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
```