receives train/val/test split before cross validated. By default, the graph dataset derived directly from `build_dataset_from_name` is not splitted yet. To split the dataset, you can:

- use `autogl.datasets.utils.graph_random_split` to pre-split `dataset` outside of `solver`.
- pass `train_split` and `val_split` directly to `solver`, which will pre-split the `dataset` for you.

Note: Solver will maintain the models with the best hyper-parameter of each model architecture you pass to solver (the `graphModelList` argument when initialized). The maintained models will then be ensembled by ensemble module. When cross-validation is used, solver will maintain `cv_fold` models of each model architecture for each fold.

After `fit()`, solver maintains the performances of every single model and the ensemble model in one leaderboard instance. You can output the performances on valid dataset by:

```
# get current leaderboard of the solver
lb = solver.get_leaderboard()
# show the leaderboard info
lb.show()
```

You can refer to the leaderboard documentation in *solver* for more usage.

3.8.3 Prediction

After optimized on the given dataset, you can make predictions using the fitted `solver`.

Prediction using ensemble

You can use the ensemble model constructed by solver to make the prediction, which is recommended and is the default choice:

```
solver.predict()
```

If you do not pass any dataset, the dataset during fitting will be used to give the prediction.

You can also pass the dataset when predicting, please make sure the `inplaced` argument is properly set.

```
solver.predict(dataset, inplaced=True, inplace=True)
```

The `predict()` function also has `inplace` argument, which is the same as the one in `fit()`. As for the `inplaced`, it means whether the passed dataset is already modified inplace or not (probably by `fit()` function). If you use `fit()` before, please make the `inplaced` of `predict()` stay the same with `inplace` in `fit()`.

Prediction using one single model

You can also make the prediction using the best single model the solver maintains by:

```
solver.predict(use_ensemble=False, use_best=True)
```

Also, you can name the single model maintained by solver to make predictions.

```
solver.predict(use_ensemble=False, use_best=False, name=the_name_of_model)
```

The names of models can be derived by calling `solver.trained_models.keys()`, which is the same as the names maintained by the leaderboard of solver.

Note: By default, solver will only make predictions on the `test` split of given datasets. Please make sure the passed dataset has the `test` split when making predictions. You can also change the default prediction split by setting argument `mask` to `train` or `valid`.

3.8.4 Appendix

Config structure

The structure of the config file or config is introduced here. The config should be a dict, with five optional keys, namely `feature`, `models`, `trainer`, `hpo` and `ensemble`. You can simply do not add one field if you want to use the default option. The default value of each module is the same as the one in `__init__()`.

For key `feature`, `hpo` and `ensemble`, their corresponding values are all dictionaries, which contains one must key `name` and other arguments when initializing the corresponding modules. The value of key `name` specifies which algorithm should be used, where `None` can be passed if you do not want to enable the module. Other arguments are used to initialize the specified algorithm.

For key `trainer`, you should specify the hyperparameter space of trainer. See *AutoGL Trainer* or *train* for the detailed hyperparameter space of different trainers.

For key `models`, the value is another dictionary with its keys being models that need optimized and the corresponding values being the hyperparameter space of that model. See *AutoGL Model* or *model* for the detailed hyperparameter space of different models.

Below shows some examples of the config dictionary.

```
config_for_node_classification = {
    'feature': {
        'name': 'deepgl',       # name of auto feature module
        # following are the deepgl specified auto feature engineer arguments
        'fixlen': 100,
        'max_epoch': 5
    },
    'models': {
        'gcn':
        # specify the hp space of gcn
        [
            {'parameterName': 'num_layers', 'type': 'DISCRETE', 'feasiblePoints': '2,
↪3,4'},
            {'parameterName': 'hidden', 'type': 'NUMERICAL_LIST', 'numericalType':
↪'INTEGER', 'length': 3,
                'minValue': [8, 8, 8], 'maxValue': [64, 64, 64], 'scalingType': 'LOG'}
↪,
```

(continues on next page)

(continued from previous page)

```
            {'parameterName': 'dropout', 'type': 'DOUBLE', 'maxValue': 0.9, 'minValue
↪': 0.1, 'scalingType': 'LINEAR'},
            {'parameterName': 'act', 'type': 'CATEGORICAL', 'feasiblePoints': ['leaky_
↪relu', 'relu', 'elu', 'tanh']}
        ],
        'gat': None,              # set to None to use default hp space
        'gin': None
    }
    'trainer': [
        # trainer hp space
        {'parameterName': 'max_epoch', 'type': 'INTEGER', 'maxValue': 300, 'minValue
↪': 10, 'scalingType': 'LINEAR'},
        {'parameterName': 'early_stopping_round', 'type': 'INTEGER', 'maxValue': 30,
↪'minValue': 10, 'scalingType': 'LINEAR'},
        {'parameterName': 'lr', 'type': 'DOUBLE', 'maxValue': 0.001, 'minValue': 0.
↪0001, 'scalingType': 'LOG'},
        {'parameterName': 'weight_decay', 'type': 'DOUBLE', 'maxValue': 0.005,
↪'minValue': 0.0005, 'scalingType': 'LOG'}
    ],
    'hpo': {
        'name': 'autone',       # name of hpo module
        # following are the autone specified auto hpo arguments
        'max_evals': 10,
        'subgraphs': 10,
        'sub_evals': 5
    },
    'ensemble': {
        'name': 'voting',       # name of ensemble module
        # following are the voting specified auto ensemble arguments
        'size': 2
    }
}

config_for_graph_classification = {
    'feature': None,            # set to None to disable this module
    # do not add field `model` to use default settings of solver
    'trainer': [
        # trainer hp space
        {'parameterName': 'max_epoch', 'type': 'INTEGER', 'maxValue': 300, 'minValue
↪': 10, 'scalingType': 'LINEAR'},
        {'parameterName': 'batch_size', 'type': 'INTEGER', 'maxValue': 128, 'minValue
↪': 32, 'scalingType': 'LOG'},
        {'parameterName': 'early_stopping_round', 'type': 'INTEGER', 'maxValue': 30,
↪'minValue': 10, 'scalingType': 'LINEAR'},
        {'parameterName': 'lr', 'type': 'DOUBLE', 'maxValue': 1e-3, 'minValue': 1e-4,
↪'scalingType': 'LOG'},
        {'parameterName': 'weight_decay', 'type': 'DOUBLE', 'maxValue': 5e-3,
↪'minValue': 5e-4, 'scalingType': 'LOG'},
    ],
    'hpo': {
        'name': 'random',       # name of hpo module
        # following are the random specified auto hpo arguments
        'max_evals': 10
    },
    'ensemble': None            # set to None to disable this module
}
```

3.9 data

class autogl.data.**Batch**(*batch=None*, ***kwargs*)

A plain old python object modeling a batch of graphs as one big (dicconnected) graph. With cogdl.data.Data being the base class, all its methods can also be used here. In addition, single graphs can be reconstructed via the assignment vector batch, which maps each node to its respective graph identifier.

cumsum(*key*, *item*)

If True, the attribute key with content item should be added up cumulatively before concatenated together.

Note: This method is for internal use only, and should only be overridden if the batch concatenation process is corrupted for a specific data attribute.

---

static from_data_list(*data_list*, *follow_batch=[]*)

Constructs a batch object from a python list holding torch_geometric.data.Data objects. The assignment vector batch is created on the fly. Additionally, creates assignment batch vectors for each key in follow_batch.

property num_graphs

Returns the number of graphs in the batch.

to_data_list()

Reconstructs the list of torch_geometric.data.Data objects from the batch object. The batch object must have been created via *from_data_list()* in order to be able reconstruct the initial objects.

class autogl.data.**Data**(*x=None*, *edge_index=None*, *edge_attr=None*, *y=None*, *pos=None*)

A plain old python object modeling a single graph with various (optional) attributes:

Parameters

- **x** (*Tensor, optional*) – Node feature matrix with shape [num_nodes, num_node_features]. (default: None)
- **edge_index** (*LongTensor, optional*) – Graph connectivity in COO format with shape [2, num_edges]. (default: None)
- **edge_attr** (*Tensor, optional*) – Edge feature matrix with shape [num_edges, num_edge_features]. (default: None)
- **y** (*Tensor, optional*) – Graph or node targets with arbitrary shape. (default: None)
- **pos** (*Tensor, optional*) – Node position matrix with shape [num_nodes, num_dimensions]. (default: None)

The data object is not restricted to these attributes and can be extented by any other additional data.

__call__(**keys*)

Iterates over all attributes *keys in the data, yielding their attribute names and content. If *keys is not given this method will iterative over all present attributes.

__contains__(*key*)

Returns True, if the attribute key is present in the data.

__getitem__(*key*)

Gets the data of the attribute key.

__inc__(*key*, *value*)

"Returns the incremental count to cumulatively increase the value of the next attribute of key when creating batches.

Note: This method is for internal use only, and should only be overridden if the batch concatenation process is corrupted for a specific data attribute.

__iter__()
Iterates over all present attributes in the data, yielding their attribute names and content.

__len__()
Returns the number of all present attributes.

__setitem__(*key*, *value*)
Sets the attribute `key` to `value`.

apply(*func*, **keys*)
Applies the function `func` to all attributes `*keys`. If `*keys` is not given, `func` is applied to all present attributes.

cat_dim(*key*, *value*)
Returns the dimension in which the attribute `key` with content `value` gets concatenated when creating batches.

Note: This method is for internal use only, and should only be overridden if the batch concatenation process is corrupted for a specific data attribute.

contiguous(**keys*)
Ensures a contiguous memory layout for all attributes `*keys`. If `*keys` is not given, all present attributes are ensured to have a contiguous memory layout.

static from_dict(*dictionary*)
Creates a data object from a python dictionary.

get_label_number()
Get the number of labels in this dataset as dict.

is_coalesced()
Returns `True`, if edge indices are ordered and do not contain duplicate entries.

property keys
Returns all names of graph attributes.

property num_edges
Returns the number of edges in the graph.

property num_features
Returns the number of features per node in the graph.

random_splits_mask(*train_ratio*, *val_ratio*, *seed=None*)
If the data has masks for train/val/test, return the splits with specific ratio.

Parameters

- **train_ratio** (*float*) – the portion of data that used for training.
- **val_ratio** (*float*) – the portion of data that used for validation.
- **seed** (*int*) – random seed for splitting dataset.

random_splits_mask_class(*num_train_per_class*, *num_val*, *num_test*, *seed=None*)
If the data has masks for train/val/test, return the splits with specific number of samples from every class for training.

Parameters

- **num_train_per_class** (*int*) – the number of samples from every class used for training.
- **num_val** (*int*) – the total number of nodes that used for validation.
- **num_test** (*int*) – the total number of nodes that used for testing.
- **seed** (*int*) – random seed for splitting dataset.

random_splits_nodes(*train_ratio*, *val_ratio*, *seed=None*)
If the data uses id of nodes for train/val/test, return the splits with specific ratio.

Parameters

- **train_ratio** (*float*) – the portion of data that used for training.
- **val_ratio** (*float*) – the portion of data that used for validation.
- **seed** (*int*) – random seed for splitting dataset.

random_splits_nodes_class(*num_train_per_class*, *num_val*, *num_test*, *seed=None*)
If the data uses id of nodes for train/val/test, return the splits with specific number of samples from every class for training.

Parameters

- **num_train_per_class** (*int*) – the number of samples from every class used for training.
- **num_val** (*int*) – the total number of nodes that used for validation.
- **num_test** (*int*) – the total number of nodes that used for testing.
- **seed** (*int*) – random seed for splitting dataset.

to(*device*, **keys*)
Performs tensor dtype and/or device conversion to all attributes `*keys`. If `*keys` is not given, the conversion is applied to all present attributes.

class autogl.data.**DataListLoader**(*dataset*, *batch_size=1*, *shuffle=True*, ***kwargs*)
Data loader which merges data objects from a `cogdl.data.dataset` to a python list.

Note: This data loader should be used for multi-gpu support via `cogdl.nn.DataParallel`.

Parameters

- **dataset** (Dataset) – The dataset from which to load the data.
- **batch_size** (*int, optional*) – How may samples per batch to load. (default: `1`)
- **shuffle** (*bool, optional*) – If set to `True`, the data will be reshuffled at every epoch (default: `True`)

class autogl.data.**DataLoader**(*dataset*, *batch_size=1*, *shuffle=True*, ***kwargs*)
Data loader which merges data objects from a `cogdl.data.dataset` to a mini-batch.

Parameters

- **dataset** (Dataset) – The dataset from which to load the data.
- **batch_size** (*int, optional*) – How may samples per batch to load. (default: `1`)

- **shuffle** (*bool, optional*) – If set to `True`, the data will be reshuffled at every epoch (default: `True`)

class autogl.data.**Dataset**(*root*, *transform=None*, *pre_transform=None*, *pre_filter=None*)
Dataset base class for creating graph datasets. See here for the accompanying tutorial.

Parameters

- **root** (*string*) – Root directory where the dataset should be saved.
- **transform** (*callable, optional*) – A function/transform that takes in an `cogdl.data.Data` object and returns a transformed version. The data object will be transformed before every access. (default: `None`)
- **pre_transform** (*callable, optional*) – A function/transform that takes in an `cogdl.data.Data` object and returns a transformed version. The data object will be transformed before being saved to disk. (default: `None`)
- **pre_filter** (*callable, optional*) – A function that takes in an `cogdl.data.Data` object and returns a boolean value, indicating whether the data object should be included in the final dataset. (default: `None`)

__getitem__(*idx*)
Gets the data object at index `idx` and transforms it (in case a `self.transform` is given).

__len__()
The number of examples in the dataset.

download()
Downloads the dataset to the `self.raw_dir` folder.

get(*idx*)
Gets the data object at index `idx`.

property get_label_number
Get the number of labels in this dataset as dict.

property num_features
Returns the number of features per node in the graph.

process()
Processes the dataset to the `self.processed_dir` folder.

property processed_file_names
The name of the files to find in the `self.processed_dir` folder in order to skip the processing.

property processed_paths
The filepaths to find in the `self.processed_dir` folder in order to skip the processing.

property raw_file_names
The name of the files to find in the `self.raw_dir` folder in order to skip the download.

property raw_paths
The filepaths to find in order to skip the download.

class autogl.data.**DenseDataLoader**(*dataset*, *batch_size=1*, *shuffle=True*, ***kwargs*)
Data loader which merges data objects from a `cogdl.data.dataset` to a mini-batch.

Note: To make use of this data loader, all graphs in the dataset needs to have the same shape for each its attributes. Therefore, this data loader should only be used when working with *dense* adjacency matrices.

Parameters

- **dataset** (Dataset) – The dataset from which to load the data.
- **batch_size** (*int, optional*) – How may samples per batch to load. (default: 1)
- **shuffle** (*bool, optional*) – If set to True, the data will be reshuffled at every epoch (default: True)

autogl.data.**download_url**(*url*, *folder*, *name=None*, *log=True*)
Downloads the content of an URL to a specific folder.

Parameters

- **url** (*string*) – The url.
- **folder** (*string*) – The folder.
- **log** (*bool, optional*) – If False, will not print anything to the console. (default: True)

autogl.data.**extract_tar**(*path*, *folder*, *mode='r:gz'*, *log=True*)
Extracts a tar archive to a specific folder.

Parameters

- **path** (*string*) – The path to the tar archive.
- **folder** (*string*) – The folder.
- **mode** (*string, optional*) – The compression mode. (default: "r:gz")
- **log** (*bool, optional*) – If False, will not print anything to the console. (default: True)

autogl.data.**extract_zip**(*path*, *folder*, *log=True*)
Extracts a zip archive to a specific folder.

Parameters

- **path** (*string*) – The path to the tar archive.
- **folder** (*string*) – The folder.
- **log** (*bool, optional*) – If False, will not print anything to the console. (default: True)

3.10 dataset

We integrate the datasets from PyTorch Geometric, CogDL and OGB. We also list some datasets from *CogDL* for simplicity.

3.11 module

The four main modules for auto graph learning are listed here.

3.11.1 feature

class autogl.module.feature.selectors.**BaseSelector**(*data_t='np'*, *multigraph=False*, ***kwargs*)

class autogl.module.feature.selectors.**SeFilterConstant**(*data_t='np'*, *multigraph=False*, ***kwargs*)

drop constant features

class autogl.module.feature.selectors.**SeGBDT**(*fixlen=10*, **args*, ***kwargs*)

simple wrapper of lightgbm , using importance ranking to select top-k features.

Parameters **fixlen** (*int*) – K for top-K important features.

3.11.2 model

3.11.3 train

3.11.4 hyper parameter optimization

3.11.5 ensemble

class autogl.module.ensemble.**Stacking**(*meta_model='gbm'*, *meta_params={}*, **args*, ***kwargs*)

A stacking ensembler. Currently we support gradient boosting as the meta-algorithm.

Parameters

- **meta_model** (*'gbm' or 'glm' (Optional)*) –

  Type of the stacker: 'gbm' : Gradient boosting model. This is the default. 'glm' : Generalized linear model.

- **meta_params** (a dict (Optional)) – When meta_model is specified, you can customize the parameters of the stacker. If this argument is not provided, the stacker will be configurated with default parameters. Default {}.

ensemble(*predictions*, *identifiers*, **args*, ***kwargs*)

Ensemble the predictions of base models.

Parameters

- **predictions** (a list of np.ndarray) – Predictions of base learners (corresponding to the elements in identifiers).
- **identifiers** (a list of str) – The names of base models.

Returns The ensembled predictions.

Return type np.ndarray

fit(*predictions*, *label*, *identifiers*, *feval*, *n_classes='auto'*, **args*, ***kwargs*)

Fit the ensembler to the given data using Stacking method.

Parameters

- **predictions** (*a list of np.ndarray*) – Predictions of base learners (corresponding to the elements in identifiers).
- **label** (*a list of int*) – Class labels of instances.
- **identifiers** (*a list of str*) – The names of base models.
- **feval** (*(a list of) autogl.module.train.evaluate*) – Performance evaluation metrices.
- **n_classes** (*int or str (Optional)*) – The number of classes. Default as `'auto'`, which will use maximum label.

Returns The validation performance of the final stacker.

Return type (a list of) float

class autogl.module.ensemble.**Voting**(*ensemble_size=10*, **args*, ***kwargs*)

An ensembler using the voting method.

Parameters **ensemble_size** (*int*) – The number of base models selected by the voter. These selected models can be redundant. Default as 10.

ensemble(*predictions*, *identifiers*, **args*, ***kwargs*)

Ensemble the predictions of base models.

Parameters

- **predictions** (*a list of np.ndarray*) – Predictions of base learners (corresponding to the elements in identifiers).
- **identifiers** (*a list of str*) – The names of base models.

Returns The ensembled predictions.

Return type np.ndarray

fit(*predictions*, *label*, *identifiers*, *feval*, **args*, ***kwargs*)

Fit the ensembler to the given data using Rich Caruana's ensemble selection method.

Parameters

- **predictions** (*a list of np.ndarray*) – Predictions of base learners (corresponding to the elements in identifiers).
- **labels** (*a list of int*) – Class labels of instances.
- **identifiers** (*a list of str*) – The names of base models.
- **feval** (*(a list of) instances in autogl.module.train.evaluate*) – Performance evaluation metrices.

Returns The validation performance of the final voter.

Return type (a list of) `float`

autogl.module.ensemble.**build_ensembler_from_name**(*name: str*) → autogl.module.ensemble.base.BaseEnsembler

Parameters **name** (`str`) – the name of ensemble module.

Returns the ensembler built using default parameters

Return type BaseEnsembler

Raises **AssertionError** – If an invalid name is passed in

3.12 solver

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

autogl.data, 22
autogl.module.ensemble, 27
autogl.module.feature.selectors, 27

INDEX

Symbols

__call__() (*autogl.data.Data method*), 22
__contains__() (*autogl.data.Data method*), 22
__getitem__() (*autogl.data.Data method*), 22
__getitem__() (*autogl.data.Dataset method*), 25
__inc__() (*autogl.data.Data method*), 22
__iter__() (*autogl.data.Data method*), 23
__len__() (*autogl.data.Data method*), 23
__len__() (*autogl.data.Dataset method*), 25
__setitem__() (*autogl.data.Data method*), 23

A

apply() (*autogl.data.Data method*), 23
autogl.data
    module, 22
autogl.module.ensemble
    module, 27
autogl.module.feature.selectors
    module, 27

B

BaseSelector (*class in autogl.module.feature.selectors*), 27
Batch (*class in autogl.data*), 22
build_ensembler_from_name() (*in module autogl.module.ensemble*), 28

C

cat_dim() (*autogl.data.Data method*), 23
contiguous() (*autogl.data.Data method*), 23
cumsum() (*autogl.data.Batch method*), 22

D

Data (*class in autogl.data*), 22
DataListLoader (*class in autogl.data*), 24
DataLoader (*class in autogl.data*), 24
Dataset (*class in autogl.data*), 25
DenseDataLoader (*class in autogl.data*), 25
download() (*autogl.data.Dataset method*), 25
download_url() (*in module autogl.data*), 26

E

ensemble() (*autogl.module.ensemble.Stacking method*), 27
ensemble() (*autogl.module.ensemble.Voting method*), 28
extract_tar() (*in module autogl.data*), 26
extract_zip() (*in module autogl.data*), 26

F

fit() (*autogl.module.ensemble.Stacking method*), 27
fit() (*autogl.module.ensemble.Voting method*), 28
from_data_list() (*autogl.data.Batch static method*), 22
from_dict() (*autogl.data.Data static method*), 23

G

get() (*autogl.data.Dataset method*), 25
get_label_number() (*autogl.data.Data method*), 23
get_label_number() (*autogl.data.Dataset property*), 25

I

is_coalesced() (*autogl.data.Data method*), 23

K

keys() (*autogl.data.Data property*), 23

M

module
    autogl.data, 22
    autogl.module.ensemble, 27
    autogl.module.feature.selectors, 27

N

num_edges() (*autogl.data.Data property*), 23
num_features() (*autogl.data.Data property*), 23
num_features() (*autogl.data.Dataset property*), 25
num_graphs() (*autogl.data.Batch property*), 22

P

process() (*autogl.data.Dataset method*), 25

processed_file_names() (*autogl.data.Dataset property*), 25
processed_paths() (*autogl.data.Dataset property*), 25

R

random_splits_mask() (*autogl.data.Data method*), 23
random_splits_mask_class() (*autogl.data.Data method*), 23
random_splits_nodes() (*autogl.data.Data method*), 24
random_splits_nodes_class() (*autogl.data.Data method*), 24
raw_file_names() (*autogl.data.Dataset property*), 25
raw_paths() (*autogl.data.Dataset property*), 25

S

SeFilterConstant (*class in autogl.module.feature.selectors*), 27
SeGBDT (*class in autogl.module.feature.selectors*), 27
Stacking (*class in autogl.module.ensemble*), 27

T

to() (*autogl.data.Data method*), 24
to_data_list() (*autogl.data.Batch method*), 22

V

Voting (*class in autogl.module.ensemble*), 28