4.6 Loading data from CSV files

Comma Separated Value (CSV) is a widely used data storage format. DGL provides `CSVDataset` for loading and parsing graph data stored in CSV format.

To create a `CSVDataset` object:

```
import dgl
ds = dgl.data.CSVDataset('/path/to/dataset')
```

The returned `ds` object is a standard `DGLDataset`. For example, one can get graph samples using `__getitem__` as well as node/edge features using `ndata` / `edata`.

```
# A demonstration of how to use the loaded dataset. The feature names
# may vary depending on the CSV contents.
g = ds[0] # get the graph
label = g.ndata['label']
feat = g.ndata['feat']
```

Data folder structure

```
/path/to/dataset/
|-- meta.yaml     # metadata of the dataset
|-- edges_0.csv   # edge data including src_id, dst_id, feature, label and so on
|-- ...           # you can have as many CSVs for edge data as you want
|-- nodes_0.csv   # node data including node_id, feature, label and so on
|-- ...           # you can have as many CSVs for node data as you want
|-- graphs.csv    # graph-level features
```

Node/edge/graph-level data are stored in CSV files. `meta.yaml` is a metadata file specifying where to read nodes/edges/graphs data and how to parse them to construct the dataset object. A minimal data folder contains one `meta.yaml` and two CSVs, one for node data and one for edge data, in which case the dataset contains only a single graph with no graph-level data.

Dataset of a single feature-less graph

When the dataset contains only one graph with no node or edge features, there need only three files in the data folder: `meta.yaml`, one CSV for node IDs and one CSV for edges:

```
./mini_featureless_dataset/
|-- meta.yaml
|-- nodes.csv
|-- edges.csv
```

`meta.yaml` contains the following information:

```yaml
dataset_name: mini_featureless_dataset
edge_data:
- file_name: edges.csv
node_data:
- file_name: nodes.csv
```

`nodes.csv` lists the node IDs under the `node_id` field:

```
node_id
0
1
2
3
4
```

`edges.csv` lists all the edges in two columns (`src_id` and `dst_id`) specifying the source and destination node ID of each edge:

```
src_id,dst_id
4,4
4,1
3,0
4,1
4,0
1,2
1,3
3,3
1,1
4,1
```

After loaded, the dataset has one graph without any features:

```
>>> import dgl
>>> dataset = dgl.data.CSVDataset('./mini_featureless_dataset')
>>> g = dataset[0]  # only one graph
>>> print(g)
Graph(num_nodes=5, num_edges=10,
      ndata_schemes={}
      edata_schemes={})
```

> **Note**
>
> Non-integer node IDs are allowed. When constructing the graph, `CSVDataset` will map each raw ID to an integer ID starting from zero. If the node IDs are already distinct integers from 0 to `num_nodes-1`, no mapping is applied.

> **Note**
>
> Edges are always directed. To have both directions, add reversed edges in the edge CSV file or use `AddReverse` to transform the loaded graph.

A graph without any feature is often of less interest. In the next example, we will show how to load and parse node or edge features.

Dataset of a single graph with features and labels

When the dataset contains a single graph with node or edge features and labels, there still need only three files in the data folder: `meta.yaml`, one CSV for node IDs and one CSV for edges:

```
./mini_feature_dataset/
|-- meta.yaml
|-- nodes.csv
|-- edges.csv
```

`meta.yaml`:

```
dataset_name: mini_feature_dataset
edge_data:
- file_name: edges.csv
node_data:
- file_name: nodes.csv
```

`edges.csv` with five synthetic edge data (`label`, `train_mask`, `val_mask`, `test_mask`, `feat`):

```
src_id,dst_id,label,train_mask,val_mask,test_mask,feat
4,0,2,False,True,True,"0.5477868606453535, 0.4470617033458436, 0.936706701616337"
4,0,0,False,False,True,"0.9794634290792008, 0.23682038840665198, 0.049629338970987646"
0,3,1,True,True,True,"0.8586722047523594, 0.5746912787380253, 0.6462162561249654"
0,1,2,True,False,False,"0.2730008213674695, 0.5937484188166621, 0.765544096939567"
0,2,1,True,True,True,"0.45441619816038514, 0.1681403185591509, 0.9952376085297715"
0,0,0,False,False,False,"0.4197669213305396, 0.849983324532477, 0.16974127573016262"
2,2,1,False,True,True,"0.5495035052928215, 0.21394654203489705, 0.7174910641836348"
1,0,2,False,True,False,"0.008790817766266334, 0.4216530595907526, 0.529195480661293"
3,0,0,True,True,True,"0.6598715708878852, 0.1932390907048961, 0.9774471538377553"
4,0,1,False,False,False,"0.16846068931179736, 0.41516080644186737, 0.002158116134429955"
```

`nodes.csv` with five synthetic node data (`label`, `train_mask`, `val_mask`, `test_mask`, `feat`):

```
node_id,label,train_mask,val_mask,test_mask,feat
0,1,False,True,True,"0.07816474278491703, 0.9137336384979067, 0.4654086994009452"
1,1,True,True,True,"0.05354099924658973, 0.8753101998792645, 0.33929432608774135"
2,1,True,False,True,"0.33234211884156384, 0.9370522452510665, 0.6694943496824788"
3,0,False,True,False,"0.9784264442230887, 0.22131880861864428, 0.3161154827254189"
4,1,True,True,False,"0.23142237259162102, 0.8715767748481147, 0.19117861103555467"
```

After loaded, the dataset has one graph. Node/edge features are stored in `ndata` and `edata` with the same column names. The example demonstrates how to specify a vector-shaped feature using comma-separated list enclosed by double quotes `"..."`.

```
>>> import dgl
>>> dataset = dgl.data.CSVDataset('./mini_feature_dataset')
>>> g = dataset[0]  # only one graph
>>> print(g)
Graph(num_nodes=5, num_edges=10,
      ndata_schemes={'label': Scheme(shape=(), dtype=torch.int64), 'train_mask': Scheme(shape=
(), dtype=torch.bool), 'val_mask': Scheme(shape=(), dtype=torch.bool), 'test_mask':
Scheme(shape=(), dtype=torch.bool), 'feat': Scheme(shape=(3,), dtype=torch.float64)}
      edata_schemes={'label': Scheme(shape=(), dtype=torch.int64), 'train_mask': Scheme(shape=
(), dtype=torch.bool), 'val_mask': Scheme(shape=(), dtype=torch.bool), 'test_mask':
Scheme(shape=(), dtype=torch.bool), 'feat': Scheme(shape=(3,), dtype=torch.float64)})
```

Note

By default, `CSVDatatset` assumes all feature data to be numerical values (e.g., int, float, bool or list) and missing values are not allowed. Users could provide custom data parser for these cases. See Custom Data Parser for more details.

Dataset of a single heterogeneous graph

One can specify multiple node and edge CSV files (each for one type) to represent a heterogeneous graph. Here is an example data with two node types and two edge types:

```
./mini_hetero_dataset/
|-- meta.yaml
|-- nodes_0.csv
|-- nodes_1.csv
|-- edges_0.csv
|-- edges_1.csv
```

The `meta.yaml` specifies the node type name (using `ntype`) and edge type name (using `etype`) of each CSV file. The edge type name is a string triplet containing the source node type name, relation name and the destination node type name.

```yaml
dataset_name: mini_hetero_dataset
edge_data:
- file_name: edges_0.csv
  etype: [user, follow, user]
- file_name: edges_1.csv
  etype: [user, like, item]
node_data:
- file_name: nodes_0.csv
  ntype: user
- file_name: nodes_1.csv
  ntype: item
```

The node and edge CSV files follow the same format as in homogeneous graphs. Here are some synthetic data for demonstration purposes:

`edges_0.csv` and `edges_1.csv`:

```
src_id,dst_id,label,feat
4,4,1,"0.736833152378035,0.10522806046048205,0.9418796835016118"
3,4,2,"0.5749339182767451,0.20181320245665535,0.490938012147181"
1,4,2,"0.7697294432580938,0.49397782380750765,0.10864079337442234"
0,4,0,"0.1364240150959487,0.1393107840629273,0.7901988878812207"
2,3,1,"0.42988138237505735,0.18389137408509248,0.18431292077750894"
0,4,2,"0.8613368738351794,0.67985810014162,0.6580438064356824"
2,4,1,"0.6594951663841697,0.26499036865016423,0.7891429392727503"
4,1,0,"0.36649684241348557,0.9511783938523962,0.8494919263589972"
1,1,2,"0.698592283371875,0.038622249776255946,0.5563827995742111"
0,4,1,"0.5227112950269823,0.3148264185956532,0.47562693094002173"
```

`nodes_0.csv` and `nodes_1.csv`:

```
node_id,label,feat
0,2,"0.5400687466285844,0.7588441197954202,0.4268254673041745"
1,1,"0.08680051341900807,0.11446843700743892,0.7196969604886617"
2,2,"0.8964389655603473,0.23368113896545695,0.8813472954005022"
3,1,"0.5454703921677284,0.7819383771535038,0.3027939452162367"
4,1,"0.5365210052235699,0.8975240205792763,0.7613943085507672"
```

After loaded, the dataset has one heterograph with features and labels:

```
>>> import dgl
>>> dataset = dgl.data.CSVDataset('./mini_hetero_dataset')
>>> g = dataset[0]  # only one graph
>>> print(g)
Graph(num_nodes={'item': 5, 'user': 5},
      num_edges={('user', 'follow', 'user'): 10, ('user', 'like', 'item'): 10},
      metagraph=[('user', 'user', 'follow'), ('user', 'item', 'like')])
>>> g.nodes['user'].data
{'label': tensor([2, 1, 2, 1, 1]), 'feat': tensor([[0.5401, 0.7588, 0.4268],
        [0.0868, 0.1145, 0.7197],
        [0.8964, 0.2337, 0.8813],
        [0.5455, 0.7819, 0.3028],
        [0.5365, 0.8975, 0.7614]], dtype=torch.float64)}
>>> g.edges['like'].data
{'label': tensor([1, 2, 2, 0, 1, 2, 1, 0, 2, 1]), 'feat': tensor([[0.7368, 0.1052, 0.9419],
        [0.5749, 0.2018, 0.4909],
        [0.7697, 0.4940, 0.1086],
        [0.1364, 0.1393, 0.7902],
        [0.4299, 0.1839, 0.1843],
        [0.8613, 0.6799, 0.6580],
        [0.6595, 0.2650, 0.7891],
        [0.3665, 0.9512, 0.8495],
        [0.6986, 0.0386, 0.5564],
        [0.5227, 0.3148, 0.4756]], dtype=torch.float64)}
```

Dataset of multiple graphs

When there are multiple graphs, one can include an additional CSV file for storing graph-level features. Here is an example:

```
./mini_multi_dataset/
|-- meta.yaml
|-- nodes.csv
|-- edges.csv
|-- graphs.csv
```

Accordingly, the `meta.yaml` should include an extra `graph_data` key to tell which CSV file to load graph-level features from.

```yaml
dataset_name: mini_multi_dataset
edge_data:
- file_name: edges.csv
node_data:
- file_name: nodes.csv
graph_data:
  file_name: graphs.csv
```

To distinguish nodes and edges of different graphs, the `node.csv` and `edge.csv` must contain an extra column `graph_id`:

`edges.csv`:

```
graph_id,src_id,dst_id,feat
0,0,4,"0.39534097273254654,0.9422093637539785,0.634899790318452"
0,3,0,"0.04486384200747007,0.6453746567017163,0.8757520744192612"
0,3,2,"0.9397636966928355,0.6526403892728874,0.8643238446466464"
0,1,1,"0.40559906615287566,0.9848072295736628,0.493888090726854"
0,4,1,"0.253458867276219,0.9168191778828504,0.47224962583565544"
0,0,1,"0.3219496197945605,0.3439899477636117,0.7051530741717352"
0,2,1,"0.692873149428549,0.4770019763881086,0.21937428942781778"
0,4,0,"0.620118223673067,0.08691420300562658,0.86573472329756"
0,2,1,"0.00743445923710373,0.5251800239734318,0.05401638555202384"
0,4,1,"0.6776417760682221,0.7291568018841328,0.4523600060547709"
1,1,3,"0.6375445528248924,0.04878384701995819,0.4081642382536248"
1,0,4,"0.776002616178397,0.8851294998284638,0.7321742043493028"
1,1,0,"0.0928555079874982,0.6156748364694707,0.6985674921582508"
1,0,2,"0.31328748118329997,0.8326121496142408,0.04133991340612775"
1,1,0,"0.36786902637778773,0.39161865931662243,0.9971749359397111"
1,1,1,"0.4647410679872376,0.8478810655406659,0.6746269314422184"
1,0,2,"0.8117650553546695,0.7893727601272978,0.41527155506593394"
1,1,3,"0.40707309111756307,0.2796588354307046,0.34846782265758314"
1,1,0,"0.18626464175355095,0.3523777809254057,0.7863421810531344"
1,3,0,"0.28357022069634585,0.13774964202156292,0.5913335505943637"
```

`nodes.csv`:

```
graph_id,node_id,feat
0,0,"0.5725330322207948,0.8451870383322376,0.44412796119211184"
0,1,"0.6624186423087752,0.6118386331195641,0.7352138669985214"
0,2,"0.7583372765843964,0.15218126307872892,0.6810484348765842"
0,3,"0.14627522432017592,0.7457985352827006,0.1037097085190507"
0,4,"0.49037522512771525,0.8778998699783784,0.0911194482288028"
1,0,"0.11158102039672668,0.08543289788089736,0.6901745368284345"
1,1,"0.28367647637469273,0.07502571020414439,0.01217200152200748"
1,2,"0.2472495901894738,0.24285506608575758,0.6494437360242048"
1,3,"0.5614197853127827,0.059172654879085296,0.4692371689047904"
1,4,"0.17583413999295983,0.5191278830882644,0.8453123358491914"
```

The `graphs.csv` contains a `graph_id` column and arbitrary number of feature columns. The example dataset here has two graphs, each with a `feat` and a `label` graph-level data.

```
graph_id,feat,label
0,"0.7426272601929126,0.5197462471155317,0.8149104951283953",0
1,"0.534822233529295,0.2863627767733977,0.1154897249106891",0
```

After loaded, the dataset has multiple homographs with features and labels:

```
>>> import dgl
>>> dataset = dgl.data.CSVDataset('./mini_multi_dataset')
>>> print(len(dataset))
2
>>> graph0, data0 = dataset[0]
>>> print(graph0)
Graph(num_nodes=5, num_edges=10,
      ndata_schemes={'feat': Scheme(shape=(3,), dtype=torch.float64)}
      edata_schemes={'feat': Scheme(shape=(3,), dtype=torch.float64)})
>>> print(data0)
{'feat': tensor([0.7426, 0.5197, 0.8149], dtype=torch.float64), 'label': tensor(0)}
>>> graph1, data1 = dataset[1]
>>> print(graph1)
Graph(num_nodes=5, num_edges=10,
      ndata_schemes={'feat': Scheme(shape=(3,), dtype=torch.float64)}
      edata_schemes={'feat': Scheme(shape=(3,), dtype=torch.float64)})
>>> print(data1)
{'feat': tensor([0.5348, 0.2864, 0.1155], dtype=torch.float64), 'label': tensor(0)}
```

If there is a single feature column in `graphs.csv`, `data0` will directly be a tensor for the feature.

Custom Data Parser

By default, `CSVDataset` assumes that all the stored node-/edge-/graph- level data are numerical values. Users can provide custom `DataParser` to `CSVDataset` to handle more complex data type. A `DataParser` needs to implement the `__call__` method which takes in the **`pandas.DataFrame`** object created from CSV file and should return a dictionary of parsed feature data. The parsed feature data will be saved to the `ndata` and `edata` of the corresponding `DGLGraph` object, and thus must be tensors or numpy arrays. Below shows an example `DataParser` which converts string type labels to integers:

Given a dataset as follows,

```
./customized_parser_dataset/
|-- meta.yaml
|-- nodes.csv
|-- edges.csv
```

`meta.yaml`:

```
dataset_name: customized_parser_dataset
edge_data:
- file_name: edges.csv
node_data:
- file_name: nodes.csv
```

`edges.csv`:

```
src_id,dst_id,label
4,0,positive
4,0,negative
0,3,positive
0,1,positive
0,2,negative
0,0,positive
2,2,negative
1,0,positive
3,0,negative
4,0,positive
```

`nodes.csv`:

```
node_id,label
0,positive
1,negative
2,positive
3,negative
4,positive
```

To parse the string type labels, one can define a `DataParser` class as follows:

```
import numpy as np
import pandas as pd

class MyDataParser:
    def __call__(self, df: pd.DataFrame):
        parsed = {}
        for header in df:
            if 'Unnamed' in header:  # Handle Unnamed column
                print("Unamed column is found. Ignored...")
                continue
            dt = df[header].to_numpy().squeeze()
            if header == 'label':
                dt = np.array([1 if e == 'positive' else 0 for e in dt])
            parsed[header] = dt
        return parsed
```

Create a `CSVDataset` using the defined `DataParser`:

```
>>> import dgl
>>> dataset = dgl.data.CSVDataset('./customized_parser_dataset',
...                               ndata_parser=MyDataParser(),
...                               edata_parser=MyDataParser())
>>> print(dataset[0].ndata['label'])
tensor([1, 0, 1, 0, 1])
>>> print(dataset[0].edata['label'])
tensor([1, 0, 1, 1, 0, 1, 0, 1, 0, 1])
```

> **Note**
>
> To specify different `DataParser`s for different node/edge types, pass a dictionary to `ndata_parser` and `edata_parser`, where the key is type name (a single string for node type; a string triplet for edge type) and the value is the `DataParser` to use.

Full YAML Specification

`CSVDataset` allows more flexible control over the loading and parsing process. For example, one can change the ID column names via `meta.yaml`. The example below lists all the supported keys.

```yaml
version: 1.0.0
dataset_name: some_complex_data
separator: ','                   # CSV separator symbol. Default: ','
edge_data:
- file_name: edges_0.csv
  etype: [user, follow, user]
  src_id_field: src_id           # Column name for source node IDs. Default: src_id
  dst_id_field: dst_id           # Column name for destination node IDs. Default: dst_id
- file_name: edges_1.csv
  etype: [user, like, item]
  src_id_field: src_id
  dst_id_field: dst_id
node_data:
- file_name: nodes_0.csv
  ntype: user
  node_id_field: node_id         # Column name for node IDs. Default: node_id
- file_name: nodes_1.csv
  ntype: item
  node_id_field: node_id         # Column name for node IDs. Default: node_id
graph_data:
  file_name: graphs.csv
  graph_id_field: graph_id       # Column name for graph IDs. Default: graph_id
```

Top-level

At the top level, only 6 keys are available:

- `version`: Optional. String. It specifies which version of `meta.yaml` is used. More feature may be added in the future.
- `dataset_name`: Required. String. It specifies the dataset name.
- `separator`: Optional. String. It specifies how to parse data in CSV files. Default: `','`.
- `edge_data`: Required. List of `EdgeData`. Meta data for parsing edge CSV files.
- `node_data`: Required. List of `NodeData`. Meta data for parsing node CSV files.
- `graph_data`: Optional. `GraphData`. Meta data for parsing the graph CSV file.

`EdgeData`

There are 4 keys:

- `file_name`: Required. String. The CSV file to load data from.
- `etype`: Optional. List of string. Edge type name in string triplet: [source node type, relation type, destination node type].
- `src_id_field`: Optional. String. Which column to read for source node IDs. Default: `src_id`.
- `dst_id_field`: Optional. String. Which column to read for destination node IDs. Default: `dst_id`.

`NodeData`

There are 3 keys:

- `file_name`: Required. String. The CSV file to load data from.
- `ntype`: Optional. String. Node type name.
- `node_id_field`: Optional. String. Which column to read for node IDs. Default: `node_id`.

`GraphData`

There are 2 keys:

- `file_name`: Required. String. The CSV file to load data from.
- `graph_id_field`: Optional. String. Which column to read for graph IDs. Default: `graph_id`.