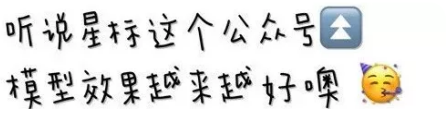
深度融合 | 当推荐系统遇见知识图谱

原创 上杉翔二 NewBeeNLP 1周前

收录于话题
#自然语言处理 41 #推荐搜索 7 #图网络学习 9



> NewBeeNLP原创出品
> 公众号专栏作者@上杉翔二
> 悠闲会 · 信息检索

上次我们看了『推荐系统 + GNN』

- **万物皆可Graph | 当推荐系统遇上图神经网络**

今天来看看『**推荐系统 + 知识图谱**』，又会有哪些有趣的玩意儿呢

Knowledge Graph

知识图谱是一种语义图，其结点（node）代表实体（entity）或者概念（concept），边（edge）代表实体/概念之间的各种语义关系（relation）。一个知识图谱由若干个三元组（h、r、t）组成，其中h和t代表一条关系的头结点和尾节点，r代表关系。

引入知识图谱进入推荐系统领域的优点在于：

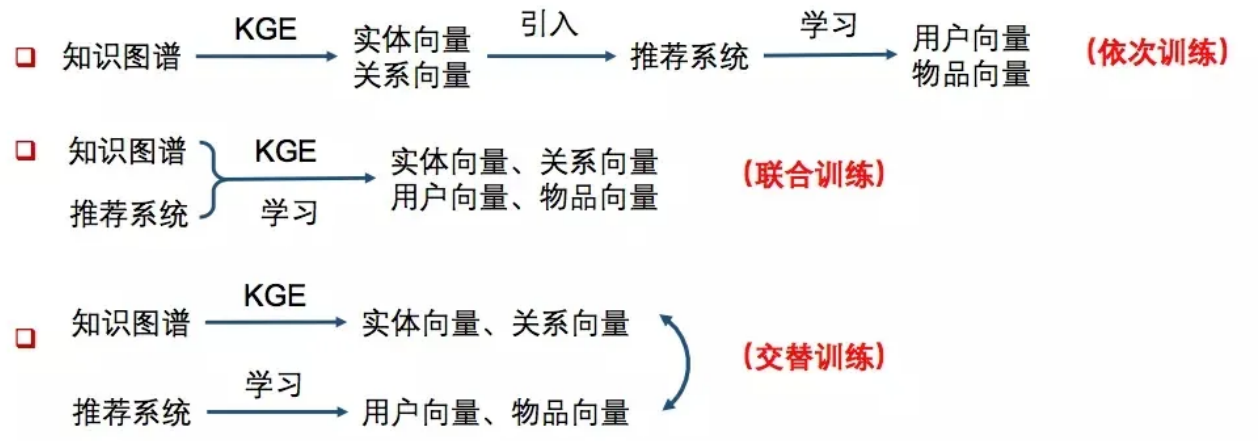
- **「精确性（precision）」**：为物品item引入了更多的语义关系，可以深层次地发现用户兴趣
- **「多样性（diversity）」**：提供了不同的关系连接种类，有利于推荐结果的发散，避免推荐结果局限于单一类型
- **「可解释性（explainability）」**：连接用户的历史记录和推荐结果，从而提高用户对推荐结果的满意度和接受度，增强用户对推荐系统的信任。

但是知识图谱难以与神经网络直接结合，所以引出了**「knowledge representation learning」**，通过学习entity和relation的embedding之后，再嵌入到神经网络中。

embedding方法主要可以分为「translational distance」方法和「semantic matching」方法两种，前者是学习从头实体到尾实体的空间关系变换（如TransE等系列），后者则是直接用神经网络对语义相似度进行计算。

将其结合到推荐里面比较困难的地方仍然有：

- 「图简化」 如何处理KG带来的多种实体和关系，按需要简化虽然可能会损失部分信息但对效率是必要的，如只对user-user或者item-item关系简化子图。
- 「多关系传播」KG的特点就是多关系，不过现有可以用attention来区分不同关系的重要性，为邻居加权。
- 「用户整合」将角色引入图结构，由于KG是外部信号，但是否也可以将用户也融入为一种实体变成内在产物呢？



一般使用知识图谱有三种模式，如上图：

- 「依次学习（one-by-one learning）」 使用知识图谱特征学习得到实体向量和关系向量，然后将这些低维向量（TransR方法等），引入推荐系统再做后面的处理。即只把知识图谱作为一个side info，多一维特征的处理方式。
- 「联合学习（joint learning）」 将知识图谱特征学习和推荐算法的目标函数结合，使用端到端（end-to-end）的方法进行联合学习。即把知识图谱的损失也纳入到最后的损失函数联合训练。
- 「交替学习（alternate learning）」 将知识图谱特征学习和推荐算法视为两个分离但又相关的任务，使用多任务学习（multi-task learning）的框架进行交替学习。这样可以让KG和RC在某种程度上融合的更加深入。

在介绍论文之前，先简要看看一般学习知识图谱的方法，一般有几种如下的处理方式：

- 「TransE」，即使其满足 h + r ≈ t，尾实体是头实体通过关系平移(翻译)得到的，但它不适合多对一和多对多，所以导致TransE在复杂关系上的表现差。公式如下

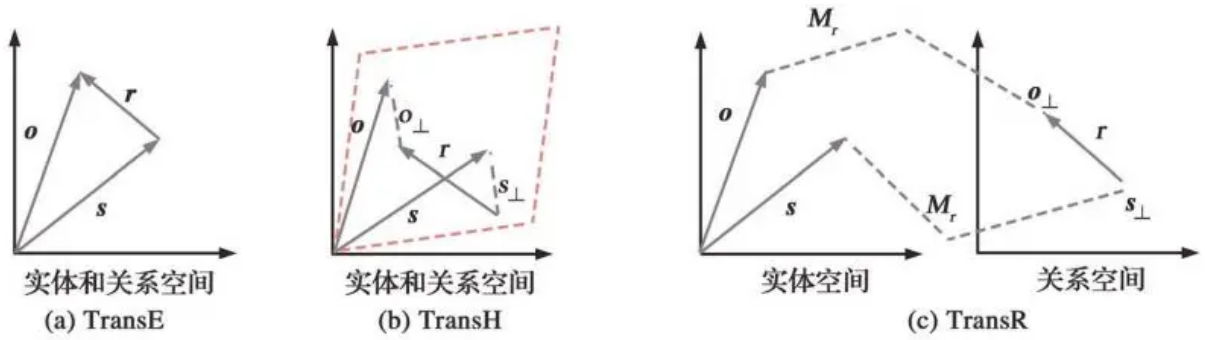
$$||h+r-t||_2^2$$

- **「TransH模型」**，即将实体投影到由关系构成的超平面上。值得注意的是它是非对称映射

$$||(h-w^Thw)+r-(t-w^Ttw)||_2^2$$

- **「TransR模型」**，该模型则认为实体和关系存在语义差异，它们应该在不同的语义空间。此外，不同的关系应该构成不同的语义空间，因此TransR通过关系投影矩阵，将实体空间转换到相应的关系空间。

$$||(hM_l+r-tM_t)||_2^2$$



- **「TransD模型」**，该模型认为头尾实体的属性通常有比较大的差异，因此它们应该拥有不同的关系投影矩阵。此外还考虑矩阵运算比较耗时，TransD将矩阵乘法改成了向量乘法，从而提升了运算速度。

- **「NTN模型」**，将每一个实体用其实体名称的词向量平均值来表示，可以共享相似实体名称中的文本信息。

接下来主要整理2篇论文，CKE和RippleNet。

CKE

> - 论文：Collaborative Knowledge base Embedding
> - 地址：https://www.kdd.org/kdd2016/papers/files/adf0066-zhangA.pdf
> - 也可以直接在公众号后台回复『0019』直接获取

发自16年KDD，将KG与CF融合做联合训练。

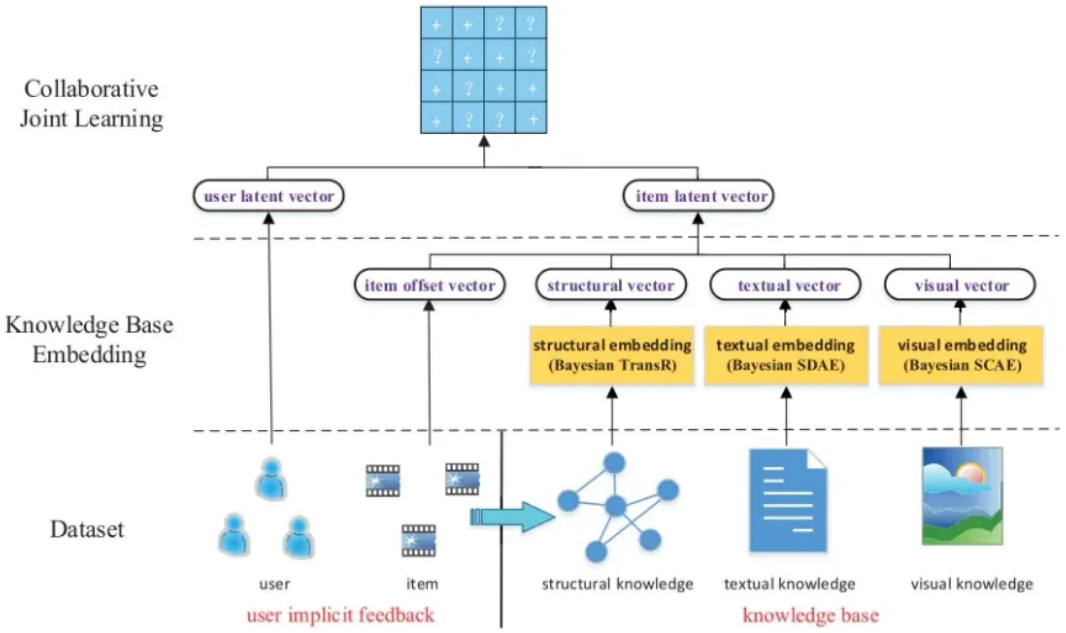


Figure 2: The flowchart of the proposed Collaborative Knowledge Base Embedding (CKE) framework for recommender systems

首先为了使用知识库，作者设计了三个组件分别从结构化知识，文本知识和视觉知识中提取语义特征，如上图中的右半部分，知识库的处理分别为：

结构化知识

知识库中的实体以及实体的联系。使用TransR提取物品的结构化信息（同时考虑nodes和relations），它的结构如下图，对于每个元组（h，r，t），首先将实体空间中的实体向关系r投影得到h_r和t_r，然后使$h_r + r \approx t_r$，能够使得头/尾实体在这个关系r下靠近彼此，使得不具有此关系r的实体彼此远离。

$$f_v(v_h, v_t) = ||v_h^r + r - v_t^r||_2^2$$

文本知识

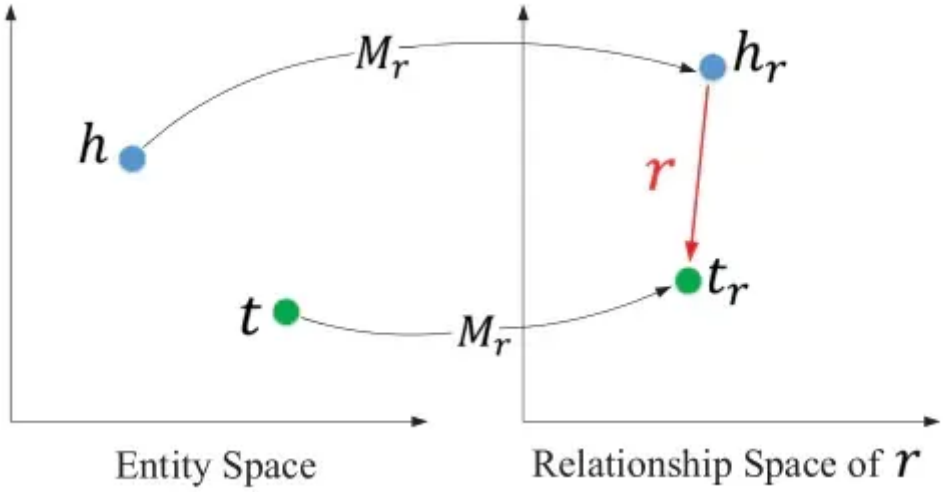
对实体的文字性描述。用多层降噪自编码器提取文本表达（SDAE），图中写的是Bayesian SDAE，意思就是让权重，偏置，输出层符合特定的正态分布，对视觉的处理也是一样的。

视觉知识

对实体的图片描述如海报等。用多层卷积自编码提取物品视觉表达（SCAE）

最后得到的item的表示为offset向量以及结构化知识，文本知识，图片知识的向量：

$$e_j = \eta_j + v_j + X_{\frac{L_2}{2},j*} + Z_{\frac{L_v}{2},j*}$$



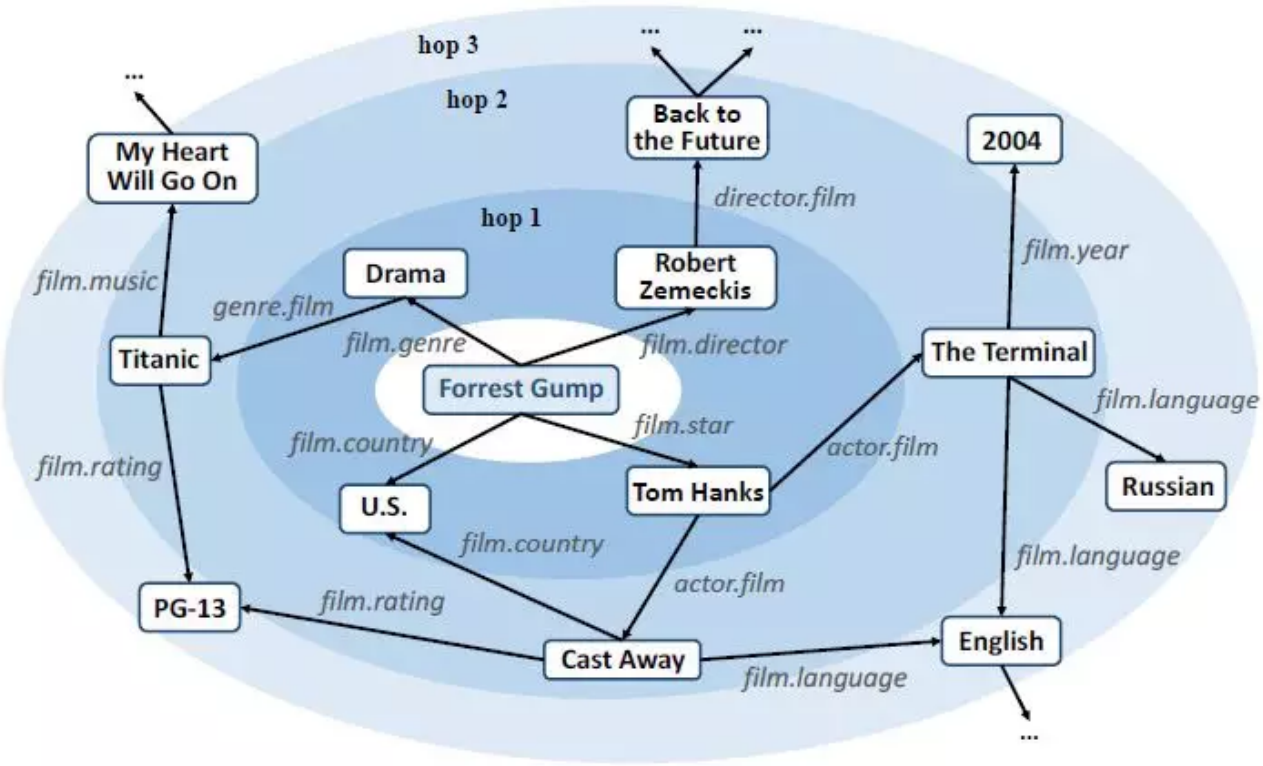
然后从知识库中提取的特征融合到collabrative filtering 中去，即与左边的用户反馈结合起来一起做CF进行训练就可以了，训练损失函数会用pair-wise的偏序优化。

```
#TransR
def projection_transR_pytorch(original, proj_matrix):
    ent_embedding_size = original.shape[1]
    rel_embedding_size = proj_matrix.shape[1] // ent_embedding_size
    original = original.view(-1, ent_embedding_size, 1)
    #借助一个投影矩阵就行
    proj_matrix = proj_matrix.view(-1, rel_embedding_size, ent_embedding_siz
    return torch.matmul(proj_matrix, original).view(-1, rel_embedding_size)
```

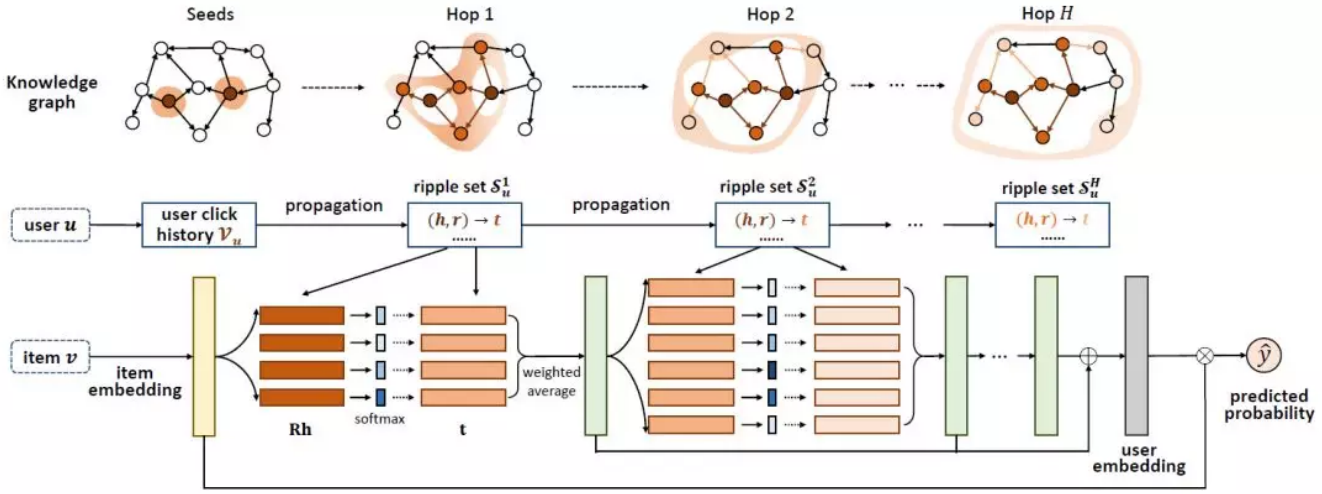
RippleNet

- 论文：RippleNet: Propagating User Preferences on the Knowledge Graph for Recommender Systems
- 地址：https://arxiv.org/abs/1803.03467
- 也可以在公众号后台回复『0020』直接获取

向来不同技术之间如果能融合的更深入，自然是能得到更好的信息。Ripple Network模拟了用户兴趣在知识图谱上的传播过程，整个过程类似于水波的传播，如上图从实体Forrest Gump开始一跳hop1，二跳hop2做传播，同时权重递减。



模型图如下图，对于给定的用户u和物品v，如何模拟用户兴趣在KG上的传播呢？



作者提出的方法就是将知识图谱中的每一个实体(h,r,t)都用用户历史的物品进行相似度计算：

$$p_i = softmax(v^T R_i h_i)$$

v是物品向量，r是关系，h是头节点，三者计算相似度（得到了图片中Rh后面的绿色方格）。然后用这个权重对该实体中的尾节点t加权就得到了第一跳/扩散的结果：

$$o_u^1 = \sum_{(h_i,r_i,t_i)\in S_u^1} p_i t_i$$

所有跳最后的用户特征为所有跳的总和，需要注意的是，Ripple Network中没有对用户直接使用向量进行刻画，而是用用户点击过的物品的向量集合作为其特征（代码中也可以只使用最后的o）：

$$u = o_u^1 + o_u^2 + \ldots + o_u^H$$

实际上求和得到的结果可以视为v在u的一跳相关实体中的一个响应。该过程可以重复在u的二跳、三跳相关实体中进行，如此，v在知识图谱上便以V为中心逐层向外扩散。最后再用用户特征计算对物品的相似度得到预测结果：

$$y = \sigma(u^T v)$$

然后来看一下模型类的代码：这部分的代码分为：数据input，得到嵌入特征，依次计算每一跳的结果并更新（按照公式依次计算），预测。最后是损失函数（由三部分组成）和训练、测试函数。

```
class RippleNet(object):
    def __init__(self, args, n_entity, n_relation):
        self._parse_args(args, n_entity, n_relation)
        self._build_inputs()
        self._build_embeddings()
        self._build_model()
        self._build_loss()
        self._build_train()

    def _parse_args(self, args, n_entity, n_relation):
        self.n_entity = n_entity
        self.n_relation = n_relation
        self.dim = args.dim
        self.n_hop = args.n_hop
        self.kge_weight = args.kge_weight
        self.l2_weight = args.l2_weight
        self.lr = args.lr
        self.n_memory = args.n_memory
        self.item_update_mode = args.item_update_mode
        self.using_all_hops = args.using_all_hops

    def _build_inputs(self):
        #输入有items id，labels和用户每一跳的ripple set记录
        self.items = tf.placeholder(dtype=tf.int32, shape=[None], name="item
        self.labels = tf.placeholder(dtype=tf.float64, shape=[None], name="l
```

```
        self.memories_h = []
        self.memories_r = []
        self.memories_t = []

        for hop in range(self.n_hop):#每一跳的结果
            self.memories_h.append(
                tf.placeholder(dtype=tf.int32, shape=[None, self.n_memory],
            self.memories_r.append(
                tf.placeholder(dtype=tf.int32, shape=[None, self.n_memory],
            self.memories_t.append(
                tf.placeholder(dtype=tf.int32, shape=[None, self.n_memory],

    def _build_embeddings(self):#得到嵌入
        self.entity_emb_matrix = tf.get_variable(name="entity_emb_matrix", d
                                                 shape=[self.n_entity, self.
                                                 initializer=tf.contrib.laye
        #relation连接head和tail所以维度是self.dim*self.dim
        self.relation_emb_matrix = tf.get_variable(name="relation_emb_matrix
                                                   shape=[self.n_relation, s
                                                   initializer=tf.contrib.la

    def _build_model(self):
        # transformation matrix for updating item embeddings at the end of e
        # 更新item嵌入的转换矩阵，这个不一定是必要的，可以使用直接替换或者加和策略。
        self.transform_matrix = tf.get_variable(name="transform_matrix", sha
                                                initializer=tf.contrib.layer

        # [batch size, dim]，得到item的嵌入
        self.item_embeddings = tf.nn.embedding_lookup(self.entity_emb_matrix

        self.h_emb_list = []
        self.r_emb_list = []
        self.t_emb_list = []
        for i in range(self.n_hop):#得到每一跳的实体，关系嵌入list
            # [batch size, n_memory, dim]
            self.h_emb_list.append(tf.nn.embedding_lookup(self.entity_emb_ma

            # [batch size, n_memory, dim, dim]
            self.r_emb_list.append(tf.nn.embedding_lookup(self.relation_emb_

            # [batch size, n_memory, dim]
            self.t_emb_list.append(tf.nn.embedding_lookup(self.entity_emb_ma

        #按公式计算每一跳的结果
        o_list = self._key_addressing()
```

```
        #得到分数
        self.scores = tf.squeeze(self.predict(self.item_embeddings, o_list))
        self.scores_normalized = tf.sigmoid(self.scores)

    def _key_addressing(self):#得到olist
        o_list = []
        for hop in range(self.n_hop):#依次计算每一跳
            # [batch_size, n_memory, dim, 1]
            h_expanded = tf.expand_dims(self.h_emb_list[hop], axis=3)

            # [batch_size, n_memory, dim]，计算Rh，使用matmul函数
            Rh = tf.squeeze(tf.matmul(self.r_emb_list[hop], h_expanded), axi

            # [batch_size, dim, 1]
            v = tf.expand_dims(self.item_embeddings, axis=2)

            # [batch_size, n_memory]，然后和v内积计算相似度
            probs = tf.squeeze(tf.matmul(Rh, v), axis=2)

            # [batch_size, n_memory]，softmax输出分数
            probs_normalized = tf.nn.softmax(probs)

            # [batch_size, n_memory, 1]
            probs_expanded = tf.expand_dims(probs_normalized, axis=2)

            # [batch_size, dim]，然后分配分数给尾节点得到o
            o = tf.reduce_sum(self.t_emb_list[hop] * probs_expanded, axis=1)

            #更新Embedding表，并且存好o
            self.item_embeddings = self.update_item_embedding(self.item_embe
            o_list.append(o)
        return o_list

    def update_item_embedding(self, item_embeddings, o):
        #计算完hop之后，更新item的Embedding操作，可以有多种策略
        if self.item_update_mode == "replace":#直接换
            item_embeddings = o
        elif self.item_update_mode == "plus":#加到一起
            item_embeddings = item_embeddings + o
        elif self.item_update_mode == "replace_transform":#用前面的转换矩阵
            item_embeddings = tf.matmul(o, self.transform_matrix)
        elif self.item_update_mode == "plus_transform":#用矩阵而且再加到一起
            item_embeddings = tf.matmul(item_embeddings + o, self.transform_
```

```
        else:
            raise Exception("Unknown item updating mode: " + self.item_updat
        return item_embeddings

    def predict(self, item_embeddings, o_list):
        y = o_list[-1]#1只用olist的最后一个向量
        if self.using_all_hops:#2或者使用所有向量的相加来代表user
            for i in range(self.n_hop - 1):
                y += o_list[i]

        # [batch_size], user和item算内积得到预测值
        scores = tf.reduce_sum(item_embeddings * y, axis=1)
        return scores

    def _build_loss(self):#损失函数有三部分
        #1用于推荐的对数损失函数
        self.base_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_log

        #2知识图谱表示的损失函数
        self.kge_loss = 0
        for hop in range(self.n_hop):
            h_expanded = tf.expand_dims(self.h_emb_list[hop], axis=2)
            t_expanded = tf.expand_dims(self.t_emb_list[hop], axis=3)
            hRt = tf.squeeze(tf.matmul(tf.matmul(h_expanded, self.r_emb_list
            self.kge_loss += tf.reduce_mean(tf.sigmoid(hRt))#为hRt的表示是否得
        self.kge_loss = -self.kge_weight * self.kge_loss

        #3正则化损失
        self.l2_loss = 0
        for hop in range(self.n_hop):
            self.l2_loss += tf.reduce_mean(tf.reduce_sum(self.h_emb_list[hop
            self.l2_loss += tf.reduce_mean(tf.reduce_sum(self.t_emb_list[hop
            self.l2_loss += tf.reduce_mean(tf.reduce_sum(self.r_emb_list[hop
            if self.item_update_mode == "replace nonlinear" or self.item_upd
                self.l2_loss += tf.nn.l2_loss(self.transform_matrix)
        self.l2_loss = self.l2_weight * self.l2_loss

        self.loss = self.base_loss + self.kge_loss + self.l2_loss #三者相加

    def _build_train(self):#使用adam优化
        self.optimizer = tf.train.AdamOptimizer(self.lr).minimize(self.loss)
        '''
        optimizer = tf.train.AdamOptimizer(self.lr)
```

```
        gradients, variables = zip(*optimizer.compute_gradients(self.loss))
        gradients = [None if gradient is None else tf.clip_by_norm(gradient,
                     for gradient in gradients]
        self.optimizer = optimizer.apply_gradients(zip(gradients, variables)
        '''

    def train(self, sess, feed_dict):#开始训练
        return sess.run([self.optimizer, self.loss], feed_dict)

    def eval(self, sess, feed_dict):#开始测试
        labels, scores = sess.run([self.labels, self.scores_normalized], fee
        #计算auc和acc
        auc = roc_auc_score(y_true=labels, y_score=scores)
        predictions = [1 if i >= 0.5 else 0 for i in scores]
        acc = np.mean(np.equal(predictions, labels))
        return auc, acc
```

完整的逐行中文注释笔记在：https://github.com/nakaizura/Source-Code-Notebook/tree/master/RippleNet

关于多跳的实现

博主在读文章的时候，始终不明白多跳是怎么实现的，下面我们看看代码是怎么写：

```
#ripple多跳时，每跳的结果集
def get_ripple_set(args, kg, user_history_dict):
    print('constructing ripple set ...')

    # user -> [(hop_0_heads, hop_0_relations, hop_0_tails), (hop_1_heads, ho
    ripple_set = collections.defaultdict(list)

    for user in user_history_dict:#对于每个用户
        for h in range(args.n_hop):#该用户的兴趣在KG多跳hop中
            memories_h = []
            memories_r = []
            memories_t = []

            if h == 0:#如果不传播，上一跳的结果就直接是该用户的历史记录
                tails_of_last_hop = user_history_dict[user]
```

```
            else:#去除上一跳的记录
                tails_of_last_hop = ripple_set[user][-1][2]

            #去除上一跳的三元组特征
            for entity in tails_of_last_hop:
                for tail_and_relation in kg[entity]:
                    memories_h.append(entity)
                    memories_r.append(tail_and_relation[1])
                    memories_t.append(tail_and_relation[0])

            # if the current ripple set of the given user is empty, we simpl
            # this won't happen for h = 0, because only the items that appea
            # this only happens on 154 users in Book-Crossing dataset (since
            if len(memories_h) == 0:
                ripple_set[user].append(ripple_set[user][-1])
            else:
                #为每个用户采样固定大小的邻居
                replace = len(memories_h) < args.n_memory
                indices = np.random.choice(len(memories_h), size=args.n_memo
                memories_h = [memories_h[i] for i in indices]
                memories_r = [memories_r[i] for i in indices]
                memories_t = [memories_t[i] for i in indices]
                ripple_set[user].append((memories_h, memories_r, memories_t)

    return ripple_set
```

即创造了一个ripple_set，这个set相当于就得到整个多跳应该访问到的节点，在每一跳里面都会为每个用户采样固定大小的邻居，然后存如到set中。所以在model模型的部分，可以直接遍历多跳计算。

一起交流

想和你一起学习进步！我们新建立了**『推荐系统、知识图谱、图神经网络』等** 专题讨论组，欢迎感兴趣的同学加入一起交流。为防止小广告造成信息骚扰，麻烦添加我的微信，手动邀请你（麻烦备注喔）