453 lines - 1 Removal

```
import os
import glob
import json
import torch
import pickle
import shutil
import numpy as np
import os.path as osp

from torch_geometric.datasets import MoleculeNet
from torch_geometric.utils import dense_to_sparse
from torch.utils.data import random_split, Subset
from torch_geometric.data import Data, InMemoryDataset, download_url, extract_zip
from torch_geometric.loader import DataLoader


def undirected_graph(data):
    data.edge_index = torch.cat([torch.stack([data.edge_index[1], data.edge_index[0]], dim=0),
                                 data.edge_index], dim=1)
    return data


def split(data, batch):
    # i-th contains elements from slice[i] to slice[i+1]
    node_slice = torch.cumsum(torch.from_numpy(np.bincount(batch)), 0)
    node_slice = torch.cat([torch.tensor([0]), node_slice])
    row, _ = data.edge_index
    edge_slice = torch.cumsum(torch.from_numpy(np.bincount(batch[row])), 0)
    edge_slice = torch.cat([torch.tensor([0]), edge_slice])

    # Edge indices should start at zero for every graph.
    data.edge_index -= node_slice[batch[row]].unsqueeze(0)
    data.__num_nodes__ = np.bincount(batch).tolist()

    slices = dict()
    slices['x'] = node_slice
    slices['edge_index'] = edge_slice
    slices['y'] = torch.arange(0, batch[-1] + 2, dtype=torch.long)
    return data, slices


def read_file(folder, prefix, name):
    file_path = osp.join(folder, prefix + f'_{name}.txt')
    return np.genfromtxt(file_path, dtype=np.int64)


def read_sentigraph_data(folder: str, prefix: str):
    txt_files = glob.glob(os.path.join(folder, "{}_*.txt".format(prefix)))
    json_files = glob.glob(os.path.join(folder, "{}_*.json".format(prefix)))
    txt_names = [f.split(os.sep)[-1][len(prefix) + 1:-4] for f in txt_files]
```

583 lines + 131 Additions

```
import os
import glob
import json
import torch
import pickle
import shutil
import numpy as np
import os.path as osp
from typing import Optional, Callable
from torch_geometric.datasets import MoleculeNet
from torch_geometric.utils import dense_to_sparse
from torch.utils.data import random_split, Subset
from torch_geometric.data import Data, InMemoryDataset, download_url, extract_zip, extract_tar
from torch_geometric.loader import DataLoader


def undirected_graph(data):
    data.edge_index = torch.cat([torch.stack([data.edge_index[1], data.edge_index[0]], dim=0),
                                 data.edge_index], dim=1)
    return data


def split(data, batch):
    # i-th contains elements from slice[i] to slice[i+1]
    node_slice = torch.cumsum(torch.from_numpy(np.bincount(batch)), 0)
    node_slice = torch.cat([torch.tensor([0]), node_slice])
    row, _ = data.edge_index
    edge_slice = torch.cumsum(torch.from_numpy(np.bincount(batch[row])), 0)
    edge_slice = torch.cat([torch.tensor([0]), edge_slice])

    # Edge indices should start at zero for every graph.
    data.edge_index -= node_slice[batch[row]].unsqueeze(0)
    data.__num_nodes__ = np.bincount(batch).tolist()

    slices = dict()
    slices['x'] = node_slice
    slices['edge_index'] = edge_slice
    slices['y'] = torch.arange(0, batch[-1] + 2, dtype=torch.long)
    return data, slices


def read_file(folder, prefix, name):
    file_path = osp.join(folder, prefix + f'_{name}.txt')
    return np.genfromtxt(file_path, dtype=np.int64)


def read_sentigraph_data(folder: str, prefix: str):
    txt_files = glob.glob(os.path.join(folder, "{}_*.txt".format(prefix)))
    json_files = glob.glob(os.path.join(folder, "{}_*.json".format(prefix)))
    txt_names = [f.split(os.sep)[-1][len(prefix) + 1:-4] for f in txt_files]
```

```
    json_names = [f.split(os.sep)[-1][len(prefix) + 1:-5] for f in json_files]
    names = txt_names + json_names

    with open(os.path.join(folder, prefix+"_node_features.pkl"), 'rb') as f:
        x: np.array = pickle.load(f)
    x: torch.FloatTensor = torch.from_numpy(x)
    edge_index: np.array = read_file(folder, prefix, 'edge_index')
    edge_index: torch.tensor = torch.tensor(edge_index, dtype=torch.long).T
    batch: np.array = read_file(folder, prefix, 'node_indicator') - 1     # from zero
    y: np.array = read_file(folder, prefix, 'graph_labels')
    y: torch.tensor = torch.tensor(y, dtype=torch.long)

    supplement = dict()
    if 'split_indices' in names:
        split_indices: np.array = read_file(folder, prefix, 'split_indices')
        split_indices = torch.tensor(split_indices, dtype=torch.long)
        supplement['split_indices'] = split_indices
    if 'sentence_tokens' in names:
        with open(os.path.join(folder, prefix + '_sentence_tokens.json')) as f:
            sentence_tokens: dict = json.load(f)
        supplement['sentence_tokens'] = sentence_tokens

    data = Data(x=x, edge_index=edge_index, y=y)
    data, slices = split(data, batch)

    return data, slices, supplement


def read_syn_data(folder: str, prefix):
    with open(os.path.join(folder, f"{prefix}.pkl"), 'rb') as f:
        adj, features, y_train, y_val, y_test, train_mask, val_mask, test_mask, edge_label_matrix = pickle.load(f)

    x = torch.from_numpy(features).float()
    y = train_mask.reshape(-1, 1) * y_train + val_mask.reshape(-1, 1) * y_val + test_mask.reshape(-1, 1) * y_test
    y = torch.from_numpy(np.where(y)[1])
    edge_index = dense_to_sparse(torch.from_numpy(adj))[0]
    data = Data(x=x, y=y, edge_index=edge_index)
    data.train_mask = torch.from_numpy(train_mask)
    data.val_mask = torch.from_numpy(val_mask)
    data.test_mask = torch.from_numpy(test_mask)
    return data


def read_ba2motif_data(folder: str, prefix):
    with open(os.path.join(folder, f"{prefix}.pkl"), 'rb') as f:
        dense_edges, node_features, graph_labels = pickle.load(f)

    data_list = []
    for graph_idx in range(dense_edges.shape[0]):
```


```
    json_names = [f.split(os.sep)[-1][len(prefix) + 1:-5] for f in json_files]
    names = txt_names + json_names

    with open(os.path.join(folder, prefix+"_node_features.pkl"), 'rb') as f:
        x: np.array = pickle.load(f)
    x: torch.FloatTensor = torch.from_numpy(x)
    edge_index: np.array = read_file(folder, prefix, 'edge_index')
    edge_index: torch.tensor = torch.tensor(edge_index, dtype=torch.long).T
    batch: np.array = read_file(folder, prefix, 'node_indicator') - 1     # from zero
    y: np.array = read_file(folder, prefix, 'graph_labels')
    y: torch.tensor = torch.tensor(y, dtype=torch.long)

    supplement = dict()
    if 'split_indices' in names:
        split_indices: np.array = read_file(folder, prefix, 'split_indices')
        split_indices = torch.tensor(split_indices, dtype=torch.long)
        supplement['split_indices'] = split_indices
    if 'sentence_tokens' in names:
        with open(os.path.join(folder, prefix + '_sentence_tokens.json')) as f:
            sentence_tokens: dict = json.load(f)
        supplement['sentence_tokens'] = sentence_tokens

    data = Data(x=x, edge_index=edge_index, y=y)
    data, slices = split(data, batch)

    return data, slices, supplement


def read_syn_data(folder: str, prefix):
    with open(os.path.join(folder, f"{prefix}.pkl"), 'rb') as f:
        adj, features, y_train, y_val, y_test, train_mask, val_mask, test_mask, edge_label_matrix = pickle.load(f)

    x = torch.from_numpy(features).float()
    y = train_mask.reshape(-1, 1) * y_train + val_mask.reshape(-1, 1) * y_val + test_mask.reshape(-1, 1) * y_test
    y = torch.from_numpy(np.where(y)[1])
    edge_index = dense_to_sparse(torch.from_numpy(adj))[0]
    data = Data(x=x, y=y, edge_index=edge_index)
    data.train_mask = torch.from_numpy(train_mask)
    data.val_mask = torch.from_numpy(val_mask)
    data.test_mask = torch.from_numpy(test_mask)
    return data


def read_ba2motif_data(folder: str, prefix):
    with open(os.path.join(folder, f"{prefix}.pkl"), 'rb') as f:
        dense_edges, node_features, graph_labels = pickle.load(f)

    data_list = []
    for graph_idx in range(dense_edges.shape[0]):
```

```
        data_list.append(Data(x=torch.from_numpy(node_features[graph_idx]).float(),
                              edge_index=dense_to_sparse(torch.from_numpy(dense_edges[graph_idx]))[0],
                              y=torch.from_numpy(np.where(graph_labels[graph_idx])[0])))
    return data_list


def get_dataset(dataset_dir, dataset_name, task=None):
    sync_dataset_dict = {
        'BA_2Motifs'.lower(): 'BA_2Motifs',
        'BA_Shapes'.lower(): 'BA_shapes',
        'BA_Community'.lower(): 'BA_Community',
        'Tree_Cycle'.lower(): 'Tree_Cycle',
        'Tree_Grids'.lower(): 'Tree_Grids',
        'BA_LRP'.lower(): 'ba_lrp'
    }

    sentigraph_names = ['Graph_SST2', 'Graph_Twitter', 'Graph_SST5']
    sentigraph_names = [name.lower() for name in sentigraph_names]
    molecule_net_dataset_names = [name.lower() for name in MoleculeNet.names.keys()]

    if dataset_name.lower() == 'Mutagenicity'.lower():
        return load_MUTAG(dataset_dir, 'mutagenicity')
    elif dataset_name.lower() in sync_dataset_dict.keys():
        sync_dataset_filename = sync_dataset_dict[dataset_name.lower()]
        return load_syn_data(dataset_dir, sync_dataset_filename)
    elif dataset_name.lower() in molecule_net_dataset_names:
        return load_MolecueNet(dataset_dir, dataset_name, task)
    elif dataset_name.lower() in sentigraph_names:
        return load_SeniGraph(dataset_dir, dataset_name)




    else:
        raise NotImplementedError
```

```
        data_list.append(Data(x=torch.from_numpy(node_features[graph_idx]).float(),
                              edge_index=dense_to_sparse(torch.from_numpy(dense_edges[graph_idx]))[0],
                              y=torch.from_numpy(np.where(graph_labels[graph_idx])[0])))
    return data_list


def get_dataset(dataset_dir, dataset_name, task=None):
    sync_dataset_dict = {
        'BA_2Motifs'.lower(): 'BA_2Motifs',
        'BA_Shapes'.lower(): 'BA_shapes',
        'BA_Community'.lower(): 'BA_Community',
        'Tree_Cycle'.lower(): 'Tree_Cycle',
        'Tree_Grids'.lower(): 'Tree_Grids',
        'BA_LRP'.lower(): 'ba_lrp'
    }

    sentigraph_names = ['Graph_SST2', 'Graph_Twitter', 'Graph_SST5']
    sentigraph_names = [name.lower() for name in sentigraph_names]
    molecule_net_dataset_names = [name.lower() for name in MoleculeNet.names.keys()]

    if dataset_name.lower() == 'Mutagenicity'.lower():
        return load_MUTAG(dataset_dir, 'mutagenicity')
    elif dataset_name.lower() in sync_dataset_dict.keys():
        sync_dataset_filename = sync_dataset_dict[dataset_name.lower()]
        return load_syn_data(dataset_dir, sync_dataset_filename)
    elif dataset_name.lower() in molecule_net_dataset_names:
        return load_MolecueNet(dataset_dir, dataset_name, task)
    elif dataset_name.lower() in sentigraph_names:
        return load_SeniGraph(dataset_dir, dataset_name)
    elif dataset_name.lower() == "MalNetTiny".lower():
        return load_MalNetTiny(dataset_dir, 'malnettiny')
    else:
        raise NotImplementedError

class MalNetTiny(InMemoryDataset):
    r"""The MalNet Tiny dataset from the
    `"A Large-Scale Database for Graph Representation Learning"
    <https://openreview.net/pdf?id=1xDTDk3XPW>`_ paper.
    :class:`MalNetTiny` contains 5,000 malicious and benign software function
    call graphs across 5 different types. Each graph contains at most 5k nodes.

    Args:
        root (str): Root directory where the dataset should be saved.
        split (str, optional): If :obj:`"train"`, loads the training dataset.
            If :obj:`"val"`, loads the validation dataset.
            If :obj:`"trainval"`, loads the training and validation dataset.
            If :obj:`"test"`, loads the test dataset.
```

```
            If :obj:`None`, loads the entire dataset.
            (default: :obj:`None`)
        transform (callable, optional): A function/transform that takes in an
            :obj:`torch_geometric.data.Data` object and returns a transformed
            version. The data object will be transformed before every access.
            (default: :obj:`None`)
        pre_transform (callable, optional): A function/transform that takes in
            an :obj:`torch_geometric.data.Data` object and returns a
            transformed version. The data object will be transformed before
            being saved to disk. (default: :obj:`None`)
        pre_filter (callable, optional): A function that takes in an
            :obj:`torch_geometric.data.Data` object and returns a boolean
            value, indicating whether the data object should be included in the
            final dataset. (default: :obj:`None`)
    """

    data_url = ('http://malnet.cc.gatech.edu/'
                'graph-data/malnet-graphs-tiny.tar.gz')
    split_url = 'http://malnet.cc.gatech.edu/split-info/split_info_tiny.zip'
    splits = ['train', 'val', 'test']

    def __init__(
        self,
        root: str,
        name: str,
        split: Optional[str] = None,
        transform: Optional[Callable] = None,
        pre_transform: Optional[Callable] = None,
        pre_filter: Optional[Callable] = None,
    ):
        self.root = root
        self.name = name.lower()
        if split not in {'train', 'val', 'trainval', 'test', None}:
            raise ValueError(f'Split "{split}" found, but expected either '
                             f'"train", "val", "trainval", "test" or None')
        super().__init__(root, transform, pre_transform, pre_filter)
        self.data, self.slices = torch.load(self.processed_paths[0])

        if split is not None:
            split_slices = torch.load(self.processed_paths[1])
            if split == 'train':
                self._indices = range(split_slices[0], split_slices[1])
            elif split == 'val':
                self._indices = range(split_slices[1], split_slices[2])
            elif split == 'trainval':
                self._indices = range(split_slices[0], split_slices[2])
            elif split == 'test':
```

```
                self._indices = range(split_slices[2], split_slices[3])

    @property
    def raw_dir(self):
        return os.path.join(self.root, self.name, 'raw')

    @property
    def raw_file_names(self):
        return ['malnet-graphs-tiny', osp.join('split_info_tiny', 'type')]

    @property
    def processed_file_names(self):
        return ['data.pt', 'split_slices.pt']

    @property
    def processed_dir(self):
        return os.path.join(self.root, self.name, 'processed')

    def download(self):
        path = download_url(self.data_url, self.raw_dir)
        extract_tar(path, self.raw_dir)
        os.unlink(path)

        path = download_url(self.split_url, self.raw_dir)
        extract_zip(path, self.raw_dir)
        os.unlink(path)

    def process(self):
        y_map = {}
        data_list = []
        split_slices = [0]

        for split in ['train', 'val', 'test']:
            with open(osp.join(self.raw_paths[1], f'{split}.txt'), 'r') as f:
                filenames = f.read().split('\n')[:-1]
                split_slices.append(split_slices[-1] + len(filenames))

            for filename in filenames:
                path = osp.join(self.raw_paths[0], f'{filename}.edgelist')
                malware_type = filename.split('/')[0]
                y = y_map.setdefault(malware_type, len(y_map))

                with open(path, 'r') as f:
                    edges = f.read().split('\n')[5:-1]

                edge_index = [[int(s) for s in edge.split()] for edge in edges]
                edge_index = torch.tensor(edge_index).t().contiguous()
                num_nodes = int(edge_index.max()) + 1
                data = Data(edge_index=edge_index, y=y, num_nodes=num_nodes)
                data_list.append(data)

        if self.pre_filter is not None:
            data_list = [data for data in data_list if self.pre_filter(data)]

        if self.pre_transform is not None:
```

```
130
131 class MUTAGDataset(InMemoryDataset):
132     def __init__(self, root, name, transform=None, pre_t
    ransform=None):
133         self.root = root
134         self.name = name.lower()
135         super(MUTAGDataset, self).__init__(root, transfo
    rm, pre_transform)
136         self.data, self.slices = torch.load(self.process
    ed_paths[0])
137
138     def __len__(self):
139         return len(self.slices['x']) - 1
140
141     @property
142     def raw_dir(self):
143         return os.path.join(self.root, self.name, 'raw')
144
145     @property
146     def raw_file_names(self):
147         return ['Mutagenicity_A', 'Mutagenicity_graph_la
    bels', 'Mutagenicity_graph_indicator', 'Mutagenicity_nod
    e_labels']
148
149     @property
150     def processed_dir(self):
151         return os.path.join(self.root, self.name, 'proce
    ssed')
152
153     @property
154     def processed_file_names(self):
155         return ['data.pt']
156
157     def download(self):
158         url = 'https://www.chrsmrrs.com/graphkerneldatas
    ets'
159         folder = osp.join(self.root, self.name)
160         path = download_url(f'{url}/{self.name}.zip', fo
    lder)
161         extract_zip(path, folder)
162         os.unlink(path)
163         shutil.rmtree(self.raw_dir)
164         os.rename(osp.join(folder, self.name), self.raw_
    dir)
165
166     def process(self):
167         r"""Processes the dataset to the :obj:`self.proc
    essed_dir` folder."""
168         with open(os.path.join(self.raw_dir, 'Mutagenici
    ty_node_labels.txt'), 'r') as f:
169             nodes_all_temp = f.read().splitlines()
170             nodes_all = [int(i) for i in nodes_all_temp]
171
172         adj_all = np.zeros((len(nodes_all), len(nodes_al
    l)))
173         with open(os.path.join(self.raw_dir, 'Mutagenici
    ty_A.txt'), 'r') as f:
174             adj_list = f.read().splitlines()
175         for item in adj_list:
```

```
249             data_list = [self.pre_transform(data) for da
    ta in data_list]
250
251         torch.save(self.collate(data_list), self.process
    ed_paths[0])
252         torch.save(split_slices, self.processed_paths
    [1])
253
254
255
256 class MUTAGDataset(InMemoryDataset):
257     def __init__(self, root, name, transform=None, pre_t
    ransform=None):
258         self.root = root
259         self.name = name.lower()
260         super(MUTAGDataset, self).__init__(root, transfo
    rm, pre_transform)
261         self.data, self.slices = torch.load(self.process
    ed_paths[0])
262
263     def __len__(self):
264         return len(self.slices['x']) - 1
265
266     @property
267     def raw_dir(self):
268         return os.path.join(self.root, self.name, 'raw')
269
270     @property
271     def raw_file_names(self):
272         return ['Mutagenicity_A', 'Mutagenicity_graph_la
    bels', 'Mutagenicity_graph_indicator', 'Mutagenicity_nod
    e_labels']
273
274     @property
275     def processed_dir(self):
276         return os.path.join(self.root, self.name, 'proce
    ssed')
277
278     @property
279     def processed_file_names(self):
280         return ['data.pt']
281
282     def download(self):
283         url = 'https://www.chrsmrrs.com/graphkerneldatas
    ets'
284         folder = osp.join(self.root, self.name)
285         path = download_url(f'{url}/{self.name}.zip', fo
    lder)
286         extract_zip(path, folder)
287         os.unlink(path)
288         shutil.rmtree(self.raw_dir)
289         os.rename(osp.join(folder, self.name), self.raw_
    dir)
290
291     def process(self):
292         r"""Processes the dataset to the :obj:`self.proc
    essed_dir` folder."""
293         with open(os.path.join(self.raw_dir, 'Mutagenici
    ty_node_labels.txt'), 'r') as f:
294             nodes_all_temp = f.read().splitlines()
295             nodes_all = [int(i) for i in nodes_all_temp]
296
297         adj_all = np.zeros((len(nodes_all), len(nodes_al
    l)))
298         with open(os.path.join(self.raw_dir, 'Mutagenici
    ty_A.txt'), 'r') as f:
299             adj_list = f.read().splitlines()
300         for item in adj_list:
```

```
            lr = item.split(', ')
            l = int(lr[0])
            r = int(lr[1])
            adj_all[l - 1, r - 1] = 1

        with open(os.path.join(self.raw_dir, 'Mutagenicity_graph_indicator.txt'), 'r') as f:
            graph_indicator_temp = f.read().splitlines()
            graph_indicator = [int(i) for i in graph_indicator_temp]
            graph_indicator = np.array(graph_indicator)

        with open(os.path.join(self.raw_dir, 'Mutagenicity_graph_labels.txt'), 'r') as f:
            graph_labels_temp = f.read().splitlines()
            graph_labels = [int(i) for i in graph_labels_temp]

        data_list = []
        for i in range(1, 4338):
            idx = np.where(graph_indicator == i)
            graph_len = len(idx[0])
            adj = adj_all[idx[0][0]:idx[0][0] + graph_len, idx[0][0]:idx[0][0] + graph_len]
            label = int(graph_labels[i - 1] == 1)
            feature = nodes_all[idx[0][0]:idx[0][0] + graph_len]
            nb_clss = 14
            targets = np.array(feature).reshape(-1)
            one_hot_feature = np.eye(nb_clss)[targets]
            data_example = Data(x=torch.from_numpy(one_hot_feature).float(),
                                edge_index=dense_to_sparse(torch.from_numpy(adj))[0],
                                y=label)
            data_list.append(data_example)

        torch.save(self.collate(data_list), self.processed_paths[0])


class SentiGraphDataset(InMemoryDataset):
    def __init__(self, root, name, transform=None, pre_transform=undirected_graph):
        self.name = name
        super(SentiGraphDataset, self).__init__(root, transform, pre_transform)
        self.data, self.slices, self.supplement = torch.load(self.processed_paths[0])

    @property
    def raw_dir(self):
        return osp.join(self.root, self.name, 'raw')

    @property
    def processed_dir(self):
        return osp.join(self.root, self.name, 'processed')

    @property
    def raw_file_names(self):
        return ['node_features', 'node_indicator', 'sentence_tokens', 'edge_index',
                'graph_labels', 'split_indices']

    @property
    def processed_file_names(self):
```

```
            lr = item.split(', ')
            l = int(lr[0])
            r = int(lr[1])
            adj_all[l - 1, r - 1] = 1

        with open(os.path.join(self.raw_dir, 'Mutagenicity_graph_indicator.txt'), 'r') as f:
            graph_indicator_temp = f.read().splitlines()
            graph_indicator = [int(i) for i in graph_indicator_temp]
            graph_indicator = np.array(graph_indicator)

        with open(os.path.join(self.raw_dir, 'Mutagenicity_graph_labels.txt'), 'r') as f:
            graph_labels_temp = f.read().splitlines()
            graph_labels = [int(i) for i in graph_labels_temp]

        data_list = []
        for i in range(1, 4338):
            idx = np.where(graph_indicator == i)
            graph_len = len(idx[0])
            adj = adj_all[idx[0][0]:idx[0][0] + graph_len, idx[0][0]:idx[0][0] + graph_len]
            label = int(graph_labels[i - 1] == 1)
            feature = nodes_all[idx[0][0]:idx[0][0] + graph_len]
            nb_clss = 14
            targets = np.array(feature).reshape(-1)
            one_hot_feature = np.eye(nb_clss)[targets]
            data_example = Data(x=torch.from_numpy(one_hot_feature).float(),
                                edge_index=dense_to_sparse(torch.from_numpy(adj))[0],
                                y=label)
            data_list.append(data_example)

        torch.save(self.collate(data_list), self.processed_paths[0])


class SentiGraphDataset(InMemoryDataset):
    def __init__(self, root, name, transform=None, pre_transform=undirected_graph):
        self.name = name
        super(SentiGraphDataset, self).__init__(root, transform, pre_transform)
        self.data, self.slices, self.supplement = torch.load(self.processed_paths[0])

    @property
    def raw_dir(self):
        return osp.join(self.root, self.name, 'raw')

    @property
    def processed_dir(self):
        return osp.join(self.root, self.name, 'processed')

    @property
    def raw_file_names(self):
        return ['node_features', 'node_indicator', 'sentence_tokens', 'edge_index',
                'graph_labels', 'split_indices']

    @property
    def processed_file_names(self):
```

```
        return ['data.pt']

    def process(self):
        # Read data into huge `Data` list.
        self.data, self.slices, self.supplement \
              = read_sentigraph_data(self.raw_dir, self.name)

        if self.pre_filter is not None:
            data_list = [self.get(idx) for idx in range(len(self))]
            data_list = [data for data in data_list if self.pre_filter(data)]
            self.data, self.slices = self.collate(data_list)

        if self.pre_transform is not None:
            data_list = [self.get(idx) for idx in range(len(self))]
            data_list = [self.pre_transform(data) for data in data_list]
            self.data, self.slices = self.collate(data_list)
        torch.save((self.data, self.slices, self.supplement), self.processed_paths[0])


class SynGraphDataset(InMemoryDataset):
    def __init__(self, root, name, transform=None, pre_transform=None):
        self.name = name
        super(SynGraphDataset, self).__init__(root, transform, pre_transform)
        self.data, self.slices = torch.load(self.processed_paths[0])

    @property
    def raw_dir(self):
        return osp.join(self.root, self.name, 'raw')

    @property
    def processed_dir(self):
        return osp.join(self.root, self.name, 'processed')

    @property
    def raw_file_names(self):
        return [f"{self.name}.pkl"]

    @property
    def processed_file_names(self):
        return ['data.pt']

    def process(self):
        # Read data into huge `Data` list.
        data = read_syn_data(self.raw_dir, self.name)
        data = data if self.pre_transform is None else self.pre_transform(data)
        torch.save(self.collate([data]), self.processed_paths[0])


class BA2MotifDataset(InMemoryDataset):
    def __init__(self, root, name, transform=None, pre_transform=None):
        self.name = name
        super(BA2MotifDataset, self).__init__(root, transform, pre_transform)
```


```
        return ['data.pt']

    def process(self):
        # Read data into huge `Data` list.
        self.data, self.slices, self.supplement \
              = read_sentigraph_data(self.raw_dir, self.name)

        if self.pre_filter is not None:
            data_list = [self.get(idx) for idx in range(len(self))]
            data_list = [data for data in data_list if self.pre_filter(data)]
            self.data, self.slices = self.collate(data_list)

        if self.pre_transform is not None:
            data_list = [self.get(idx) for idx in range(len(self))]
            data_list = [self.pre_transform(data) for data in data_list]
            self.data, self.slices = self.collate(data_list)
        torch.save((self.data, self.slices, self.supplement), self.processed_paths[0])


class SynGraphDataset(InMemoryDataset):
    def __init__(self, root, name, transform=None, pre_transform=None):
        self.name = name
        super(SynGraphDataset, self).__init__(root, transform, pre_transform)
        self.data, self.slices = torch.load(self.processed_paths[0])

    @property
    def raw_dir(self):
        return osp.join(self.root, self.name, 'raw')

    @property
    def processed_dir(self):
        return osp.join(self.root, self.name, 'processed')

    @property
    def raw_file_names(self):
        return [f"{self.name}.pkl"]

    @property
    def processed_file_names(self):
        return ['data.pt']

    def process(self):
        # Read data into huge `Data` list.
        data = read_syn_data(self.raw_dir, self.name)
        data = data if self.pre_transform is None else self.pre_transform(data)
        torch.save(self.collate([data]), self.processed_paths[0])


class BA2MotifDataset(InMemoryDataset):
    def __init__(self, root, name, transform=None, pre_transform=None):
        self.name = name
        super(BA2MotifDataset, self).__init__(root, transform, pre_transform)
```

```
        self.data, self.slices = torch.load(self.processed_paths[0])

    @property
    def raw_dir(self):
        return osp.join(self.root, self.name, 'raw')

    @property
    def processed_dir(self):
        return osp.join(self.root, self.name, 'processed')

    @property
    def raw_file_names(self):
        return [f"{self.name}.pkl"]

    @property
    def processed_file_names(self):
        return ['data.pt']

    def process(self):
        # Read data into huge `Data` list.
        data_list = read_ba2motif_data(self.raw_dir, self.name)

        if self.pre_filter is not None:
            data_list = [self.get(idx) for idx in range(len(self))]
            data_list = [data for data in data_list if self.pre_filter(data)]
            self.data, self.slices = self.collate(data_list)

        if self.pre_transform is not None:
            data_list = [self.get(idx) for idx in range(len(self))]
            data_list = [self.pre_transform(data) for data in data_list]
            self.data, self.slices = self.collate(data_list)

        torch.save(self.collate(data_list), self.processed_paths[0])


def load_MUTAG(dataset_dir, dataset_name):
    """ 188 molecules where label = 1 denotes mutagenic effect """
    dataset = MUTAGDataset(root=dataset_dir, name=dataset_name)
```

```
    return dataset


class BA_LRP(InMemoryDataset):

    def __init__(self, root, num_per_class, transform=None, pre_transform=None):
        self.num_per_class = num_per_class
        super().__init__(root, transform, pre_transform)
        self.data, self.slices = torch.load(self.processed_paths[0])

    @property
```

```
        self.data, self.slices = torch.load(self.processed_paths[0])

    @property
    def raw_dir(self):
        return osp.join(self.root, self.name, 'raw')

    @property
    def processed_dir(self):
        return osp.join(self.root, self.name, 'processed')

    @property
    def raw_file_names(self):
        return [f"{self.name}.pkl"]

    @property
    def processed_file_names(self):
        return ['data.pt']

    def process(self):
        # Read data into huge `Data` list.
        data_list = read_ba2motif_data(self.raw_dir, self.name)

        if self.pre_filter is not None:
            data_list = [self.get(idx) for idx in range(len(self))]
            data_list = [data for data in data_list if self.pre_filter(data)]
            self.data, self.slices = self.collate(data_list)

        if self.pre_transform is not None:
            data_list = [self.get(idx) for idx in range(len(self))]
            data_list = [self.pre_transform(data) for data in data_list]
            self.data, self.slices = self.collate(data_list)

        torch.save(self.collate(data_list), self.processed_paths[0])


def load_MUTAG(dataset_dir, dataset_name):
    """ 188 molecules where label = 1 denotes mutagenic effect """
    dataset = MUTAGDataset(root=dataset_dir, name=dataset_name)
    return dataset

def load_MalNetTiny(dataset_dir, dataset_name):

    dataset = MalNetTiny(root=dataset_dir, name=dataset_name)
    return dataset


class BA_LRP(InMemoryDataset):

    def __init__(self, root, num_per_class, transform=None, pre_transform=None):
        self.num_per_class = num_per_class
        super().__init__(root, transform, pre_transform)
        self.data, self.slices = torch.load(self.processed_paths[0])

    @property
```

```
    def processed_file_names(self):
        return [f'data{self.num_per_class}.pt']

    def gen_class1(self):
        x = torch.tensor([[1], [1]], dtype=torch.float)
        edge_index = torch.tensor([[0, 1], [1, 0]], dtyp
e=torch.long)
        data = Data(x=x, edge_index=edge_index, y=torch.
tensor([[0]], dtype=torch.float))

        for i in range(2, 20):
            data.x = torch.cat([data.x, torch.tensor
([[1]], dtype=torch.float)], dim=0)
            deg = torch.stack([(data.edge_index[0] == no
de_idx).float().sum() for node_idx in range(i)], dim=0)
            sum_deg = deg.sum(dim=0, keepdim=True)
            probs = (deg / sum_deg).unsqueeze(0)
            prob_dist = torch.distributions.Categorical
(probs)
            node_pick = prob_dist.sample().squeeze()
            data.edge_index = torch.cat([data.edge_inde
x,
                                         torch.tensor
([[node_pick, i], [i, node_pick]], dtype=torch.long)], d
im=1)

        return data

    def gen_class2(self):
        x = torch.tensor([[1], [1]], dtype=torch.float)
        edge_index = torch.tensor([[0, 1], [1, 0]], dtyp
e=torch.long)
        data = Data(x=x, edge_index=edge_index, y=torch.
tensor([[1]], dtype=torch.float))
        epsilon = 1e-30

        for i in range(2, 20):
            data.x = torch.cat([data.x, torch.tensor
([[1]], dtype=torch.float)], dim=0)
            deg_reciprocal = torch.stack([1 / ((data.edg
e_index[0] == node_idx).float().sum() + epsilon) for nod
e_idx in range(i)], dim=0)
            sum_deg_reciprocal = deg_reciprocal.sum(dim=
0, keepdim=True)
            probs = (deg_reciprocal / sum_deg_reciproca
l).unsqueeze(0)
            prob_dist = torch.distributions.Categorical
(probs)
            node_pick = -1
            for _ in range(1 if i % 5 != 4 else 2):
                new_node_pick = prob_dist.sample().squee
ze()
                while new_node_pick == node_pick:
                    new_node_pick = prob_dist.sample().s
queeze()
                node_pick = new_node_pick
                data.edge_index = torch.cat([data.edge_i
ndex,
                                             torch.tenso
r([[node_pick, i], [i, node_pick]], dtype=torch.long)],
 dim=1)

        return data

    def process(self):
        data_list = []
        for i in range(self.num_per_class):
```

```
    def processed_file_names(self):
        return [f'data{self.num_per_class}.pt']

    def gen_class1(self):
        x = torch.tensor([[1], [1]], dtype=torch.float)
        edge_index = torch.tensor([[0, 1], [1, 0]], dtyp
e=torch.long)
        data = Data(x=x, edge_index=edge_index, y=torch.
tensor([[0]], dtype=torch.float))

        for i in range(2, 20):
            data.x = torch.cat([data.x, torch.tensor
([[1]], dtype=torch.float)], dim=0)
            deg = torch.stack([(data.edge_index[0] == no
de_idx).float().sum() for node_idx in range(i)], dim=0)
            sum_deg = deg.sum(dim=0, keepdim=True)
            probs = (deg / sum_deg).unsqueeze(0)
            prob_dist = torch.distributions.Categorical
(probs)
            node_pick = prob_dist.sample().squeeze()
            data.edge_index = torch.cat([data.edge_inde
x,
                                         torch.tensor
([[node_pick, i], [i, node_pick]], dtype=torch.long)], d
im=1)

        return data

    def gen_class2(self):
        x = torch.tensor([[1], [1]], dtype=torch.float)
        edge_index = torch.tensor([[0, 1], [1, 0]], dtyp
e=torch.long)
        data = Data(x=x, edge_index=edge_index, y=torch.
tensor([[1]], dtype=torch.float))
        epsilon = 1e-30

        for i in range(2, 20):
            data.x = torch.cat([data.x, torch.tensor
([[1]], dtype=torch.float)], dim=0)
            deg_reciprocal = torch.stack([1 / ((data.edg
e_index[0] == node_idx).float().sum() + epsilon) for nod
e_idx in range(i)], dim=0)
            sum_deg_reciprocal = deg_reciprocal.sum(dim=
0, keepdim=True)
            probs = (deg_reciprocal / sum_deg_reciproca
l).unsqueeze(0)
            prob_dist = torch.distributions.Categorical
(probs)
            node_pick = -1
            for _ in range(1 if i % 5 != 4 else 2):
                new_node_pick = prob_dist.sample().squee
ze()
                while new_node_pick == node_pick:
                    new_node_pick = prob_dist.sample().s
queeze()
                node_pick = new_node_pick
                data.edge_index = torch.cat([data.edge_i
ndex,
                                             torch.tenso
r([[node_pick, i], [i, node_pick]], dtype=torch.long)],
 dim=1)

        return data

    def process(self):
        data_list = []
        for i in range(self.num_per_class):
```

```
            data_list.append(self.gen_class1())
            data_list.append(self.gen_class2())

        data, slices = self.collate(data_list)
        torch.save((data, slices), self.processed_paths[0])


def load_syn_data(dataset_dir, dataset_name):
    """ The synthetic dataset """
    if dataset_name.lower() == 'BA_2Motifs'.lower():
        dataset = BA2MotifDataset(root=dataset_dir, name=dataset_name)
    elif dataset_name.lower() == 'BA_LRP'.lower():
        dataset = BA_LRP(root=os.path.join(dataset_dir, 'ba_lrp'), num_per_class=10000)
    else:
        dataset = SynGraphDataset(root=dataset_dir, name=dataset_name)
    dataset.node_type_dict = {k: v for k, v in enumerate(range(dataset.num_classes))}
    dataset.node_color = None
    return dataset


def load_MolecueNet(dataset_dir, dataset_name, task=None):
    """ Attention the multi-task problems not solved yet """
    molecule_net_dataset_names = {name.lower(): name for name in MoleculeNet.names.keys()}
    dataset = MoleculeNet(root=dataset_dir, name=molecule_net_dataset_names[dataset_name.lower()])
    dataset.data.x = dataset.data.x.float()
    if task is None:
        dataset.data.y = dataset.data.y.squeeze().long()
    else:
        dataset.data.y = dataset.data.y[task].long()
    dataset.node_type_dict = None
    dataset.node_color = None
    return dataset


def load_SeniGraph(dataset_dir, dataset_name):
    dataset = SentiGraphDataset(root=dataset_dir, name=dataset_name)
    return dataset


def get_dataloader(dataset, batch_size, random_split_flag=True, data_split_ratio=None, seed=2):
    """
    Args:
        dataset:
        batch_size: int
        random_split_flag: bool
        data_split_ratio: list, training, validation and testing ratio
        seed: random seed to split the dataset randomly
    Returns:
        a dictionary of training, validation, and testing dataLoader
    """

    if not random_split_flag and hasattr(dataset, 'supplement'):
```

```
            data_list.append(self.gen_class1())
            data_list.append(self.gen_class2())

        data, slices = self.collate(data_list)
        torch.save((data, slices), self.processed_paths[0])


def load_syn_data(dataset_dir, dataset_name):
    """ The synthetic dataset """
    if dataset_name.lower() == 'BA_2Motifs'.lower():
        dataset = BA2MotifDataset(root=dataset_dir, name=dataset_name)
    elif dataset_name.lower() == 'BA_LRP'.lower():
        dataset = BA_LRP(root=os.path.join(dataset_dir, 'ba_lrp'), num_per_class=10000)
    else:
        dataset = SynGraphDataset(root=dataset_dir, name=dataset_name)
    dataset.node_type_dict = {k: v for k, v in enumerate(range(dataset.num_classes))}
    dataset.node_color = None
    return dataset


def load_MolecueNet(dataset_dir, dataset_name, task=None):
    """ Attention the multi-task problems not solved yet """
    molecule_net_dataset_names = {name.lower(): name for name in MoleculeNet.names.keys()}
    dataset = MoleculeNet(root=dataset_dir, name=molecule_net_dataset_names[dataset_name.lower()])
    dataset.data.x = dataset.data.x.float()
    if task is None:
        dataset.data.y = dataset.data.y.squeeze().long()
    else:
        dataset.data.y = dataset.data.y[task].long()
    dataset.node_type_dict = None
    dataset.node_color = None
    return dataset


def load_SeniGraph(dataset_dir, dataset_name):
    dataset = SentiGraphDataset(root=dataset_dir, name=dataset_name)
    return dataset


def get_dataloader(dataset, batch_size, random_split_flag=True, data_split_ratio=None, seed=2):
    """
    Args:
        dataset:
        batch_size: int
        random_split_flag: bool
        data_split_ratio: list, training, validation and testing ratio
        seed: random seed to split the dataset randomly
    Returns:
        a dictionary of training, validation, and testing dataLoader
    """

    if not random_split_flag and hasattr(dataset, 'supplement'):
```

```
        assert 'split_indices' in dataset.supplement.keys(), "split idx"
        split_indices = dataset.supplement['split_indices']
        train_indices = torch.where(split_indices == 0)[0].numpy().tolist()
        dev_indices = torch.where(split_indices == 1)[0].numpy().tolist()
        test_indices = torch.where(split_indices == 2)[0].numpy().tolist()

        train = Subset(dataset, train_indices)
        eval = Subset(dataset, dev_indices)
        test = Subset(dataset, test_indices)
    else:
        num_train = int(data_split_ratio[0] * len(dataset))
        num_eval = int(data_split_ratio[1] * len(dataset))
        num_test = len(dataset) - num_train - num_eval

        train, eval, test = random_split(dataset, lengths=[num_train, num_eval, num_test],
                                         generator=torch.Generator().manual_seed(seed))

    dataloader = dict()
    dataloader['train'] = DataLoader(train, batch_size=batch_size, shuffle=True)
    dataloader['eval'] = DataLoader(eval, batch_size=batch_size, shuffle=False)
    dataloader['test'] = DataLoader(test, batch_size=batch_size, shuffle=False)
    return dataloader


if __name__ == '__main__':
    get_dataset(dataset_dir='./datasets', dataset_name='bbbp')
```


```
        assert 'split_indices' in dataset.supplement.keys(), "split idx"
        split_indices = dataset.supplement['split_indices']
        train_indices = torch.where(split_indices == 0)[0].numpy().tolist()
        dev_indices = torch.where(split_indices == 1)[0].numpy().tolist()
        test_indices = torch.where(split_indices == 2)[0].numpy().tolist()

        train = Subset(dataset, train_indices)
        eval = Subset(dataset, dev_indices)
        test = Subset(dataset, test_indices)
    else:
        num_train = int(data_split_ratio[0] * len(dataset))
        num_eval = int(data_split_ratio[1] * len(dataset))
        num_test = len(dataset) - num_train - num_eval

        train, eval, test = random_split(dataset, lengths=[num_train, num_eval, num_test],
                                         generator=torch.Generator().manual_seed(seed))

    dataloader = dict()
    dataloader['train'] = DataLoader(train, batch_size=batch_size, shuffle=True)
    dataloader['eval'] = DataLoader(eval, batch_size=batch_size, shuffle=False)
    dataloader['test'] = DataLoader(test, batch_size=batch_size, shuffle=False)
    return dataloader


if __name__ == '__main__':
    get_dataset(dataset_dir='./datasets', dataset_name='bbbp')
```