Author: Tamirlan Seidakhmetov

This colab is built as a part of the CS224W Final Project. The Final Project draft blogpost could be found here

This colab will walk us through building a Movie Recommender System using the Graph Neural Network approach. Specifically, we will employ an Inductive Graph Based Matrix Completion (IGMC) framework introduced at the ICLR 2020 conference. The code structure has been inspired/adapted from the paper's official Github page.

First, we start by installing the necessary packages.

```
!pip install torch-geometric==2.0.1
!pip install torch-scatter -f https://data.pyg.org/whl/torch-1.10.0+cu113.html
!pip install torch-sparse -f https://data.pyg.org/whl/torch-1.10.0+cu113.html
```

```
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (f
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (fro
Collecting rdflib (from torch-geometric==2.0.1)
  Downloading rdflib-7.0.0-py3-none-any.whl (531 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 531.9/531.9 kB 30.4 MB/s eta 0:00:00
Requirement already satisfied: googledrivedownloader in /usr/local/lib/python3.10/dis
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (fro
Requirement already satisfied: pyparsing in /usr/local/lib/python3.10/dist-packages (
Collecting yacs (from torch-geometric==2.0.1)
  Downloading yacs-0.1.8-py3-none-any.whl (14 kB)
Requirement already satisfied: PyYAML in /usr/local/lib/python3.10/dist-packages (fro
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/di
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-package
Collecting isodate<0.7.0,>=0.6.0 (from rdflib->torch-geometric==2.0.1)
  Downloading isodate-0.6.1-py2.py3-none-any.whl (41 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 41.7/41.7 kB 2.4 MB/s eta 0:00:00
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages (from i
Building wheels for collected packages: torch-geometric
  Building wheel for torch-geometric (setup.py) ... done
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 168.0/168.0 kB 1.8 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: torch-scatter
  Building wheel for torch-scatter (setup.py) ... done
  Created wheel for torch-scatter: filename=torch_scatter-2.1.2-cp310-cp310-linux_x86
  Stored in directory: /root/.cache/pip/wheels/92/f1/2b/3b46d54b134259f58c83635685690
Successfully built torch-scatter
Installing collected packages: torch-scatter
Successfully installed torch-scatter-2.1.2
Looking in links: https://data.pyg.org/whl/torch-1.10.0+cu113.html
Collecting torch-sparse
  Downloading torch_sparse-0.6.18.tar.gz (209 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 210.0/210.0 kB 5.3 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: numpy<1.28.0,>=1.21.6 in /usr/local/lib/python3.10/dis
Building wheels for collected packages: torch-sparse
  Building wheel for torch-sparse (setup.py) ... done
  Created wheel for torch-sparse: filename=torch_sparse-0.6.18-cp310-cp310-linux_x86_
  Stored in directory: /root/.cache/pip/wheels/c9/dd/0f/a6a16f9f3b0236733d257b4b4ea91
Successfully built torch-sparse
Installing collected packages: torch-sparse
```

Next, we clone the public Github code that will help us download the data and do some preprocessing. We move the required files outside of the cloned folder to use them later

```
from __future__ import division
from __future__ import print_function

import numpy as np
import pandas as pd
import scipy.sparse as sp
import random
import pdb

# For automatic dataset downloading
from urllib.request import urlopen
from zipfile import ZipFile
import shutil
import os.path
from tqdm import tqdm


def data_iterator(data, batch_size):
    """
    A simple data iterator from https://indico.io/blog/tensorflow-data-inputs-part1-placeho
    :param data: list of numpy tensors that need to be randomly batched across their first
    :param batch_size: int, batch_size of data_iterator.
    Assumes same first dimension size of all numpy tensors.
    :return: iterator over batches of numpy tensors
    """
    # shuffle labels and features
    max_idx = len(data[0])
    idxs = np.arange(0, max_idx)
    np.random.shuffle(idxs)
    shuf_data = [dat[idxs] for dat in data]

    # Does not yield last remainder of size less than batch_size
    for i in range(max_idx//batch_size):
        data_batch = [dat[i*batch_size:(i+1)*batch_size] for dat in shuf_data]
        yield data_batch


def map_data(data):
    """
    Map data to proper indices in case they are not in a continues [0, N) range

    Parameters
    ----------
    data : np.int32 arrays

    Returns
    -------
    mapped_data : np.int32 arrays
    n : length of mapped_data

    """
```

```
        uniq = list(set(data))

        id_dict = {old: new for new, old in enumerate(sorted(uniq))}
        data = np.array([id_dict[x] for x in data])
        n = len(uniq)

        return data, id_dict, n




    def load_data(fname, seed=1234, verbose=True):
        """ Loads dataset and creates adjacency matrix
        and feature matrix

        Parameters
        ----------
        fname : str, dataset
        seed: int, dataset shuffling seed
        verbose: to print out statements or not

        Returns
        -------
        num_users : int
            Number of users and items respectively

        num_items : int

        u_nodes : np.int32 arrays
            User indices

        v_nodes : np.int32 array
            item (movie) indices

        ratings : np.float32 array
            User/item ratings s.t. ratings[k] is the rating given by user u_nodes[k] to
            item v_nodes[k]. Note that that the all pairs u_nodes[k]/v_nodes[k] are unique, but
            not necessarily all u_nodes[k] or all v_nodes[k] separately.

        u_features: np.float32 array, or None
            If present in dataset, contains the features of the users.

        v_features: np.float32 array, or None
            If present in dataset, contains the features of the users.

        seed: int,
            For datashuffling seed with pythons own random.shuffle, as in CF-NADE.

        """
        data_dir = '.'
```

```
    u_features = None
    v_features = None

    print('Loading dataset', fname)



    # Check if files exist and download otherwise
    files = ['/u.data', '/u.item', '/u.user']



    sep = '\t'
    filename = data_dir + files[0]

    dtypes = {
        'u_nodes': np.int32, 'v_nodes': np.int32,
        'ratings': np.float32, 'timestamp': np.float64}

    data = pd.read_csv(
        filename, sep=sep, header=None,
        names=['u_nodes', 'v_nodes', 'ratings', 'timestamp'], dtype=dtypes)

    # shuffle here like cf-nade paper with python's own random class
    # make sure to convert to list, otherwise random.shuffle acts weird on it without a war
    data_array = data.values.tolist()
    random.seed(seed)
    random.shuffle(data_array)
    data_array = np.array(data_array)

    u_nodes_ratings = data_array[:, 0].astype(dtypes['u_nodes'])
    v_nodes_ratings = data_array[:, 1].astype(dtypes['v_nodes'])
    ratings = data_array[:, 2].astype(dtypes['ratings'])

    u_nodes_ratings, u_dict, num_users = map_data(u_nodes_ratings)
    v_nodes_ratings, v_dict, num_items = map_data(v_nodes_ratings)

    u_nodes_ratings, v_nodes_ratings = u_nodes_ratings.astype(np.int64), v_nodes_ratings.as
    ratings = ratings.astype(np.float64)

    # Movie features (genres)
    sep = r'|'
    movie_file = data_dir + files[1]
    movie_headers = ['movie id', 'movie title', 'release date', 'video release date',
                     'IMDb URL', 'unknown', 'Action', 'Adventure', 'Animation',
                     'Childrens', 'Comedy', 'Crime', 'Documentary', 'Drama', 'Fantasy',
                     'Film-Noir', 'Horror', 'Musical', 'Mystery', 'Romance', 'Sci-Fi',
                     'Thriller', 'War', 'Western']
    movie_df = pd.read_csv(movie_file, sep=sep, header=None,
                           names=movie_headers, engine='python')
```

```
    genre_headers = movie_df.columns.values[6:]
    num_genres = genre_headers.shape[0]

    v_features = np.zeros((num_items, num_genres), dtype=np.float32)
    for movie_id, g_vec in zip(movie_df['movie id'].values.tolist(), movie_df[genre_headers
        # Check if movie_id was listed in ratings file and therefore in mapping dictionary
        if movie_id in v_dict.keys():
            v_features[v_dict[movie_id], :] = g_vec

    # User features

    sep = r'|'
    users_file = data_dir + files[2]
    users_headers = ['user id', 'age', 'gender', 'occupation', 'zip code']
    users_df = pd.read_csv(users_file, sep=sep, header=None,
                           names=users_headers, engine='python')

    occupation = set(users_df['occupation'].values.tolist())

    gender_dict = {'M': 0., 'F': 1.}
    occupation_dict = {f: i for i, f in enumerate(occupation, start=2)}

    num_feats = 2 + len(occupation_dict)

    u_features = np.zeros((num_users, num_feats), dtype=np.float32)
    for _, row in users_df.iterrows():
        u_id = row['user id']
        if u_id in u_dict.keys():
            # age
            u_features[u_dict[u_id], 0] = row['age']
            # gender
            u_features[u_dict[u_id], 1] = gender_dict[row['gender']]
            # occupation
            u_features[u_dict[u_id], occupation_dict[row['occupation']]] = 1.

    u_features = sp.csr_matrix(u_features)
    v_features = sp.csr_matrix(v_features)

    if verbose:
        print('Number of users = %d' % num_users)
        print('Number of items = %d' % num_items)
        print('Number of links = %d' % ratings.shape[0])
        print('Fraction of positive links = %.4f' % (float(ratings.shape[0]) / (num_users *

    return num_users, num_items, u_nodes_ratings, v_nodes_ratings, ratings, u_features, v_f
```

```
from __future__ import print_function
import numpy as np
import random
from tqdm import tqdm
import os, sys, pdb, math, time
from copy import deepcopy
import multiprocessing as mp
import networkx as nx
import argparse
import scipy.io as sio
import scipy.sparse as ssp
import torch
from torch_geometric.data import Data, Dataset, InMemoryDataset
import warnings
warnings.simplefilter('ignore', ssp.SparseEfficiencyWarning)
import torch.multiprocessing
torch.multiprocessing.set_sharing_strategy('file_system')

class SparseRowIndexer:
    def __init__(self, csr_matrix):
        data = []
        indices = []
        indptr = []

        for row_start, row_end in zip(csr_matrix.indptr[:-1], csr_matrix.indptr[1:]):
            data.append(csr_matrix.data[row_start:row_end])
            indices.append(csr_matrix.indices[row_start:row_end])
            indptr.append(row_end - row_start)  # nnz of the row

        self.data = np.array(data, dtype=object)
        self.indices = np.array(indices, dtype=object)
        self.indptr = np.array(indptr, dtype=object)
        self.shape = csr_matrix.shape

    def __getitem__(self, row_selector):
        indices = np.concatenate(self.indices[row_selector])
        data = np.concatenate(self.data[row_selector])
        indptr = np.append(0, np.cumsum(self.indptr[row_selector]))
        shape = [indptr.shape[0] - 1, self.shape[1]]
        return ssp.csr_matrix((data, indices, indptr), shape=shape)


class SparseColIndexer:
    def __init__(self, csc_matrix):
        data = []
        indices = []
        indptr = []

        for col_start, col_end in zip(csc_matrix.indptr[:-1], csc_matrix.indptr[1:]):
            data.append(csc_matrix.data[col_start:col_end])
            indices.append(csc_matrix.indices[col_start:col_end])
```

```
                indptr.append(col_end - col_start)

            self.data = np.array(data, dtype=object)
            self.indices = np.array(indices, dtype=object)
            self.indptr = np.array(indptr, dtype=object)
            self.shape = csc_matrix.shape

        def __getitem__(self, col_selector):
            indices = np.concatenate(self.indices[col_selector])
            data = np.concatenate(self.data[col_selector])
            indptr = np.append(0, np.cumsum(self.indptr[col_selector]))

            shape = [self.shape[0], indptr.shape[0] - 1]
            return ssp.csc_matrix((data, indices, indptr), shape=shape)


    class MyDataset(InMemoryDataset):
        def __init__(self, root, A, links, labels, h, sample_ratio, max_nodes_per_hop,
                     u_features, v_features, class_values, max_num=None, parallel=True):
            self.Arow = SparseRowIndexer(A)
            self.Acol = SparseColIndexer(A.tocsc())
            self.links = links
            self.labels = labels
            self.h = h
            self.sample_ratio = sample_ratio
            self.max_nodes_per_hop = max_nodes_per_hop
            self.u_features = u_features
            self.v_features = v_features
            self.class_values = class_values
            self.parallel = parallel
            self.max_num = max_num
            if max_num is not None:
                np.random.seed(123)
                num_links = len(links[0])
                perm = np.random.permutation(num_links)
                perm = perm[:max_num]
                self.links = (links[0][perm], links[1][perm])
                self.labels = labels[perm]
            super(MyDataset, self).__init__(root)
            self.data, self.slices = torch.load(self.processed_paths[0])

        @property
        def processed_file_names(self):
            name = 'data.pt'
            if self.max_num is not None:
                name = 'data_{}.pt'.format(self.max_num)
            return [name]

        def process(self):
            # Extract enclosing subgraphs and save to disk
            data_list = links2subgraphs(self.Arow, self.Acol, self.links, self.labels, self.h,
```

```
                                        self.sample_ratio, self.max_nodes_per_hop,
                                        self.u_features, self.v_features,
                                        self.class_values, self.parallel)

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


class MyDynamicDataset(Dataset):
    def __init__(self, root, A, links, labels, h, sample_ratio, max_nodes_per_hop,
                 u_features, v_features, class_values, max_num=None):
        super(MyDynamicDataset, self).__init__(root)
        self.Arow = SparseRowIndexer(A)
        self.Acol = SparseColIndexer(A.tocsc())
        self.links = links
        self.labels = labels
        self.h = h
        self.sample_ratio = sample_ratio
        self.max_nodes_per_hop = max_nodes_per_hop
        self.u_features = u_features
        self.v_features = v_features
        self.class_values = class_values
        if max_num is not None:
            np.random.seed(123)
            num_links = len(links[0])
            perm = np.random.permutation(num_links)
            perm = perm[:max_num]
            self.links = (links[0][perm], links[1][perm])
            self.labels = labels[perm]

    def __len__(self):
        return len(self.links[0])

    def len(self):
        return len(self.links[0])

    def get(self, idx):
        i, j = self.links[0][idx], self.links[1][idx]
        g_label = self.labels[idx]
        tmp = subgraph_extraction_labeling(
            (i, j), self.Arow, self.Acol, self.h, self.sample_ratio, self.max_nodes_per_hop
            self.u_features, self.v_features, self.class_values, g_label
        )
        return construct_pyg_graph(*tmp)


def links2subgraphs(Arow,
                    Acol,
                    links,
                    labels,
```

```
                    h=1,
                    sample_ratio=1.0,
                    max_nodes_per_hop=None,
                    u_features=None,
                    v_features=None,
                    class_values=None,
                    parallel=True):
    # extract enclosing subgraphs
    print('Enclosing subgraph extraction begins...')
    g_list = []
    if not parallel:
        with tqdm(total=len(links[0])) as pbar:
            for i, j, g_label in zip(links[0], links[1], labels):
                tmp = subgraph_extraction_labeling(
                    (i, j), Arow, Acol, h, sample_ratio, max_nodes_per_hop, u_features,
                    v_features, class_values, g_label
                )
                data = construct_pyg_graph(*tmp)
                g_list.append(data)
                pbar.update(1)
    else:
        start = time.time()
        pool = mp.Pool(mp.cpu_count())
        results = pool.starmap_async(
            subgraph_extraction_labeling,
            [
                ((i, j), Arow, Acol, h, sample_ratio, max_nodes_per_hop, u_features,
                v_features, class_values, g_label)
                for i, j, g_label in zip(links[0], links[1], labels)
            ]
        )
        remaining = results._number_left
        pbar = tqdm(total=remaining)
        while True:
            pbar.update(remaining - results._number_left)
            if results.ready(): break
            remaining = results._number_left
            time.sleep(1)
        results = results.get()
        pool.close()
        pbar.close()
        end = time.time()
        print("Time elapsed for subgraph extraction: {}s".format(end-start))
        print("Transforming to pytorch_geometric graphs...")
        g_list = []
        pbar = tqdm(total=len(results))
        while results:
            tmp = results.pop()
            g_list.append(construct_pyg_graph(*tmp))
            pbar.update(1)
        pbar.close()
```

```
        end2 = time.time()
        print("Time elapsed for transforming to pytorch_geometric graphs: {}s".format(end2-
    return g_list


def subgraph_extraction_labeling(ind, Arow, Acol, h=1, sample_ratio=1.0, max_nodes_per_hop=
                                 u_features=None, v_features=None, class_values=None,
                                 y=1):
    # extract the h-hop enclosing subgraph around link 'ind'
    u_nodes, v_nodes = [ind[0]], [ind[1]]
    u_dist, v_dist = [0], [0]
    u_visited, v_visited = set([ind[0]]), set([ind[1]])
    u_fringe, v_fringe = set([ind[0]]), set([ind[1]])
    for dist in range(1, h+1):
        v_fringe, u_fringe = neighbors(u_fringe, Arow), neighbors(v_fringe, Acol)
        u_fringe = u_fringe - u_visited
        v_fringe = v_fringe - v_visited
        u_visited = u_visited.union(u_fringe)
        v_visited = v_visited.union(v_fringe)
        if sample_ratio < 1.0:
            u_fringe = random.sample(u_fringe, int(sample_ratio*len(u_fringe)))
            v_fringe = random.sample(v_fringe, int(sample_ratio*len(v_fringe)))
        if max_nodes_per_hop is not None:
            if max_nodes_per_hop < len(u_fringe):
                u_fringe = random.sample(u_fringe, max_nodes_per_hop)
            if max_nodes_per_hop < len(v_fringe):
                v_fringe = random.sample(v_fringe, max_nodes_per_hop)
        if len(u_fringe) == 0 and len(v_fringe) == 0:
            break
        u_nodes = u_nodes + list(u_fringe)
        v_nodes = v_nodes + list(v_fringe)
        u_dist = u_dist + [dist] * len(u_fringe)
        v_dist = v_dist + [dist] * len(v_fringe)
    subgraph = Arow[u_nodes][:, v_nodes]
    # remove link between target nodes
    subgraph[0, 0] = 0

    # prepare pyg graph constructor input
    u, v, r = ssp.find(subgraph)  # r is 1, 2... (rating labels + 1)
    v += len(u_nodes)
    r = r - 1  # transform r back to rating label
    num_nodes = len(u_nodes) + len(v_nodes)
    node_labels = [x*2 for x in u_dist] + [x*2+1 for x in v_dist]
    max_node_label = 2*h + 1
    y = class_values[y]

    # get node features
    if u_features is not None:
        u_features = u_features[u_nodes]
    if v_features is not None:
        v_features = v_features[v_nodes]
```

```
        node_features = None
        if False:
            # directly use padded node features
            if u_features is not None and v_features is not None:
                u_extended = np.concatenate(
                    [u_features, np.zeros([u_features.shape[0], v_features.shape[1]])], 1
                )
                v_extended = np.concatenate(
                    [np.zeros([v_features.shape[0], u_features.shape[1]]), v_features], 1
                )
                node_features = np.concatenate([u_extended, v_extended], 0)
        if False:
            # use identity features (one-hot encodings of node idxes)
            u_ids = one_hot(u_nodes, Arow.shape[0] + Arow.shape[1])
            v_ids = one_hot([x+Arow.shape[0] for x in v_nodes], Arow.shape[0] + Arow.shape[1])
            node_ids = np.concatenate([u_ids, v_ids], 0)
            #node_features = np.concatenate([node_features, node_ids], 1)
            node_features = node_ids
        if True:
            # only output node features for the target user and item
            if u_features is not None and v_features is not None:
                node_features = [u_features[0], v_features[0]]

        return u, v, r, node_labels, max_node_label, y, node_features


    def construct_pyg_graph(u, v, r, node_labels, max_node_label, y, node_features):
        u, v = torch.LongTensor(u), torch.LongTensor(v)
        r = torch.LongTensor(r)
        edge_index = torch.stack([torch.cat([u, v]), torch.cat([v, u])], 0)
        edge_type = torch.cat([r, r])
        x = torch.FloatTensor(one_hot(node_labels, max_node_label+1))
        y = torch.FloatTensor([y])
        data = Data(x, edge_index, edge_type=edge_type, y=y)

        if node_features is not None:
            if type(node_features) == list:  # a list of u_feature and v_feature
                u_feature, v_feature = node_features
                data.u_feature = torch.FloatTensor(u_feature).unsqueeze(0)
                data.v_feature = torch.FloatTensor(v_feature).unsqueeze(0)
            else:
                x2 = torch.FloatTensor(node_features)
                data.x = torch.cat([data.x, x2], 1)
        return data


    def neighbors(fringe, A):
        # find all 1-hop neighbors of nodes in fringe from A
        if not fringe:
            return set([])
        return set(A[list(fringe)].indices)
```

```
def one_hot(idx, length):
    idx = np.array(idx)
    x = np.zeros([len(idx), length])
    x[np.arange(len(idx)), idx] = 1.0
    return x


def PyGGraph_to_nx(data):
    edges = list(zip(data.edge_index[0, :].tolist(), data.edge_index[1, :].tolist()))
    g = nx.from_edgelist(edges)
    g.add_nodes_from(range(len(data.x)))  # in case some nodes are isolated
    # transform r back to rating label
    edge_types = {(u, v): data.edge_type[i].item() for i, (u, v) in enumerate(edges)}
    nx.set_edge_attributes(g, name='type', values=edge_types)
    node_types = dict(zip(range(data.num_nodes), torch.argmax(data.x, 1).tolist()))
    nx.set_node_attributes(g, name='type', values=node_types)
    g.graph['rating'] = data.y.item()
    return g
```

```
from __future__ import division
from __future__ import print_function

import numpy as np
import scipy.sparse as sp
import pickle as pkl
import os
import h5py
import pandas as pd
import pdb


def normalize_features(feat):

    degree = np.asarray(feat.sum(1)).flatten()

    # set zeros to inf to avoid dividing by zero
    degree[degree == 0.] = np.inf

    degree_inv = 1. / degree
    degree_inv_mat = sp.diags([degree_inv], [0])
    feat_norm = degree_inv_mat.dot(feat)

    if feat_norm.nnz == 0:
        print('ERROR: normalized adjacency matrix has only zero entries!!!!!')
        exit

    return feat_norm


def load_matlab_file(path_file, name_field):
    """
    load '.mat' files
    inputs:
        path_file, string containing the file path
        name_field, string containig the field name (default='shape')
    warning:
        '.mat' files should be saved in the '-v7.3' format
    """
    db = h5py.File(path_file, 'r')
    ds = db[name_field]
    try:
        if 'ir' in ds.keys():
            data = np.asarray(ds['data'])
            ir = np.asarray(ds['ir'])
            jc = np.asarray(ds['jc'])
            out = sp.csc_matrix((data, ir, jc)).astype(np.float32)
    except AttributeError:
        # Transpose in case is a dense matrix because of the row- vs column- major orderin
        out = np.asarray(ds).astype(np.float32).T
```

```
        db.close()

        return out


    def preprocess_user_item_features(u_features, v_features):
        """
        Creates one big feature matrix out of user features and item features.
        Stacks item features under the user features.
        """

        zero_csr_u = sp.csr_matrix((u_features.shape[0], v_features.shape[1]), dtype=u_feature
        zero_csr_v = sp.csr_matrix((v_features.shape[0], u_features.shape[1]), dtype=v_feature

        u_features = sp.hstack([u_features, zero_csr_u], format='csr')
        v_features = sp.hstack([zero_csr_v, v_features], format='csr')

        return u_features, v_features


    def globally_normalize_bipartite_adjacency(adjacencies, verbose=False, symmetric=True):
        """ Globally Normalizes set of bipartite adjacency matrices """

        if verbose:
            print('Symmetrically normalizing bipartite adj')
        # degree_u and degree_v are row and column sums of adj+I

        adj_tot = np.sum(adj for adj in adjacencies)
        degree_u = np.asarray(adj_tot.sum(1)).flatten()
        degree_v = np.asarray(adj_tot.sum(0)).flatten()

        # set zeros to inf to avoid dividing by zero
        degree_u[degree_u == 0.] = np.inf
        degree_v[degree_v == 0.] = np.inf

        degree_u_inv_sqrt = 1. / np.sqrt(degree_u)
        degree_v_inv_sqrt = 1. / np.sqrt(degree_v)
        degree_u_inv_sqrt_mat = sp.diags([degree_u_inv_sqrt], [0])
        degree_v_inv_sqrt_mat = sp.diags([degree_v_inv_sqrt], [0])

        degree_u_inv = degree_u_inv_sqrt_mat.dot(degree_u_inv_sqrt_mat)

        if symmetric:
            adj_norm = [degree_u_inv_sqrt_mat.dot(adj).dot(degree_v_inv_sqrt_mat) for adj in a

        else:
            adj_norm = [degree_u_inv.dot(adj) for adj in adjacencies]

        return adj_norm
```

```
def sparse_to_tuple(sparse_mx):
    """ change of format for sparse matrix. This format is used
    for the feed_dict where sparse matrices need to be linked to placeholders
    representing sparse matrices. """

    if not sp.isspmatrix_coo(sparse_mx):
        sparse_mx = sparse_mx.tocoo()
    coords = np.vstack((sparse_mx.row, sparse_mx.col)).transpose()
    values = sparse_mx.data
    shape = sparse_mx.shape
    return coords, values, shape


def create_trainvaltest_split(dataset, seed=1234, testing=False, datasplit_path=None,
                              datasplit_from_file=False, verbose=True, rating_map=None,
                              post_rating_map=None, ratio=1.0):
    """
    Splits data set into train/val/test sets from full bipartite adjacency matrix. Shuffli
    load_data function.
    For each split computes 1-of-num_classes labels. Also computes training
    adjacency matrix.
    """

    if datasplit_from_file and os.path.isfile(datasplit_path):
        print('Reading processed dataset from file...')
        with open(datasplit_path, 'rb') as f:
            num_users, num_items, u_nodes, v_nodes, ratings, u_features, v_features = pkl.

        if verbose:
            print('Number of users = %d' % num_users)
            print('Number of items = %d' % num_items)
            print('Number of links = %d' % ratings.shape[0])
            print('Fraction of positive links = %.4f' % (float(ratings.shape[0]) / (num_us

    else:
        num_users, num_items, u_nodes, v_nodes, ratings, u_features, v_features = load_dat


        with open(datasplit_path, 'wb') as f:
            pkl.dump([num_users, num_items, u_nodes, v_nodes, ratings, u_features, v_featu

    if rating_map is not None:
        for i, x in enumerate(ratings):
            ratings[i] = rating_map[x]

    rating_dict = {r: i for i, r in enumerate(np.sort(np.unique(ratings)).tolist())}

    # number of test and validation edges
    if dataset == 'ml_25m':
        print("Split dataset into train/val/test by time ...")
        num_train = int(ratings.shape[0] * 0.7)
```

```
            num_val = int(ratings.shape[0] * 0.8) - num_train
            num_test = ratings.shape[0] - num_train - num_val
        else:
            print("Using random dataset split ...")
            num_test = int(np.ceil(ratings.shape[0] * 0.1))
            if dataset == 'ml_100k':
                num_val = int(np.ceil(ratings.shape[0] * 0.9 * 0.05))
            else:
                num_val = int(np.ceil(ratings.shape[0] * 0.9 * 0.05))
            num_train = ratings.shape[0] - num_val - num_test

        pairs_nonzero = np.vstack([u_nodes, v_nodes]).transpose()

        train_pairs_idx = pairs_nonzero[0:int(num_train*ratio)]
        val_pairs_idx = pairs_nonzero[num_train:num_train + num_val]
        test_pairs_idx = pairs_nonzero[num_train + num_val:]

        u_test_idx, v_test_idx = test_pairs_idx.transpose()
        u_val_idx, v_val_idx = val_pairs_idx.transpose()
        u_train_idx, v_train_idx = train_pairs_idx.transpose()

        # create labels
        all_labels = np.array([rating_dict[r] for r in ratings], dtype=np.int32)
        train_labels = all_labels[0:int(num_train*ratio)]
        val_labels = all_labels[num_train:num_train + num_val]
        test_labels = all_labels[num_train + num_val:]

        if testing:
            u_train_idx = np.hstack([u_train_idx, u_val_idx])
            v_train_idx = np.hstack([v_train_idx, v_val_idx])
            train_labels = np.hstack([train_labels, val_labels])

        class_values = np.sort(np.unique(ratings))

        # make training adjacency matrix
        if post_rating_map is None:
            data = train_labels + 1.
        else:
            data = np.array([post_rating_map[r] for r in class_values[train_labels]]) + 1.
        data = data.astype(np.float32)

        rating_mx_train = sp.csr_matrix((data, [u_train_idx, v_train_idx]),
                                        shape=[num_users, num_items], dtype=np.float32)

        return u_features, v_features, rating_mx_train, train_labels, u_train_idx, v_train_idx
            val_labels, u_val_idx, v_val_idx, test_labels, u_test_idx, v_test_idx, class_value


    def load_data_monti(dataset, testing=False, rating_map=None, post_rating_map=None):
        """
        Loads data from Monti et al. paper.
```

```
    if rating_map is given, apply this map to the original rating matrix
    if post_rating_map is given, apply this map to the processed rating_mx_train without a
    """

    path_dataset = 'raw_data/' + dataset + '/training_test_dataset.mat'

    M = load_matlab_file(path_dataset, 'M')
    if rating_map is not None:
        M[np.where(M)] = [rating_map[x] for x in M[np.where(M)]]

    Otraining = load_matlab_file(path_dataset, 'Otraining')
    Otest = load_matlab_file(path_dataset, 'Otest')

    num_users = M.shape[0]
    num_items = M.shape[1]

    if dataset == 'flixster':
        Wrow = load_matlab_file(path_dataset, 'W_users')
        Wcol = load_matlab_file(path_dataset, 'W_movies')
        u_features = Wrow
        v_features = Wcol
    elif dataset == 'douban':
        Wrow = load_matlab_file(path_dataset, 'W_users')
        u_features = Wrow
        v_features = np.eye(num_items)
    elif dataset == 'yahoo_music':
        Wcol = load_matlab_file(path_dataset, 'W_tracks')
        u_features = np.eye(num_users)
        v_features = Wcol

    u_nodes_ratings = np.where(M)[0]
    v_nodes_ratings = np.where(M)[1]
    ratings = M[np.where(M)]

    u_nodes_ratings, v_nodes_ratings = u_nodes_ratings.astype(np.int64), v_nodes_ratings.a
    ratings = ratings.astype(np.float64)

    u_nodes = u_nodes_ratings
    v_nodes = v_nodes_ratings

    print('number of users = ', len(set(u_nodes)))
    print('number of item = ', len(set(v_nodes)))

    neutral_rating = -1  # int(np.ceil(np.float(num_classes)/2.)) - 1

    # assumes that ratings_train contains at least one example of every rating type
    rating_dict = {r: i for i, r in enumerate(np.sort(np.unique(ratings)).tolist())}

    labels = np.full((num_users, num_items), neutral_rating, dtype=np.int32)
    labels[u_nodes, v_nodes] = np.array([rating_dict[r] for r in ratings])
```

```
    for i in range(len(u_nodes)):
        assert(labels[u_nodes[i], v_nodes[i]] == rating_dict[ratings[i]])

    labels = labels.reshape([-1])

    # number of test and validation edges

    num_train = np.where(Otraining)[0].shape[0]
    num_test = np.where(Otest)[0].shape[0]
    num_val = int(np.ceil(num_train * 0.2))
    num_train = num_train - num_val

    pairs_nonzero_train = np.array([[u, v] for u, v in zip(np.where(Otraining)[0], np.wher
    idx_nonzero_train = np.array([u * num_items + v for u, v in pairs_nonzero_train])

    pairs_nonzero_test = np.array([[u, v] for u, v in zip(np.where(Otest)[0], np.where(Ote
    idx_nonzero_test = np.array([u * num_items + v for u, v in pairs_nonzero_test])

    # Internally shuffle training set (before splitting off validation set)
    rand_idx = list(range(len(idx_nonzero_train)))
    np.random.seed(42)
    np.random.shuffle(rand_idx)
    idx_nonzero_train = idx_nonzero_train[rand_idx]
```