大数据分析第八次作业——知识图谱补全

【目的】

对知识图谱进行补全，预测出三元组中缺失的部分，从而使知识图谱变得更加完整.根据补全类型分类，使得三元组的部分得到补全，具体到 freebase 数据集中，就是补全每个人的出生等与自身关系的知识图谱。

【数据集】

Freebase15k

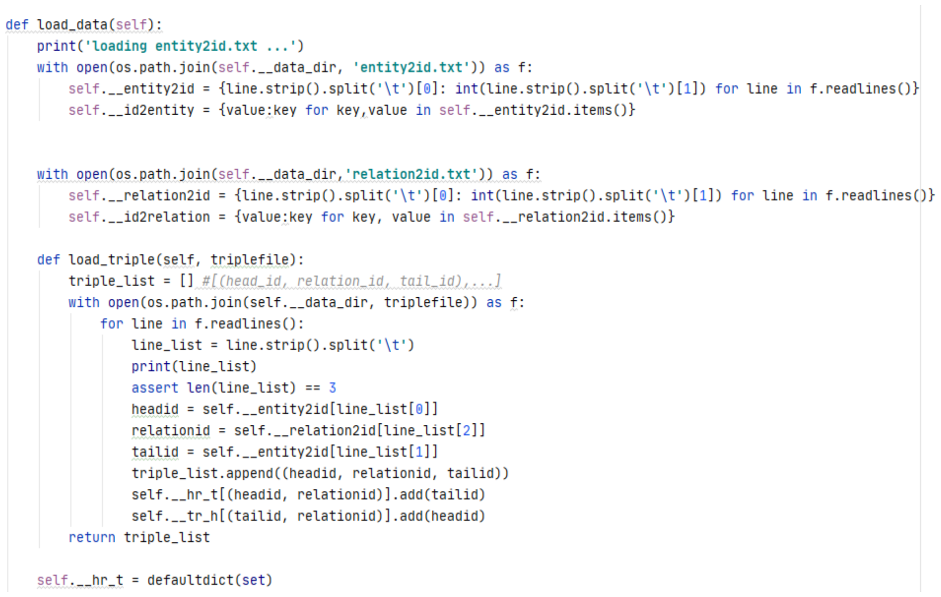
【使用方法】

使用 TransE 加 tensorflow 模拟神经网络来实现，TransE 算法，是 Bordes 等人 2013 年发表在 NIPS 上的文章提出的算法，根据我找到的 Bordes 的论文，TransE 方法，就是基于实体和关系的分布式向量表示，将每个三元组实例（head, relation, tail）中的关系 relation 看做从实体 head 到实体 tail 的翻译（其实我一直很纳闷为什么叫做 translating，其实就是向量相加），通过不断调整 h、r 和 t（head、relation 和 tail 的向量），使（h + r） 尽可能与 t 相等，即 h + r = t。基本原理：

$$f_r(h,t) = -\|\mathbf{h}_\perp + \mathbf{r} - \mathbf{t}_\perp\|_2^2.$$

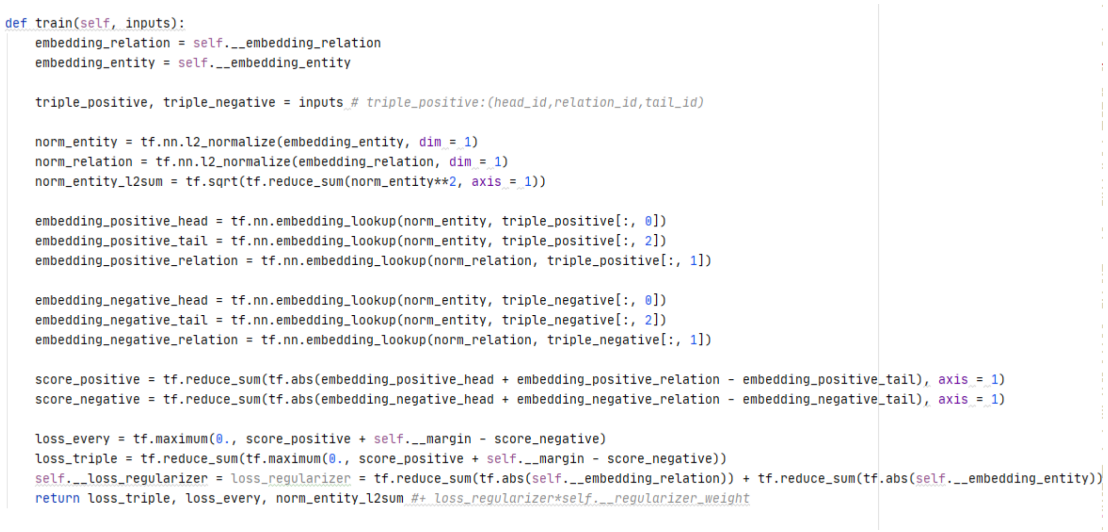
【算法细节】

1. 读取数据



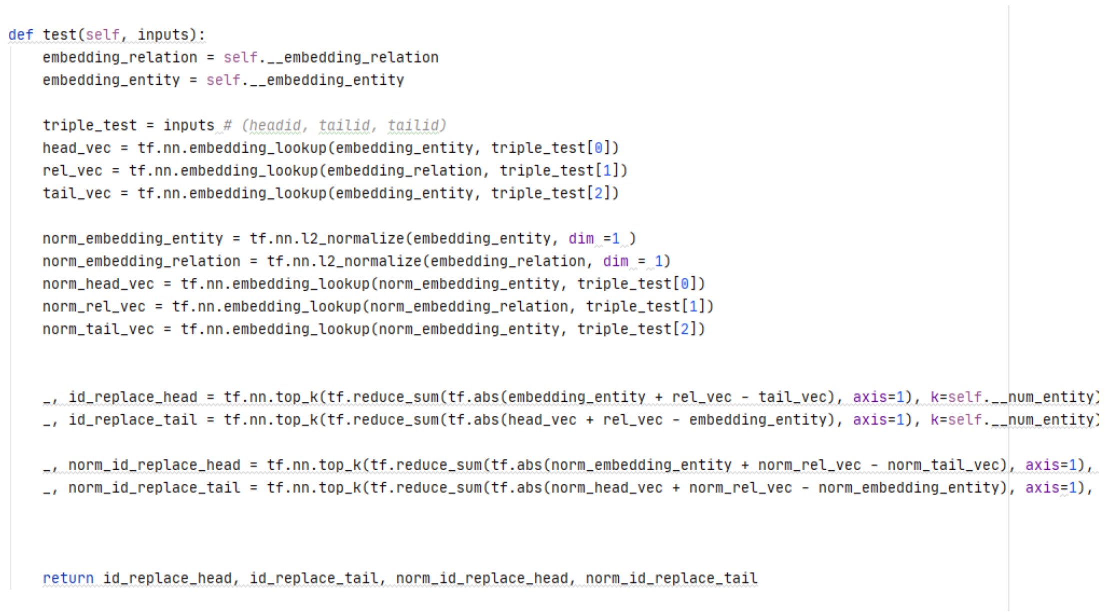
```
def load_data(self):
    print('loading entity2id.txt ...')
    with open(os.path.join(self.__data_dir, 'entity2id.txt')) as f:
        self.__entity2id = {line.strip().split('\t')[0]: int(line.strip().split('\t')[1]) for line in f.readlines()}
        self.__id2entity = {value:key for key,value in self.__entity2id.items()}


    with open(os.path.join(self.__data_dir,'relation2id.txt')) as f:
        self.__relation2id = {line.strip().split('\t')[0]: int(line.strip().split('\t')[1]) for line in f.readlines()}
        self.__id2relation = {value:key for key, value in self.__relation2id.items()}

    def load_triple(self, triplefile):
        triple_list = [] #[(head_id, relation_id, tail_id),...]
        with open(os.path.join(self.__data_dir, triplefile)) as f:
            for line in f.readlines():
                line_list = line.strip().split('\t')
                print(line_list)
                assert len(line_list) == 3
                headid = self.__entity2id[line_list[0]]
                relationid = self.__relation2id[line_list[2]]
                tailid = self.__entity2id[line_list[1]]
                triple_list.append((headid, relationid, tailid))
                self.__hr_t[(headid, relationid)].add(tailid)
                self.__tr_h[(tailid, relationid)].add(headid)
        return triple_list

    self.__hr_t = defaultdict(set)
```

2. 训练



```
def train(self, inputs):
    embedding_relation = self.__embedding_relation
    embedding_entity = self.__embedding_entity

    triple_positive, triple_negative = inputs # triple_positive:(head_id,relation_id,tail_id)

    norm_entity = tf.nn.l2_normalize(embedding_entity, dim = 1)
    norm_relation = tf.nn.l2_normalize(embedding_relation, dim = 1)
    norm_entity_l2sum = tf.sqrt(tf.reduce_sum(norm_entity**2, axis = 1))

    embedding_positive_head = tf.nn.embedding_lookup(norm_entity, triple_positive[:, 0])
    embedding_positive_tail = tf.nn.embedding_lookup(norm_entity, triple_positive[:, 2])
    embedding_positive_relation = tf.nn.embedding_lookup(norm_relation, triple_positive[:, 1])

    embedding_negative_head = tf.nn.embedding_lookup(norm_entity, triple_negative[:, 0])
    embedding_negative_tail = tf.nn.embedding_lookup(norm_entity, triple_negative[:, 2])
    embedding_negative_relation = tf.nn.embedding_lookup(norm_relation, triple_negative[:, 1])

    score_positive = tf.reduce_sum(tf.abs(embedding_positive_head + embedding_positive_relation - embedding_positive_tail), axis = 1)
    score_negative = tf.reduce_sum(tf.abs(embedding_negative_head + embedding_negative_relation - embedding_negative_tail), axis = 1)

    loss_every = tf.maximum(0., score_positive + self.__margin - score_negative)
    loss_triple = tf.reduce_sum(tf.maximum(0., score_positive + self.__margin - score_negative))
    self.__loss_regularizer = loss_regularizer = tf.reduce_sum(tf.abs(self.__embedding_relation)) + tf.reduce_sum(tf.abs(self.__embedding_entity))
    return loss_triple, loss_every, norm_entity_l2sum #+ loss_regularizer*self.__regularizer_weight
```

3. 测试



```
def test(self, inputs):
    embedding_relation = self.__embedding_relation
    embedding_entity = self.__embedding_entity

    triple_test = inputs # (headid, tailid, tailid)
    head_vec = tf.nn.embedding_lookup(embedding_entity, triple_test[0])
    rel_vec = tf.nn.embedding_lookup(embedding_relation, triple_test[1])
    tail_vec = tf.nn.embedding_lookup(embedding_entity, triple_test[2])

    norm_embedding_entity = tf.nn.l2_normalize(embedding_entity, dim =1 )
    norm_embedding_relation = tf.nn.l2_normalize(embedding_relation, dim = 1)
    norm_head_vec = tf.nn.embedding_lookup(norm_embedding_entity, triple_test[0])
    norm_rel_vec = tf.nn.embedding_lookup(norm_embedding_relation, triple_test[1])
    norm_tail_vec = tf.nn.embedding_lookup(norm_embedding_entity, triple_test[2])


    _, id_replace_head = tf.nn.top_k(tf.reduce_sum(tf.abs(embedding_entity + rel_vec - tail_vec), axis=1), k=self.__num_entity
    _, id_replace_tail = tf.nn.top_k(tf.reduce_sum(tf.abs(head_vec + rel_vec - embedding_entity), axis=1), k=self.__num_entity

    _, norm_id_replace_head = tf.nn.top_k(tf.reduce_sum(tf.abs(norm_embedding_entity + norm_rel_vec - norm_tail_vec), axis=1),
    _, norm_id_replace_tail = tf.nn.top_k(tf.reduce_sum(tf.abs(norm_head_vec + norm_rel_vec - norm_embedding_entity), axis=1),



    return id_replace_head, id_replace_tail, norm_id_replace_head, norm_id_replace_tail
```

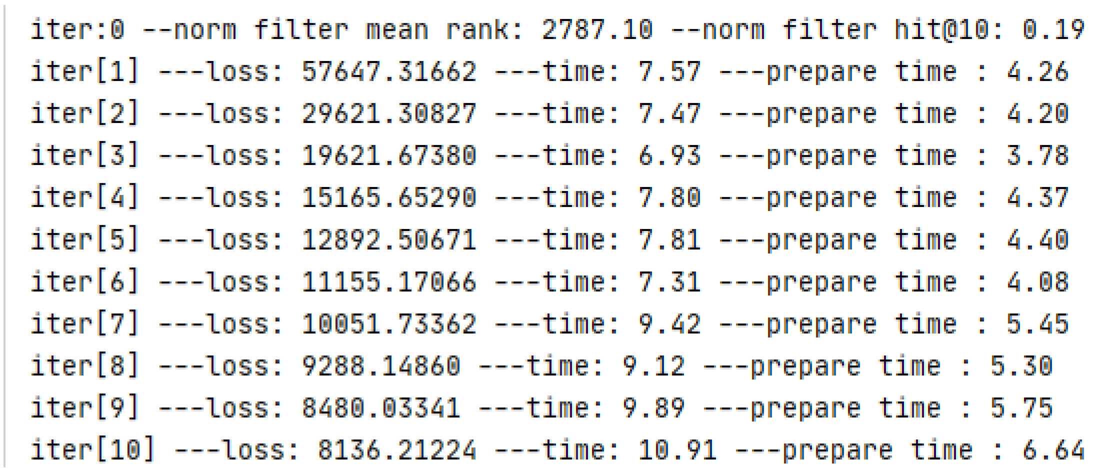
【指标】

在我的算法中，误差（loss）就是 SVM 中对应的误差模型，误差函数设计如下

$$\min \sum_{(h,r,t)\in S} \sum_{(h',r,t')\in S'} [\gamma + d(h+r,t) - d(h'+r,t')]_+$$

，另外具有 hit 与 meanrank，hit：即后面的数字个个体中能否遇到真正的实体，meanrank 即平均计算到多少个才能遇到正确的实体。

【结论】



```
iter:0 --norm filter mean rank: 2787.10 --norm filter hit@10: 0.19
iter[1] ---loss: 57647.31662 ---time: 7.57 ---prepare time : 4.26
iter[2] ---loss: 29621.30827 ---time: 7.47 ---prepare time : 4.20
iter[3] ---loss: 19621.67380 ---time: 6.93 ---prepare time : 3.78
iter[4] ---loss: 15165.65290 ---time: 7.80 ---prepare time : 4.37
iter[5] ---loss: 12892.50671 ---time: 7.81 ---prepare time : 4.40
iter[6] ---loss: 11155.17066 ---time: 7.31 ---prepare time : 4.08
iter[7] ---loss: 10051.73362 ---time: 9.42 ---prepare time : 5.45
iter[8] ---loss: 9288.14860 ---time: 9.12 ---prepare time : 5.30
iter[9] ---loss: 8480.03341 ---time: 9.89 ---prepare time : 5.75
iter[10] ---loss: 8136.21224 ---time: 10.91 ---prepare time : 6.64
```

如图所示为一个 iter 的运行结果数据，为每一次迭代的数据。