Algorithm/DeepFM/main.py

if \_\_name\_\_ == '\_\_main\_\_':

import Recommender\_System.utility.gpu\_memory\_growth

from Recommender\_System.data import data\_loader, data\_process

from Recommender\_System.algorithm.DeepFM.model import DeepFM\_model

from Recommender\_System.algorithm.train import train

n\_user, n\_item, train\_data, test\_data, topk\_data = data\_process.pack(data\_loader.ml100k)

model = DeepFM\_model(n\_user, n\_item, dim=8, layers=[16, 16, 16], l2=1e-5)

train(model, train\_data, test\_data, topk\_data, epochs=10)

algorithm/DeepFM/model.py

import tensorflow as tf

from Recommender\_System.utility.decorator import logger

@logger('初始化DeepFM模型：', ('n\_user', 'n\_item', 'dim', 'layers', 'l2'))

def DeepFM\_model(n\_user: int, n\_item: int, dim=8, layers=[16, 16, 16], l2=1e-6) -> tf.keras.Model:

l2 = tf.keras.regularizers.l2(l2)

user\_id = tf.keras.Input(shape=(), name='user\_id', dtype=tf.int32)

user\_embedding = tf.keras.layers.Embedding(n\_user, dim, embeddings\_regularizer=l2)(user\_id)

item\_id = tf.keras.Input(shape=(), name='item\_id', dtype=tf.int32)

item\_embedding = tf.keras.layers.Embedding(n\_item, dim, embeddings\_regularizer=l2)(item\_id)

user\_bias = tf.keras.layers.Embedding(n\_user, 1, embeddings\_initializer='zeros')(user\_id)

item\_bias = tf.keras.layers.Embedding(n\_item, 1, embeddings\_initializer='zeros')(item\_id)

fm = tf.reduce\_sum(user\_embedding \* item\_embedding, axis=1, keepdims=True) + user\_bias + item\_bias

deep = tf.concat([user\_embedding, item\_embedding], axis=1)

for layer in layers:

deep = tf.keras.layers.Dense(layer, activation='relu', kernel\_regularizer=l2)(deep)

deep = tf.keras.layers.Dense(1, kernel\_regularizer=l2)(deep)

out = tf.keras.activations.sigmoid(fm + deep)

return tf.keras.Model(inputs=[user\_id, item\_id], outputs=out)

if \_\_name\_\_ == '\_\_main\_\_':

tf.keras.utils.plot\_model(DeepFM\_model(1, 1), 'graph.png', show\_shapes=True)

algorithm/FM/main.py

if \_\_name\_\_ == '\_\_main\_\_':

import Recommender\_System.utility.gpu\_memory\_growth

from Recommender\_System.data import data\_loader, data\_process

from Recommender\_System.algorithm.FM.model import FM\_model

from Recommender\_System.algorithm.train import train

n\_user, n\_item, train\_data, test\_data, topk\_data = data\_process.pack(data\_loader.ml100k)

model = FM\_model(n\_user, n\_item, dim=16, l2=1e-6)

train(model, train\_data, test\_data, topk\_data, epochs=10, batch=512)

algorithm/FM/model.py

import tensorflow as tf

from Recommender\_System.utility.decorator import logger

@logger('初始化FM模型：', ('n\_user', 'n\_item', 'dim', 'l2'))

def FM\_model(n\_user: int, n\_item: int, dim=8, l2=1e-6) -> tf.keras.Model:

l2 = tf.keras.regularizers.l2(l2)

user\_id = tf.keras.Input(shape=(), name='user\_id', dtype=tf.int32)

user\_embedding = tf.keras.layers.Embedding(n\_user, dim, embeddings\_regularizer=l2)(user\_id)

user\_bias = tf.keras.layers.Embedding(n\_user, 1, embeddings\_initializer='zeros')(user\_id)

item\_id = tf.keras.Input(shape=(), name='item\_id', dtype=tf.int32)

item\_embedding = tf.keras.layers.Embedding(n\_item, dim, embeddings\_regularizer=l2)(item\_id)

item\_bias = tf.keras.layers.Embedding(n\_item, 1, embeddings\_initializer='zeros')(item\_id)

x = tf.reduce\_sum(user\_embedding \* item\_embedding, axis=1, keepdims=True) + user\_bias + item\_bias

out = tf.keras.activations.sigmoid(x)

return tf.keras.Model(inputs=[user\_id, item\_id], outputs=out)

if \_\_name\_\_ == '\_\_main\_\_':

tf.keras.utils.plot\_model(FM\_model(1, 1), 'graph.png', show\_shapes=True)

algorithm/KGCN/layer.py

from abc import abstractmethod

import tensorflow as tf

class Aggregator(tf.keras.layers.Layer):

def \_\_init\_\_(self, activation='relu', kernel\_regularizer=None, \*\*kwargs):

super(Aggregator, self).\_\_init\_\_(\*\*kwargs)

self.activation = tf.keras.activations.get(activation)

self.kernel\_regularizer = tf.keras.regularizers.get(kernel\_regularizer)

def call(self, inputs, \*\*kwargs):

self\_vectors, neighbor\_vectors, neighbor\_relations, user\_embeddings = inputs

\_, neighbor\_iter, dim = self\_vectors.shape

neighbor\_size = kwargs['neighbor\_size']

neighbor\_vectors = tf.reshape(neighbor\_vectors, shape=(-1, neighbor\_iter, neighbor\_size, dim))

neighbor\_relations = tf.reshape(neighbor\_relations, shape=(-1, neighbor\_iter, neighbor\_size, dim))

outputs = self.\_call(self\_vectors, neighbor\_vectors, neighbor\_relations, user\_embeddings, \*\*kwargs)

if self.activation is not None:

outputs = self.activation(outputs)

return outputs

@abstractmethod

def \_call(self, self\_vectors, neighbor\_vectors, neighbor\_relations, user\_embeddings, \*\*kwargs):

# self\_vectors: [batch, neighbor\_iter, dim]

# neighbor\_vectors: [batch, neighbor\_iter, neighbor\_size, dim]

# neighbor\_relations: [batch, neighbor\_iter, neighbor\_size, dim]

# user\_embeddings: [batch, dim]

pass

def \_mix\_neighbor\_vectors(self, neighbor\_vectors, neighbor\_relations, user\_embeddings):

dim = user\_embeddings.shape[-1]

avg = False

if not avg:

user\_embeddings = tf.reshape(user\_embeddings, shape=(-1, 1, 1, dim)) # [batch, 1, 1, dim]

user\_relation\_scores = tf.reduce\_mean(user\_embeddings \* neighbor\_relations, axis=-1) # [batch, neighbor\_iter, neighbor\_size]

user\_relation\_scores\_normalized = tf.nn.softmax(user\_relation\_scores, axis=-1) # [batch, neighbor\_iter, neighbor\_size]

user\_relation\_scores\_normalized = tf.expand\_dims(user\_relation\_scores\_normalized, axis=-1) # [batch, neighbor\_iter, neighbor\_size, 1]

neighbors\_aggregated = tf.reduce\_mean(user\_relation\_scores\_normalized \* neighbor\_vectors, axis=2) # [batch, neighbor\_iter, dim]

else:

neighbors\_aggregated = tf.reduce\_mean(neighbor\_vectors, axis=2) # [batch, neighbor\_iter, dim]

return neighbors\_aggregated

class SumAggregator(Aggregator):

def build(self, input\_shape):

dim = input\_shape[-1][-1]

self.kernel = self.add\_weight('kernel', shape=(dim, dim), initializer='glorot\_uniform', regularizer=self.kernel\_regularizer)

self.bias = self.add\_weight('bias', shape=(dim,), initializer='zeros')

def \_call(self, self\_vectors, neighbor\_vectors, neighbor\_relations, user\_embeddings, \*\*kwargs):

\_, neighbor\_iter, dim = self\_vectors.shape

neighbors\_agg = self.\_mix\_neighbor\_vectors(neighbor\_vectors, neighbor\_relations, user\_embeddings) # [batch, neighbor\_iter, dim]

output = tf.reshape(self\_vectors + neighbors\_agg, shape=(-1, dim)) # [batch \* neighbor\_iter, dim]

#if kwargs['training']:

# output = tf.nn.dropout(output, rate=0.2)

output = tf.nn.bias\_add(tf.matmul(output, self.kernel), self.bias) # [batch \* neighbor\_iter, dim]

return tf.reshape(output, shape=(-1, neighbor\_iter, dim)) # [batch, neighbor\_iter, dim]

class ConcatAggregator(Aggregator):

def build(self, input\_shape):

dim = input\_shape[-1][-1]

self.kernel = self.add\_weight('kernel', shape=(dim \* 2, dim), initializer='glorot\_uniform', regularizer=self.kernel\_regularizer)

self.bias = self.add\_weight('bias', shape=(dim,), initializer='zeros')

def \_call(self, self\_vectors, neighbor\_vectors, neighbor\_relations, user\_embeddings, \*\*kwargs):

\_, neighbor\_iter, dim = self\_vectors.shape

neighbors\_agg = self.\_mix\_neighbor\_vectors(neighbor\_vectors, neighbor\_relations, user\_embeddings) # [batch, neighbor\_iter, dim]

output = tf.concat([self\_vectors, neighbors\_agg], axis=2) # [batch, neighbor\_iter, dim \* 2]

output = tf.reshape(output, shape=(-1, dim \* 2)) # [batch \* neighbor\_iter, dim \* 2]

#if kwargs['training']:

# output = tf.nn.dropout(output, rate=0.2)

output = tf.nn.bias\_add(tf.matmul(output, self.kernel), self.bias) # [batch \* neighbor\_iter, dim]

return tf.reshape(output, shape=(-1, neighbor\_iter, dim)) # [batch, neighbor\_iter, dim]

class NeighborAggregator(Aggregator):

def build(self, input\_shape):

dim = input\_shape[-1][-1]

self.kernel = self.add\_weight('kernel', shape=(dim, dim), initializer='glorot\_uniform', regularizer=self.kernel\_regularizer)

self.bias = self.add\_weight('bias', shape=(dim,), initializer='zeros')

def \_call(self, self\_vectors, neighbor\_vectors, neighbor\_relations, user\_embeddings, \*\*kwargs):

\_, neighbor\_iter, dim = self\_vectors.shape

neighbors\_agg = self.\_mix\_neighbor\_vectors(neighbor\_vectors, neighbor\_relations, user\_embeddings) # [batch, neighbor\_iter, dim]

output = tf.reshape(neighbors\_agg, shape=(-1, dim)) # [batch \* neighbor\_iter, dim]

#if kwargs['training']:

# output = tf.nn.dropout(output, rate=0.2)

output = tf.nn.bias\_add(tf.matmul(output, self.kernel), self.bias) # [batch \* neighbor\_iter, dim]

return tf.reshape(output, shape=(-1, neighbor\_iter, dim)) # [batch, neighbor\_iter, dim]

algorithm/KGCN/main.py

if \_\_name\_\_ == '\_\_main\_\_':

import Recommender\_System.utility.gpu\_memory\_growth

from Recommender\_System.algorithm.KGCN.tool import construct\_undirected\_kg, get\_adj\_list

from Recommender\_System.algorithm.KGCN.model import KGCN\_model

from Recommender\_System.algorithm.KGCN.train import train

from Recommender\_System.data import kg\_loader, data\_process

import tensorflow as tf

n\_user, n\_item, n\_entity, n\_relation, train\_data, test\_data, kg, topk\_data = data\_process.pack\_kg(kg\_loader.ml1m\_kg1m, negative\_sample\_threshold=4)

neighbor\_size = 16

adj\_entity, adj\_relation = get\_adj\_list(construct\_undirected\_kg(kg), n\_entity, neighbor\_size)

model = KGCN\_model(n\_user, n\_entity, n\_relation, adj\_entity, adj\_relation, neighbor\_size, iter\_size=1, dim=16, l2=1e-7, aggregator='sum')

train(model, train\_data, test\_data, topk\_data, optimizer=tf.keras.optimizers.Adam(0.01), epochs=10, batch=512)

algorithm/KGCN/model.py

if \_\_name\_\_ == '\_\_main\_\_':

import Recommender\_System.utility.gpu\_memory\_growth

from Recommender\_System.algorithm.KGCN.tool import construct\_undirected\_kg, get\_adj\_list

from Recommender\_System.algorithm.KGCN.model import KGCN\_model

from Recommender\_System.algorithm.KGCN.train import train

from Recommender\_System.data import kg\_loader, data\_process

import tensorflow as tf

n\_user, n\_item, n\_entity, n\_relation, train\_data, test\_data, kg, topk\_data = data\_process.pack\_kg(kg\_loader.ml1m\_kg1m, negative\_sample\_threshold=4)

neighbor\_size = 16

adj\_entity, adj\_relation = get\_adj\_list(construct\_undirected\_kg(kg), n\_entity, neighbor\_size)

model = KGCN\_model(n\_user, n\_entity, n\_relation, adj\_entity, adj\_relation, neighbor\_size, iter\_size=1, dim=16, l2=1e-7, aggregator='sum')

train(model, train\_data, test\_data, topk\_data, optimizer=tf.keras.optimizers.Adam(0.01), epochs=10, batch=512)

algorithm/KGCN/train.py

import time

from typing import List, Tuple

import tensorflow as tf

from Recommender\_System.utility.decorator import logger

from Recommender\_System.utility.evaluation import TopkData

from Recommender\_System.algorithm.train import prepare\_ds, get\_score\_fn

from Recommender\_System.algorithm.common import log, topk

@logger('开始训练，', ('epochs', 'batch'))

def train(model: tf.keras.Model, train\_data: List[Tuple[int, int, int]], test\_data: List[Tuple[int, int, int]],

topk\_data: TopkData = None, optimizer=None, epochs=100, batch=512):

if optimizer is None:

optimizer = tf.keras.optimizers.Adam()

train\_ds, test\_ds = prepare\_ds(train\_data, test\_data, batch)

loss\_mean\_metric = tf.keras.metrics.Mean()

auc\_metric = tf.keras.metrics.AUC()

precision\_metric = tf.keras.metrics.Precision()

recall\_metric = tf.keras.metrics.Recall()

loss\_object = tf.keras.losses.BinaryCrossentropy()

if topk\_data:

score\_fn = get\_score\_fn(model)

def reset\_metrics():

for metric in [loss\_mean\_metric, auc\_metric, precision\_metric, recall\_metric]:

tf.py\_function(metric.reset\_states, [], [])

def update\_metrics(loss, label, score):

loss\_mean\_metric.update\_state(loss)

auc\_metric.update\_state(label, score)

precision\_metric.update\_state(label, score)

recall\_metric.update\_state(label, score)

def get\_metric\_results():

return loss\_mean\_metric.result(), auc\_metric.result(), precision\_metric.result(), recall\_metric.result()

@tf.function

def train\_batch(ui, label):

with tf.GradientTape() as tape:

score = model(ui, training=True)

loss = loss\_object(label, score) + sum(model.losses)

gradients = tape.gradient(loss, model.trainable\_variables)

optimizer.apply\_gradients(zip(gradients, model.trainable\_variables))

update\_metrics(loss, label, score)

@tf.function

def test\_batch(ui, label):

score = model(ui)

loss = loss\_object(label, score) + sum(model.losses)

update\_metrics(loss, label, score)

for epoch in range(epochs):

epoch\_start\_time = time.time()

reset\_metrics()

for ui, label in train\_ds:

train\_batch(ui, label)

train\_loss, train\_auc, train\_precision, train\_recall = get\_metric\_results()

reset\_metrics()

for ui, label in test\_ds:

test\_batch(ui, label)

test\_loss, test\_auc, test\_precision, test\_recall = get\_metric\_results()

log(epoch, train\_loss, train\_auc, train\_precision, train\_recall, test\_loss, test\_auc, test\_precision, test\_recall)

if topk\_data:

topk(topk\_data, score\_fn)

print('epoch\_time=', time.time() - epoch\_start\_time, 's', sep='')

algorithm/KGCN/tool.py

from Recommender\_System.utility.decorator import logger

from typing import List, Tuple, Dict

from collections import defaultdict

import numpy as np

@logger('根据知识图谱结构构建无向图')

def construct\_undirected\_kg(kg: List[Tuple[int, int, int]]) -> Dict[int, List[Tuple[int, int]]]:

kg\_dict = defaultdict(list)

for head\_id, relation\_id, tail\_id in kg:

kg\_dict[head\_id].append((relation\_id, tail\_id))

kg\_dict[tail\_id].append((relation\_id, head\_id)) # 将知识图谱视为无向图

return kg\_dict

@logger('根据知识图谱无向图构建邻接表，', ('n\_entity', 'neighbor\_size'))

def get\_adj\_list(kg\_dict: Dict[int, List[Tuple[int, int]]], n\_entity: int, neighbor\_size: int) ->\

Tuple[List[List[int]], List[List[int]]]:

adj\_entity, adj\_relation = [None for \_ in range(n\_entity)], [None for \_ in range(n\_entity)]

for entity\_id in range(n\_entity):

neighbors = kg\_dict[entity\_id]

n\_neighbor = len(neighbors)

sample\_indices = np.random.choice(range(n\_neighbor), size=neighbor\_size, replace=n\_neighbor < neighbor\_size)

adj\_relation[entity\_id] = [neighbors[i][0] for i in sample\_indices]

adj\_entity[entity\_id] = [neighbors[i][1] for i in sample\_indices]

return adj\_entity, adj\_relation

algorithm/MKR/layer.py

import tensorflow as tf

class CrossLayer(tf.keras.layers.Layer):

def call(self, inputs):

v, e = inputs # (batch, dim)

v = tf.expand\_dims(v, axis=2) # (batch, dim, 1)

e = tf.expand\_dims(e, axis=1) # (batch, 1, dim)

c\_matrix = tf.matmul(v, e) # (batch, dim, dim)

c\_matrix\_t = tf.transpose(c\_matrix, perm=[0, 2, 1]) # (batch, dim, dim)

return c\_matrix, c\_matrix\_t

class CompressLayer(tf.keras.layers.Layer):

def \_\_init\_\_(self, weight\_regularizer, \*\*kwargs):

super(CompressLayer, self).\_\_init\_\_(\*\*kwargs)

self.weight\_regularizer = tf.keras.regularizers.get(weight\_regularizer)

def build(self, input\_shape):

self.dim = input\_shape[0][-1]

self.weight = self.add\_weight(shape=(self.dim, 1), regularizer=self.weight\_regularizer, name='weight')

self.weight\_t = self.add\_weight(shape=(self.dim, 1), regularizer=self.weight\_regularizer, name='weight\_t')

self.bias = self.add\_weight(shape=self.dim, initializer='zeros', name='bias')

def call(self, inputs):

c\_matrix, c\_matrix\_t = inputs # (batch, dim, dim)

c\_matrix = tf.reshape(c\_matrix, shape=[-1, self.dim]) # (batch \* dim, dim)

c\_matrix\_t = tf.reshape(c\_matrix\_t, shape=[-1, self.dim]) # (batch \* dim, dim)

return tf.reshape(tf.matmul(c\_matrix, self.weight) + tf.matmul(c\_matrix\_t, self.weight\_t),

shape=[-1, self.dim]) + self.bias # (batch, dim)

def cross\_compress\_unit(inputs, weight\_regularizer):

cross\_feature\_matrix = CrossLayer()(inputs)

v\_out = CompressLayer(weight\_regularizer)(cross\_feature\_matrix)

e\_out = CompressLayer(weight\_regularizer)(cross\_feature\_matrix)

return v\_out, e\_out

algorithm/MKR/main.py

if \_\_name\_\_ == '\_\_main\_\_':

import Recommender\_System.utility.gpu\_memory\_growth

from tensorflow.keras.optimizers import Adam

from Recommender\_System.data import kg\_loader, data\_process

from Recommender\_System.algorithm.MKR.model import MKR\_model

from Recommender\_System.algorithm.MKR.train import train

n\_user, n\_item, n\_entity, n\_relation, train\_data, test\_data, kg, topk\_data = data\_process.pack\_kg(kg\_loader.ml1m\_kg20k, keep\_all\_head=False, negative\_sample\_threshold=4)

model\_rs, model\_kge = MKR\_model(n\_user, n\_item, n\_entity, n\_relation, dim=8, L=1, H=1, l2=1e-6)

train(model\_rs, model\_kge, train\_data, test\_data, kg, topk\_data, kge\_interval=3,

optimizer\_rs=Adam(0.02), optimizer\_kge=Adam(0.01), epochs=20, batch=4096)

'''

n\_user, n\_item, n\_entity, n\_relation, train\_data, test\_data, kg, topk\_data = data\_process.pack\_kg(kg\_loader.lastfm\_kg15k, keep\_all\_head=False)

model\_rs, model\_kge = MKR\_model(n\_user, n\_item, n\_entity, n\_relation, dim=4, L=2, H=1, l2=1e-6)

train(model\_rs, model\_kge, train\_data, test\_data, kg, topk\_data, kge\_interval=2,

optimizer\_rs=Adam(1e-3), optimizer\_kge=Adam(2e-4), epochs=10, batch=256)

'''

'''

n\_user, n\_item, n\_entity, n\_relation, train\_data, test\_data, kg, topk\_data = data\_process.pack\_kg(kg\_loader.bx\_kg20k, keep\_all\_head=False)

model\_rs, model\_kge = MKR\_model(n\_user, n\_item, n\_entity, n\_relation, dim=8, L=1, H=1, l2=1e-6)

train(model\_rs, model\_kge, train\_data, test\_data, kg, topk\_data, kge\_interval=2,

optimizer\_rs=Adam(2e-4), optimizer\_kge=Adam(2e-5), epochs=10, batch=32)

'''

Algorithm/MKR/model.py

from typing import Tuple

import tensorflow as tf

from Recommender\_System.algorithm.MKR.layer import cross\_compress\_unit

from Recommender\_System.utility.decorator import logger

@logger('初始化MKR模型：', ('n\_user', 'n\_item', 'n\_entity', 'n\_relation', 'dim', 'L', 'H', 'l2'))

def MKR\_model(n\_user: int, n\_item: int, n\_entity: int, n\_relation: int, dim=8, L=1, H=1, l2=1e-6) -> Tuple[tf.keras.Model, tf.keras.Model]:

l2 = tf.keras.regularizers.l2(l2)

user\_id = tf.keras.Input(shape=(), name='user\_id', dtype=tf.int32)

item\_id = tf.keras.Input(shape=(), name='item\_id', dtype=tf.int32)

head\_id = tf.keras.Input(shape=(), name='head\_id', dtype=tf.int32)

relation\_id = tf.keras.Input(shape=(), name='relation\_id', dtype=tf.int32)

tail\_id = tf.keras.Input(shape=(), name='tail\_id', dtype=tf.int32)

user\_embedding = tf.keras.layers.Embedding(n\_user, dim, embeddings\_regularizer=l2)

item\_embedding = tf.keras.layers.Embedding(n\_item, dim, embeddings\_regularizer=l2)

entity\_embedding = tf.keras.layers.Embedding(n\_entity, dim, embeddings\_regularizer=l2)

relation\_embedding = tf.keras.layers.Embedding(n\_relation, dim, embeddings\_regularizer=l2)

u = user\_embedding(user\_id)

i = item\_embedding(item\_id)

h = entity\_embedding(head\_id)

r = relation\_embedding(relation\_id)

t = entity\_embedding(tail\_id)

for \_ in range(L):

u = tf.keras.layers.Dense(dim, activation='relu', kernel\_regularizer=l2)(u)

i, h = cross\_compress\_unit(inputs=(i, h), weight\_regularizer=l2)

t = tf.keras.layers.Dense(dim, activation='relu', kernel\_regularizer=l2)(t)

#rs = tf.concat([u, i], axis=1)

rs = tf.keras.activations.sigmoid(tf.reduce\_sum(u \* i, axis=1, keepdims=True))

kge = tf.concat([h, r], axis=1)

for \_ in range(H - 1):

#rs = tf.keras.layers.Dense(dim \* 2, activation='relu', kernel\_regularizer=reg\_l2(l2))(rs)

kge = tf.keras.layers.Dense(dim \* 2, activation='relu', kernel\_regularizer=l2)(kge)

#rs = tf.keras.layers.Dense(1, activation='sigmoid', kernel\_regularizer=reg\_l2(l2))(rs)

kge = tf.keras.layers.Dense(dim, activation='sigmoid', kernel\_regularizer=l2)(kge)

kge = -tf.keras.activations.sigmoid(tf.reduce\_sum(t \* kge, axis=1))

return tf.keras.Model(inputs=[user\_id, item\_id, head\_id], outputs=rs),\

tf.keras.Model(inputs=[item\_id, head\_id, relation\_id, tail\_id], outputs=kge)

if \_\_name\_\_ == '\_\_main\_\_':

rs\_model, kge\_model = MKR\_model(2, 2, 2, 2)

u = tf.constant([0, 1])

i = tf.constant([1, 0])

h = tf.constant([0, 1])

r = tf.constant([1, 0])

t = tf.constant([0, 1])

print(rs\_model({'user\_id': u, 'item\_id': i, 'head\_id': h}))

print(kge\_model({'item\_id': i, 'head\_id': h, 'relation\_id': r, 'tail\_id': t}))

ds = tf.data.Dataset.from\_tensor\_slices(({'item\_id': i, 'head\_id': h, 'relation\_id': r, 'tail\_id': t}, tf.constant([0] \* 2))).batch(2)

kge\_model.compile(optimizer='adam', loss=lambda y\_true, y\_pre: y\_pre)

kge\_model.fit(ds, epochs=3)

#ds = tf.data.Dataset.from\_tensor\_slices(({'user\_id': u, 'item\_id': i, 'head\_id': h}, tf.constant([0., 1.]))).batch(2)

#rs\_model.compile(optimizer='adam', loss=tf.keras.losses.BinaryCrossentropy())

#rs\_model.fit(ds, epochs=3)

algorithm/MKR/train.py

from typing import List, Tuple

import tensorflow as tf

from Recommender\_System.algorithm.train import RsCallback

from Recommender\_System.utility.evaluation import TopkData

from Recommender\_System.utility.decorator import logger

class \_KgeCallback(tf.keras.callbacks.Callback):

def on\_epoch\_end(self, epoch, logs=None):

tf.print('KGE: epoch=', epoch + 1, ', loss=', logs['loss'], sep='')

def \_get\_score\_fn(model):

@tf.function(experimental\_relax\_shapes=True)

def \_fast\_model(inputs):

return tf.squeeze(model(inputs))

def \_score\_fn(inputs):

inputs = {k: tf.constant(v, dtype=tf.int32) for k, v in inputs.items()}

inputs['head\_id'] = inputs['item\_id']

return \_fast\_model(inputs).numpy()

return \_score\_fn

@logger('开始训练，', ('epochs', 'batch'))

def train(model\_rs: tf.keras.Model, model\_kge: tf.keras.Model, train\_data: List[Tuple[int, int, int]],

test\_data: List[Tuple[int, int, int]], kg: List[Tuple[int, int, int]], topk\_data: TopkData,

optimizer\_rs=None, optimizer\_kge=None, kge\_interval=3, epochs=100, batch=512):

if optimizer\_rs is None:

optimizer\_rs = tf.keras.optimizers.Adam()

if optimizer\_kge is None:

optimizer\_kge = tf.keras.optimizers.Adam()

def xy(data):

user\_id = tf.constant([d[0] for d in data], dtype=tf.int32)

item\_id = tf.constant([d[1] for d in data], dtype=tf.int32)

head\_id = tf.constant([d[1] for d in data], dtype=tf.int32)

label = tf.constant([d[2] for d in data], dtype=tf.float32)

return {'user\_id': user\_id, 'item\_id': item\_id, 'head\_id': head\_id}, label

def xy\_kg(kg):

item\_id = tf.constant([d[0] for d in kg], dtype=tf.int32)

head\_id = tf.constant([d[0] for d in kg], dtype=tf.int32)

relation\_id = tf.constant([d[1] for d in kg], dtype=tf.int32)

tail\_id = tf.constant([d[2] for d in kg], dtype=tf.int32)

label = tf.constant([0] \* len(kg), dtype=tf.float32)

return {'item\_id': item\_id, 'head\_id': head\_id, 'relation\_id': relation\_id, 'tail\_id': tail\_id}, label

train\_ds = tf.data.Dataset.from\_tensor\_slices(xy(train\_data)).shuffle(len(train\_data)).batch(batch)

test\_ds = tf.data.Dataset.from\_tensor\_slices(xy(test\_data)).batch(batch)

kg\_ds = tf.data.Dataset.from\_tensor\_slices(xy\_kg(kg)).shuffle(len(kg)).batch(batch)

model\_rs.compile(optimizer=optimizer\_rs, loss='binary\_crossentropy', metrics=['AUC', 'Precision', 'Recall'])

model\_kge.compile(optimizer=optimizer\_kge, loss=lambda y\_true, y\_pre: y\_pre)

for epoch in range(epochs):

model\_rs.fit(train\_ds, epochs=epoch + 1, verbose=0, validation\_data=test\_ds,

callbacks=[RsCallback(topk\_data, \_get\_score\_fn(model\_rs))], initial\_epoch=epoch)

if epoch % kge\_interval == 0:

model\_kge.fit(kg\_ds, epochs=epoch + 1, verbose=0, callbacks=[\_KgeCallback()], initial\_epoch=epoch)

algorithm/common.py

from typing import List, Callable, Dict

from Recommender\_System.utility.evaluation import TopkData, topk\_evaluate

def log(epoch, train\_loss, train\_auc, train\_precision, train\_recall, test\_loss, test\_auc, test\_precision, test\_recall):

train\_f1 = 2. \* train\_precision \* train\_recall / pr if (pr := train\_precision + train\_recall) else 0

test\_f1 = 2. \* test\_precision \* test\_recall / pr if (pr := test\_precision + test\_recall) else 0

print('epoch=%d, train\_loss=%.5f, train\_auc=%.5f, train\_f1=%.5f, test\_loss=%.5f, test\_auc=%.5f, test\_f1=%.5f' %

(epoch + 1, train\_loss, train\_auc, train\_f1, test\_loss, test\_auc, test\_f1))

def topk(topk\_data: TopkData, score\_fn: Callable[[Dict[str, List[int]]], List[float]], ks=[10, 36, 100]):

precisions, recalls = topk\_evaluate(topk\_data, score\_fn, ks)

for k, precision, recall in zip(ks, precisions, recalls):

f1 = 2. \* precision \* recall / pr if (pr := precision + recall) else 0

print('[k=%d, precision=%.3f%%, recall=%.3f%%, f1=%.3f%%]' %

(k, 100. \* precision, 100. \* recall, 100. \* f1), end='')

print()

algorithm/train.py (not-a-script meaning a dependency file)

from typing import List, Tuple, Callable, Dict

import tensorflow as tf

from Recommender\_System.algorithm.common import log, topk

from Recommender\_System.utility.evaluation import TopkData

from Recommender\_System.utility.decorator import logger

def prepare\_ds(train\_data: List[Tuple[int, int, int]], test\_data: List[Tuple[int, int, int]],

batch: int) -> Tuple[tf.data.Dataset, tf.data.Dataset]:

def xy(data):

user\_ids = tf.constant([d[0] for d in data], dtype=tf.int32)

item\_ids = tf.constant([d[1] for d in data], dtype=tf.int32)

labels = tf.constant([d[2] for d in data], dtype=tf.keras.backend.floatx())

return {'user\_id': user\_ids, 'item\_id': item\_ids}, labels

train\_ds = tf.data.Dataset.from\_tensor\_slices(xy(train\_data)).shuffle(len(train\_data)).batch(batch)

test\_ds = tf.data.Dataset.from\_tensor\_slices(xy(test\_data)).batch(batch)

return train\_ds, test\_ds

def \_evaluate(model, dataset, loss\_object, mean\_metric=tf.keras.metrics.Mean(), auc\_metric=tf.keras.metrics.AUC(),

precision\_metric=tf.keras.metrics.Precision(), recall\_metric=tf.keras.metrics.Recall()):

for metric in [mean\_metric, auc\_metric, precision\_metric, recall\_metric]:

tf.py\_function(metric.reset\_states, [], [])

@tf.function

def evaluate\_batch(ui, label):

score = tf.squeeze(model(ui))

loss = loss\_object(label, score) + sum(model.losses)

return score, loss

for ui, label in dataset:

score, loss = evaluate\_batch(ui, label)

mean\_metric.update\_state(loss)

auc\_metric.update\_state(label, score)

precision\_metric.update\_state(label, score)

recall\_metric.update\_state(label, score)

return mean\_metric.result(), auc\_metric.result(), precision\_metric.result(), recall\_metric.result()

def \_train\_graph(model, train\_ds, test\_ds, topk\_data, optimizer, loss\_object, epochs):

score\_fn = get\_score\_fn(model)

@tf.function

def train\_batch(ui, label):

with tf.GradientTape() as tape:

score = tf.squeeze(model(ui, training=True))

loss = loss\_object(label, score) + sum(model.losses)

gradients = tape.gradient(loss, model.trainable\_variables)

optimizer.apply\_gradients(zip(gradients, model.trainable\_variables))

for epoch in range(epochs):

for ui, label in train\_ds:

train\_batch(ui, label)

train\_loss, train\_auc, train\_precision, train\_recall = \_evaluate(model, train\_ds, loss\_object)

test\_loss, test\_auc, test\_precision, test\_recall = \_evaluate(model, test\_ds, loss\_object)

log(epoch, train\_loss, train\_auc, train\_precision, train\_recall, test\_loss, test\_auc, test\_precision, test\_recall)

topk(topk\_data, score\_fn)

def \_train\_eager(model, train\_ds, test\_ds, topk\_data, optimizer, loss\_object, epochs):

model.compile(optimizer=optimizer, loss=loss\_object, metrics=['AUC', 'Precision', 'Recall'])

model.fit(train\_ds, epochs=epochs, verbose=0, validation\_data=test\_ds,

callbacks=[RsCallback(topk\_data, get\_score\_fn(model))])

class RsCallback(tf.keras.callbacks.Callback):

def \_\_init\_\_(self, topk\_data: TopkData, score\_fn: Callable[[Dict[str, List[int]]], List[float]]):

super(RsCallback, self).\_\_init\_\_()

self.topk\_data = topk\_data

self.score\_fn = score\_fn

def on\_epoch\_end(self, epoch, logs=None):

log(epoch, logs['loss'], logs['auc'], logs['precision'], logs['recall'],

logs['val\_loss'], logs['val\_auc'], logs['val\_precision'], logs['val\_recall'])

topk(self.topk\_data, self.score\_fn)

@logger('开始训练，', ('epochs', 'batch', 'execution'))

def train(model: tf.keras.Model, train\_data: List[Tuple[int, int, int]], test\_data: List[Tuple[int, int, int]],

topk\_data: TopkData, optimizer=None, loss\_object=None, epochs=100, batch=512, execution='eager') -> None:

"""

通用训练流程。

:param model: 模型

:param train\_data: 训练集

:param test\_data: 测试集

:param topk\_data: 用于topk评估数据

:param optimizer: 优化器，默认为Adam

:param loss\_object: 损失函数，默认为BinaryCrossentropy

:param epochs: 迭代次数

:param batch: 批数量

:param execution: 执行模式，为eager或graph。在eager模式下，用model.fit；在graph模式下，用tf.function和GradientTape

"""

if optimizer is None:

optimizer = tf.keras.optimizers.Adam()

if loss\_object is None:

loss\_object = tf.keras.losses.BinaryCrossentropy()

train\_ds, test\_ds = prepare\_ds(train\_data, test\_data, batch)

train\_fn = \_train\_eager if execution == 'eager' else \_train\_graph

train\_fn(model, train\_ds, test\_ds, topk\_data, optimizer, loss\_object, epochs)

@logger('开始测试，', ('batch',))

def test(model: tf.keras.Model, train\_data: List[Tuple[int, int, int]], test\_data: List[Tuple[int, int, int]],

topk\_data: TopkData, loss\_object=None, batch=512) -> None:

"""

通用测试流程。

:param model: 模型

:param train\_data: 训练集

:param test\_data: 测试集

:param topk\_data: 用于topk评估数据

:param loss\_object: 损失函数，默认为BinaryCrossentropy

:param batch: 批数量

"""

if loss\_object is None:

loss\_object = tf.keras.losses.BinaryCrossentropy()

train\_ds, test\_ds = prepare\_ds(train\_data, test\_data, batch)

train\_loss, train\_auc, train\_precision, train\_recall = \_evaluate(model, train\_ds, loss\_object)

test\_loss, test\_auc, test\_precision, test\_recall = \_evaluate(model, test\_ds, loss\_object)

log(-1, train\_loss, train\_auc, train\_precision, train\_recall, test\_loss, test\_auc, test\_precision, test\_recall)

topk(topk\_data, get\_score\_fn(model))

def get\_score\_fn(model):

@tf.function(experimental\_relax\_shapes=True)

def \_fast\_model(ui):

return tf.squeeze(model(ui))

def score\_fn(ui):

ui = {k: tf.constant(v, dtype=tf.int32) for k, v in ui.items()}

return \_fast\_model(ui).numpy()

return score\_fn

data/data\_loader.py

import os

from typing import List, Callable, Tuple

from Recommender\_System.utility.decorator import logger

# 记下ds文件夹的路径，确保其它py文件调用时读文件路径正确

ds\_path = os.path.join(os.path.dirname(\_\_file\_\_), 'ds')

def \_read\_ml(relative\_path: str, separator: str) -> List[Tuple[int, int, int, int]]:

data = []

with open(os.path.join(ds\_path, relative\_path), 'r') as f:

for line in f.readlines():

values = line.strip().split(separator)

user\_id, movie\_id, rating, timestamp = int(values[0]), int(values[1]), int(values[2]), int(values[3])

data.append((user\_id, movie\_id, rating, timestamp))

return data

def \_read\_ml100k() -> List[Tuple[int, int, int, int]]:

return \_read\_ml('ml-100k/u.data', '\t')

def \_read\_ml1m() -> List[Tuple[int, int, int, int]]:

return \_read\_ml('ml-1m/ratings.dat', '::')

def \_read\_ml20m() -> List[Tuple[int, int, float, int]]:

data = []

with open(os.path.join(ds\_path, 'ml-20m/ratings.csv'), 'r') as f:

for line in f.readlines()[1:]:

values = line.strip().split(',')

user\_id, movie\_id, rating, timestamp = int(values[0]), int(values[1]), float(values[2]), int(values[3])

data.append((user\_id, movie\_id, rating, timestamp))

return data

def \_read\_lastfm() -> List[Tuple[int, int, int]]:

data = []

with open(os.path.join(ds\_path, 'lastfm-2k/user\_artists.dat'), 'r') as f:

for line in f.readlines()[1:]:

values = line.strip().split('\t')

user\_id, artist\_id, weight = int(values[0]), int(values[1]), int(values[2])

data.append((user\_id, artist\_id, weight))

return data

def \_read\_book\_crossing() -> List[Tuple[int, str, int]]:

data = []

with open(os.path.join(ds\_path, 'Book-Crossing/BX-Book-Ratings.csv'), 'r', encoding='utf-8') as f:

for line in f.readlines()[1:]:

values = line.strip().split(';')

user\_id, book\_id, rating = int(values[0][1:-1]), values[1][1:-1], int(values[2][1:-1])

data.append((user\_id, book\_id, rating))

return data

@logger('开始读数据，', ('data\_name', 'expect\_length', 'expect\_user', 'expect\_item'))

def \_load\_data(read\_data\_fn: Callable[[], List[tuple]], expect\_length: int, expect\_user: int, expect\_item: int,

data\_name: str) -> List[tuple]:

data = read\_data\_fn()

n\_user, n\_item = len(set(d[0] for d in data)), len(set(d[1] for d in data))

assert len(data) == expect\_length, data\_name + ' length ' + str(len(data)) + ' != ' + str(expect\_length)

assert n\_user == expect\_user, data\_name + ' user ' + str(n\_user) + ' != ' + str(expect\_user)

assert n\_item == expect\_item, data\_name + ' item ' + str(n\_item) + ' != ' + str(expect\_item)

return data

def ml100k() -> List[Tuple[int, int, int, int]]:

return \_load\_data(\_read\_ml100k, 100000, 943, 1682, 'ml100k')

def ml1m() -> List[Tuple[int, int, int, int]]:

return \_load\_data(\_read\_ml1m, 1000209, 6040, 3706, 'ml1m')

def ml20m() -> List[Tuple[int, int, float, int]]:

return \_load\_data(\_read\_ml20m, 20000263, 138493, 26744, 'ml20m')

def lastfm() -> List[Tuple[int, int, int]]:

return \_load\_data(\_read\_lastfm, 92834, 1892, 17632, 'lastfm')

def book\_crossing() -> List[Tuple[int, str, int]]:

return \_load\_data(\_read\_book\_crossing, 1149780, 105283, 340555, 'Book-Crossing')

# 测试数据读的是否正确

if \_\_name\_\_ == '\_\_main\_\_':

data = book\_crossing()

data/data\_process.py

import os

import random

import numpy as np

from typing import Tuple, List, Callable

from collections import defaultdict

from Recommender\_System.utility.evaluation import TopkData

from Recommender\_System.utility.decorator import logger

@logger('开始采集负样本，', ('ratio', 'threshold', 'method'))

def negative\_sample(data: List[tuple], ratio=1, threshold=0, method='random') -> List[tuple]:

"""

采集负样本

保证了每个用户都有正样本，但是不保证每个物品都有正样本，可能会减少用户数量和物品数量

:param data: 原数据，至少有三列，第一列是用户id，第二列是物品id，第三列是权重

:param ratio: 负正样本比例

:param threshold: 权重阈值，权重大于或者等于此值为正样例，小于此值既不是正样例也不是负样例

:param method: 采集方式，random是均匀随机采集，popular是按流行度随机采集

:return: 带上负样本的数据集

"""

# 负样本采集权重

if method == 'random':

negative\_sample\_weight = {d[1]: 1 for d in data}

elif method == 'popular':

negative\_sample\_weight = {d[1]: 0 for d in data}

for d in data:

negative\_sample\_weight[d[1]] += 1

else:

raise ValueError("参数method必须是'random'或'popular'")

# 得到每个用户正样本与非正样本集合

user\_positive\_set, user\_unpositive\_set = defaultdict(set), defaultdict(set)

for d in data:

user\_id, item\_id, weight = d[0], d[1], d[2]

(user\_positive\_set if weight >= threshold else user\_unpositive\_set)[user\_id].add(item\_id)

# 仅为有正样例的用户采集负样例

user\_list = list(user\_positive\_set.keys())

arg\_positive\_set = [user\_positive\_set[user\_id] for user\_id in user\_list]

arg\_unpositive\_set = [user\_unpositive\_set[user\_id] for user\_id in user\_list]

from concurrent.futures import ProcessPoolExecutor

with ProcessPoolExecutor(max\_workers=os.cpu\_count()//2, initializer=\_negative\_sample\_init, initargs=(ratio, negative\_sample\_weight)) as executor:

sampled\_negative\_items = executor.map(\_negative\_sample, arg\_positive\_set, arg\_unpositive\_set, chunksize=100)

# 构建新的数据集

new\_data = []

for user\_id, negative\_items in zip(user\_list, sampled\_negative\_items):

new\_data.extend([(user\_id, item\_id, 0) for item\_id in negative\_items])

for user\_id, positive\_items in user\_positive\_set.items():

new\_data.extend([(user\_id, item\_id, 1) for item\_id in positive\_items])

return new\_data

def \_negative\_sample\_init(\_ratio, \_negative\_sample\_weight): # 用于子进程初始化全局变量

global item\_set, ratio, negative\_sample\_weight

item\_set, ratio, negative\_sample\_weight = set(\_negative\_sample\_weight.keys()), \_ratio, \_negative\_sample\_weight

def \_negative\_sample(positive\_set, unpositive\_set): # 对单个用户进行负采样

valid\_negative\_list = list(item\_set - positive\_set - unpositive\_set) # 可以取负样例的物品id列表

n\_negative\_sample = min(int(len(positive\_set) \* ratio), len(valid\_negative\_list)) # 采集负样例数量

if n\_negative\_sample <= 0:

return []

weights = np.array([negative\_sample\_weight[item\_id] for item\_id in valid\_negative\_list], dtype=np.float)

weights /= weights.sum() # 负样本采集权重

# 采集n\_negative\_sample个负样例（通过下标采样是为了防止物品id类型从int或str变成np.int或np.str）

sample\_indices = np.random.choice(range(len(valid\_negative\_list)), n\_negative\_sample, False, weights)

return [valid\_negative\_list[i] for i in sample\_indices]

@logger('开始进行id规整化')

def neaten\_id(data: List[tuple]) -> Tuple[List[Tuple[int, int, int]], int, int, dict, dict]:

"""

对数据的用户id和物品id进行规整化，使其id变为从0开始到数量减1

:param data: 原数据，有三列，第一列是用户id，第二列是物品id，第三列是标签

:return: 新数据，用户数量，物品数量，用户id旧到新映射，物品id旧到新映射

"""

new\_data = []

n\_user, n\_item = 0, 0

user\_id\_old2new, item\_id\_old2new = {}, {}

for user\_id\_old, item\_id\_old, label in data:

if user\_id\_old not in user\_id\_old2new:

user\_id\_old2new[user\_id\_old] = n\_user

n\_user += 1

if item\_id\_old not in item\_id\_old2new:

item\_id\_old2new[item\_id\_old] = n\_item

n\_item += 1

new\_data.append((user\_id\_old2new[user\_id\_old], item\_id\_old2new[item\_id\_old], label))

return new\_data, n\_user, n\_item, user\_id\_old2new, item\_id\_old2new

@logger('开始数据切分，', ('test\_ratio', 'shuffle', 'ensure\_positive'))

def split(data: List[tuple], test\_ratio=0.4, shuffle=True, ensure\_positive=False) -> Tuple[List[tuple], List[tuple]]:

"""

将数据切分为训练集数据和测试集数据

:param data: 原数据，第一列为用户id，第二列为物品id，第三列为标签

:param test\_ratio: 测试集数据占比，这个值在0和1之间

:param shuffle: 是否对原数据随机排序

:param ensure\_positive: 是否确保训练集每个用户都有正样例

:return: 训练集数据和测试集数据

"""

if shuffle:

random.shuffle(data)

n\_test = int(len(data) \* test\_ratio)

test\_data, train\_data = data[:n\_test], data[n\_test:]

if ensure\_positive:

user\_set = {d[0] for d in data} - {user\_id for user\_id, \_, label in train\_data if label == 1}

if len(user\_set) > 0:

print('警告：为了确保训练集数据每个用户都有正样例，%d(%f%%)条数据从测试集随机插入训练集'

% (len(user\_set), 100 \* len(user\_set) / len(data)))

i = len(test\_data) - 1

while len(user\_set) > 0:

assert i >= 0, '无法确保训练集每个用户都有正样例，因为存在没有正样例的用户：' + str(user\_set)

if test\_data[i][0] in user\_set and test\_data[i][2] == 1:

user\_set.remove(test\_data[i][0])

train\_data.insert(random.randint(0, len(train\_data)), test\_data.pop(i))

i -= 1

return train\_data, test\_data

@logger('开始准备topk评估数据，', ('n\_sample\_user',))

def prepare\_topk(train\_data: List[Tuple[int, int, int]], test\_data: List[Tuple[int, int, int]],

n\_user: int, n\_item: int, n\_sample\_user=None) -> TopkData:

"""

准备用于topk评估的数据

:param train\_data: 训练集数据，有三列，分别是user\_id, item\_id, label

:param test\_data: 测试集数据，有三列，分别是user\_id, item\_id, label

:param n\_user: 用户数量

:param n\_item: 物品数量

:param n\_sample\_user: 用户取样数量，为None则表示采样所有用户

:return: 用于topk评估的数据，类型为TopkData，其包括在测试集里每个用户的（可推荐物品集合）与（有行为物品集合）

"""

if n\_sample\_user is None or n\_sample\_user > n\_user:

n\_sample\_user = n\_user

user\_set = np.random.choice(range(n\_user), n\_sample\_user, False)

def get\_user\_item\_set(data: List[Tuple[int, int, int]], only\_positive=False):

user\_item\_set = {user\_id: set() for user\_id in user\_set}

for user\_id, item\_id, label in data:

if user\_id in user\_set and (not only\_positive or label == 1):

user\_item\_set[user\_id].add(item\_id)

return user\_item\_set

test\_user\_item\_set = {user\_id: set(range(n\_item)) - item\_set

for user\_id, item\_set in get\_user\_item\_set(train\_data).items()}

test\_user\_positive\_item\_set = get\_user\_item\_set(test\_data, only\_positive=True)

return TopkData(test\_user\_item\_set, test\_user\_positive\_item\_set)

def pack(data\_loader\_fn: Callable[[], List[tuple]],

negative\_sample\_ratio=1, negative\_sample\_threshold=0, negative\_sample\_method='random',

split\_test\_ratio=0.4, shuffle\_before\_split=True, split\_ensure\_positive=False,

topk\_sample\_user=300) -> Tuple[int, int, List[Tuple[int, int, int]], List[Tuple[int, int, int]], TopkData]:

"""

读数据，负采样，训练集测试集切分，准备TopK评估数据

:param data\_loader\_fn: data\_loader里面的读数据函数

:param negative\_sample\_ratio: 负正样本比例，为0代表不采样

:param negative\_sample\_threshold: 负采样的权重阈值，权重大于或者等于此值为正样例，小于此值既不是正样例也不是负样例

:param negative\_sample\_method: 负采样方法，值为'random'或'popular'

:param split\_test\_ratio: 切分时测试集占比，这个值在0和1之间

:param shuffle\_before\_split: 切分前是否对数据集随机顺序

:param split\_ensure\_positive: 切分时是否确保训练集每个用户都有正样例

:param topk\_sample\_user: 用来计算TopK指标时用户采样数量，为None则表示采样所有用户

:return: 用户数量，物品数量，训练集，测试集，用于TopK评估数据

"""

data = data\_loader\_fn()

if negative\_sample\_ratio > 0:

data = negative\_sample(data, negative\_sample\_ratio, negative\_sample\_threshold, negative\_sample\_method)

else:

data = [(d[0], d[1], 1) for d in data] # 变成隐反馈数据

data, n\_user, n\_item, \_, \_ = neaten\_id(data)

train\_data, test\_data = split(data, split\_test\_ratio, shuffle\_before\_split, split\_ensure\_positive)

topk\_data = prepare\_topk(train\_data, test\_data, n\_user, n\_item, topk\_sample\_user)

return n\_user, n\_item, train\_data, test\_data, topk\_data

def pack\_kg(kg\_loader\_config: Tuple[str, Callable[[], List[tuple]], type], keep\_all\_head=True,

negative\_sample\_ratio=1, negative\_sample\_threshold=0, negative\_sample\_method='random',

split\_test\_ratio=0.4, shuffle\_before\_split=True, split\_ensure\_positive=False,

topk\_sample\_user=100) -> Tuple[int, int, int, int, List[Tuple[int, int, int]],

List[Tuple[int, int, int]], List[Tuple[int, int, int]], TopkData]:

"""

联合读数据和知识图谱，训练集测试集切分，准备TopK评估数据

:param kg\_loader\_config: kg\_loader里面的读知识图谱配置

:param keep\_all\_head: 若为False，则读取知识图谱结构时，删除头实体在数据集里面没有对应物品的三元组

:param negative\_sample\_ratio: 负正样本比例，为0代表不采样

:param negative\_sample\_threshold: 负采样的权重阈值，权重大于或者等于此值为正样例，小于此值既不是正样例也不是负样例

:param negative\_sample\_method: 负采样方法，值为'random'或'popular'

:param split\_test\_ratio: 切分时测试集占比，这个值在0和1之间

:param shuffle\_before\_split: 切分前是否对数据集随机顺序

:param split\_ensure\_positive: 切分时是否确保训练集每个用户都有正样例

:param topk\_sample\_user: 用来计算TopK指标时用户采样数量，为None则表示采样所有用户

:return: 用户数量，物品数量，实体数量，关系数量，训练集，测试集，知识图谱，用于TopK评估数据

"""

from Recommender\_System.data.kg\_loader import \_read\_data\_with\_kg

data, kg, n\_user, n\_item, n\_entity, n\_relation = \_read\_data\_with\_kg(

kg\_loader\_config, negative\_sample\_ratio, negative\_sample\_threshold, negative\_sample\_method, keep\_all\_head)

train\_data, test\_data = split(data, split\_test\_ratio, shuffle\_before\_split, split\_ensure\_positive)

topk\_data = prepare\_topk(train\_data, test\_data, n\_user, n\_item, topk\_sample\_user)

return n\_user, n\_item, n\_entity, n\_relation, train\_data, test\_data, kg, topk\_data

data/kg\_loader.py

import os

from typing import Dict, List, Tuple, Callable, Any

from Recommender\_System.data import data\_loader, data\_process

from Recommender\_System.utility.decorator import logger

# 记下kg文件夹的路径，确保其它py文件调用时读文件路径正确

kg\_path = os.path.join(os.path.dirname(\_\_file\_\_), 'kg')

@logger('开始读物品实体映射关系，', ('kg\_directory', 'item\_id\_type'))

def \_read\_item\_id2entity\_id\_file(kg\_directory: str, item\_id\_type: type = int) -> Tuple[Dict[Any, int], Dict[int, Any]]:

item\_to\_entity = {}

entity\_to\_item = {}

with open(os.path.join(kg\_path, kg\_directory, 'item\_id2entity\_id.txt')) as f:

for line in f.readlines():

values = line.strip().split('\t')

item\_id = values[0] if item\_id\_type == str else item\_id\_type(values[0])

entity\_id = int(values[1])

item\_to\_entity[item\_id] = entity\_id

entity\_to\_item[entity\_id] = item\_id

return item\_to\_entity, entity\_to\_item

@logger('开始读知识图谱结构图，', ('kg\_directory', 'keep\_all\_head',))

def \_read\_kg\_file(kg\_directory: str, entity\_id\_old2new: Dict[int, int], keep\_all\_head=True) ->\

Tuple[List[Tuple[int, int, int]], int, int]:

n\_entity = len(entity\_id\_old2new)

relation\_id\_old2new = {}

n\_relation = 0

kg = []

with open(os.path.join(kg\_path, kg\_directory, 'kg.txt')) as f:

for line in f.readlines():

values = line.strip().split('\t')

head\_old, relation\_old, tail\_old = int(values[0]), values[1], int(values[2])

if head\_old not in entity\_id\_old2new:

if keep\_all\_head:

entity\_id\_old2new[head\_old] = n\_entity

n\_entity += 1

else:

continue

head = entity\_id\_old2new[head\_old]

if tail\_old not in entity\_id\_old2new:

entity\_id\_old2new[tail\_old] = n\_entity

n\_entity += 1

tail = entity\_id\_old2new[tail\_old]

if relation\_old not in relation\_id\_old2new:

relation\_id\_old2new[relation\_old] = n\_relation

n\_relation += 1

relation = relation\_id\_old2new[relation\_old]

kg.append((head, relation, tail))

return kg, n\_entity, n\_relation

@logger('----------开始载入带知识图谱的数据集：', end\_message='----------带知识图谱的数据集载入完成', log\_time=False)

def \_read\_data\_with\_kg(kg\_loader\_config: Tuple[str, Callable[[], List[tuple]], type],

negative\_sample\_ratio=1, negative\_sample\_threshold=0, negative\_sample\_method='random',

keep\_all\_head=True) -> Tuple[List[Tuple[int, int, int]], List[Tuple[int, int, int]],

int, int, int, int]:

kg\_directory, data\_loader\_fn, item\_id\_type = kg\_loader\_config

old\_item\_to\_old\_entity, old\_entity\_to\_old\_item = \_read\_item\_id2entity\_id\_file(kg\_directory, item\_id\_type)

data = data\_loader\_fn()

data = [d for d in data if d[1] in old\_item\_to\_old\_entity] # 去掉知识图谱中不存在的物品

data = data\_process.negative\_sample(data, negative\_sample\_ratio, negative\_sample\_threshold, negative\_sample\_method)

data, n\_user, n\_item, \_, item\_id\_old2new = data\_process.neaten\_id(data)

entity\_id\_old2new = {old\_entity: item\_id\_old2new[old\_item] for old\_entity, old\_item in old\_entity\_to\_old\_item.items()}

kg, n\_entity, n\_relation = \_read\_kg\_file(kg\_directory, entity\_id\_old2new, keep\_all\_head)

return data, kg, n\_user, n\_item, n\_entity, n\_relation

# kg\_loader\_configs: (kg\_directory, data\_loader\_fn, item\_id\_type)

bx\_kg20k = 'bx-kg20k', data\_loader.book\_crossing, str

bx\_kg150k = 'bx-kg150k', data\_loader.book\_crossing, str

lastfm\_kg15k = 'lastfm-kg15k', data\_loader.lastfm, int

ml1m\_kg20k = 'ml1m-kg20k', data\_loader.ml1m, int

ml1m\_kg1m = 'ml1m-kg1m', data\_loader.ml1m, int

ml20m\_kg500k = 'ml20m-kg500k', data\_loader.ml20m, int

if \_\_name\_\_ == '\_\_main\_\_':

data, kg, n\_user, n\_item, n\_entity, n\_relation = \_read\_data\_with\_kg(ml1m\_kg1m)

utility/competition.py

if \_\_name\_\_ == '\_\_main\_\_':

import Recommender\_System.utility.gpu\_memory\_growth

import tensorflow as tf

from Recommender\_System.data import data\_loader, data\_process

from Recommender\_System.algorithm.FM.model import FM\_model

from Recommender\_System.algorithm.GMF.model import GMF\_model

from Recommender\_System.algorithm.LFM.model import LFM\_model

from Recommender\_System.algorithm.MLP.model import MLP\_model

from Recommender\_System.algorithm.NeuMF.model import NeuMF\_model

from Recommender\_System.algorithm.DeepFM.model import DeepFM\_model

from Recommender\_System.algorithm.train import train

n\_user, n\_item, train\_data, test\_data, topk\_data = data\_process.pack(data\_loader.ml100k)

dim = 16

model = FM\_model(n\_user, n\_item, dim=dim, l2=0)

train(model, train\_data, test\_data, topk\_data, epochs=10)

model = GMF\_model(n\_user, n\_item, dim=dim, l2=0)

train(model, train\_data, test\_data, topk\_data, epochs=10)

model = LFM\_model(n\_user, n\_item, dim=dim, l2=0)

train(model, train\_data, test\_data, topk\_data, loss\_object=tf.losses.MeanSquaredError(), epochs=10)

model = MLP\_model(n\_user, n\_item, dim=dim \* 2, layers=[dim \* 2, dim, dim // 2], l2=0)

train(model, train\_data, test\_data, topk\_data, epochs=10)

model, \_, \_ = NeuMF\_model(n\_user, n\_item, gmf\_dim=dim // 2, mlp\_dim=dim \* 2, layers=[dim \* 2, dim, dim // 2], l2=0)

train(model, train\_data, test\_data, topk\_data, epochs=10)

model = DeepFM\_model(n\_user, n\_item, dim // 2, layers=[dim, dim, dim], l2=0)

train(model, train\_data, test\_data, topk\_data, epochs=10)

utility/decorator.py

import time

import inspect

from functools import wraps

from typing import Tuple

def arg\_value(arg\_name, f, args, kwargs):

if arg\_name in kwargs:

return kwargs[arg\_name]

i = f.\_\_code\_\_.co\_varnames.index(arg\_name)

if i < len(args):

return args[i]

return inspect.signature(f).parameters[arg\_name].default

def logger(begin\_message: str = None, log\_args: Tuple[str] = None, end\_message: str = None, log\_time: bool = True):

def logger\_decorator(f):

@wraps(f)

def decorated(\*args, \*\*kwargs):

if begin\_message is not None:

print(begin\_message, end='\n' if log\_args is None else '')

if log\_args is not None:

arg\_logs = [arg\_name + '=' + str(arg\_value(arg\_name, f, args, kwargs)) for arg\_name in log\_args]

print(', '.join(arg\_logs))

start\_time = time.time()

result = f(\*args, \*\*kwargs)

spent\_time = time.time() - start\_time

if end\_message is not None:

print(end\_message)

if log\_time:

print('（耗时', spent\_time, '秒）', sep='')

return result

return decorated

return logger\_decorator

utility/evaluation.py

from dataclasses import dataclass

from typing import Tuple, List, Callable, Dict

@dataclass

class TopkData:

test\_user\_item\_set: dict # 在测试集上每个用户可以参与推荐的物品集合

test\_user\_positive\_item\_set: dict # 在测试集上每个用户有行为的物品集合

@dataclass

class TopkStatistic:

hit: int = 0 # 命中数

ru: int = 0 # 推荐数

tu: int = 0 # 行为数

def topk\_evaluate(topk\_data: TopkData, score\_fn: Callable[[Dict[str, List[int]]], List[float]],

ks=[1, 2, 5, 10, 20, 50, 100]) -> Tuple[List[float], List[float]]:

kv = {k: TopkStatistic() for k in ks}

for user\_id, item\_set in topk\_data.test\_user\_item\_set.items():

ui = {'user\_id': [user\_id] \* len(item\_set), 'item\_id': list(item\_set)}

item\_score\_list = list(zip(item\_set, score\_fn(ui)))

sorted\_item\_list = [x[0] for x in sorted(item\_score\_list, key=lambda x: x[1], reverse=True)]

positive\_set = topk\_data.test\_user\_positive\_item\_set[user\_id]

for k in ks:

topk\_set = set(sorted\_item\_list[:k])

kv[k].hit += len(topk\_set & positive\_set)

kv[k].ru += len(topk\_set)

kv[k].tu += len(positive\_set)

return [kv[k].hit / kv[k].ru for k in ks], [kv[k].hit / kv[k].tu for k in ks] # precision, recall

utility/gpu\_memory\_growth.py

"""

import此文件后将gpu设置为显存增量模式

"""

from tensorflow import config

gpus = physical\_devices = config.list\_physical\_devices('GPU')

if len(gpus) == 0:

print('当前没有检测到gpu，设置显存增量模式无效。')

for gpu in gpus:

try:

config.experimental.set\_memory\_growth(gpu, True)

except RuntimeError as e:

print(e)