实验报告

1. 引言

查询规模估计是数据库管理系统进行查询处理的一个重要步骤，也是数据库管理系统进行查询优化的重要参考，对于数据库合理高效地完成查询操作具有非常重要的意义。

在本次任务之中，给定一个与电影和影评有关的数据库，数据库中总共有6张表，分别是：

title表，总共有id,kind_id,production_year三个属性

movie_companies表，总共有id,company_id,movie_id三个属性

cast_info表,总共有id,movie_id,person_id,role_id四个属性

movie_info表，总共有id,movie_id,info_type_id三个属性

movie_info_idx表，总共有id,movie_id,info_type_id三个属性

movie_keyword表，总共有id,movie_id,keyword_id三个属性

现在有若干基于这些表的合法查询语句，这些语句只有select，from，where，and关键字，没有分组，排序，聚集函数等操作，并且规定from之中最多选择两张表，where之中最多使用一次连接操作。其中有100000条sql语句已经知道了最终的查询结果的规模（符合要求的元组数量），在training_data.csv之中给出，作为训练集，剩余5000条sql语句未知查询结果，在testing_data.csv之中给出了这些sql语句。要求通过SVM，CNN，线性回归等机器学习或者深度学习算法，通过训练获得模型，并利用这个模型预测得出剩余5000条sql语句的查询结果规模。

我们使用了两种方法：MSCN和XGBoost，分别在本文的第三部分和第四部分进行详细介绍。最终在测试集上的MSLE成绩为：

MSCN：1.15

XGBoost： 0.77

2. 文献调研

本次Cardinality Estimation大作业中，我们组工作整体上复现了 Efficiently Approximating Selectivity Functions using Low Overhead Regression Models，使用低开销的监督学习回归模型来尝试解决这一问题。

我们首先仔细研读了本次大作业的参考文献Efficiently Approximating Selectivity Functions using Low Overhead Regression Models，这篇论文的主要工作和贡献如下：

- 提出IterTrain算法，跟据训练的要求来估计所需要的训练集规模的大小，并迭代地生成训练数据集直至满足训练要求。具体做法是，在每一轮迭代中，使用十折交叉验证来训练并估计模型准确率的95%的置信区间，如果达不到训练要求则一点一点增加训练集。
- 为了进一步减少获得训练集的标签所需要的代价开销，提出ApproxLabel算法，并不使用数据库中全部的数据进行查询，而是仅仅抽取了一个小的sample，在这个样本之中进行查询，并使用小样本中的查询结果来估计在数据库中查询的总规模。
- 建立模型，对于查询表达式语句，尤其是包含有join连接运算的SQL查询语句，可以较为精确地估计其查询结果。由于我们此次的Cardinality Estimation 问题中我们所拥有的数据只有sql语句，数据库的query plan以及最终的查询规模，因此整体上是query based问题，而非data based问题，因此我们复现的主要是这一部分的工作，对于低成本地生成训练数据没有太多涉及。

3. 方法一：Multi Set Convolution Network

3.1 算法思路

首先通过pandas设置'#'作为分隔符将数据读取出来，然后将以'#'分割的每一列再次使用','作为分隔符来进一步分割，分割完成的结果是每一个query查询语句都转化为表名、连接条件、选择条件三个list，list中的每一个元素都是以','分割的字符串，将他们连接起来就是完整的SQL语句表达。

所有的选择条件其实内部的模式都是固定的，可以统一表达为如下模式：.而整个选择条件就是由若干组这样的三元组所构成的，因此我们可以从选择条件中以三个为一组，抽取出比较运算符和属性名。

统计所有的属性名的种类数和所有运算符的种类数，发现属性名总共有id,movie_id,kind_id,company_id等11种属性，所以使用一个长度为11维的01向量来表达一个属性，某一个属性被编码到第几位，独热向量中的第几位就是1，其余位是0。对于运算符，总共只有大于、小于、等于三种类型，使用一个长度为3的01向量进行独热表达即可，表名同理。

经过以上操作以后，每一个SQL语句都可以使用三个list来表达，一个list是table,将涉及到的一个或者多个表使用独热表达并连接起来；一个list是join,将涉及到的最多一个多表连接条件使用独热表达；还有一个list是predict，将涉及到的一个或者多个选择条件进行表达。但是这样做存在的一个问题是，有的SQL查询语句可能涉及到了多个表和多个选择条件，而有的SQL查询语句可能只涉及到了一个表和一个选择条件，这就使得所有表的向量化表达长度是不相同的，为此，使用zero-padding，全部按照最长的向量用零进行补全，并转化为tensor张量，方便后续模型使用。

在选择模型时，我们的思考是，既然一个SQL查询语句由许多不同的部分来组成，这三部分将相互作用，共同决定这个SQL语句最终得到的查询规模，那么我们应当针对这三个部分，分别采用一种小的模型来进行处理，最终再将结果进行汇总。

基于此，我们决定首先尝试使用深度学习领域最基础的模型，多层感知机（MLP），来作为我们训练一个SQL语句的表名、连接条件、选择条件时所使用的小模型，在一轮迭代结束之后，将多层感知机训练这三个部分得到的结果进行拼接起来，作为下一轮训练的输入，也就是说，三个部分分别是由MLP进行模型参数的训练，来尽可能地捕捉捕捉到一个SQL查询语句在这三个不同层面上的特征。统一使用线性激活函数ReLU作为每一个中间隐藏层节点单元的激活函数，在最终输出时使用sigmoid函数进行转换。

在训练时，我们最初打算使用和评测标准相同MSLE来做我们的误差函数，但是通过查阅相关的资料和文献，发现这样做并不是最优的选择， *Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors*一文中提出使用q-error来作为查询规模估计类问题时的误差函数，因为跟据q-error，可以界定预测结果的上界和下界，以及预测结果和实际结果之间的比值。另外，*On the Propagation of Errors in the Size of Join Results* 一文之中提到，每有一次join的操作，都有可能会使得查询估计的误差指数级的增长，所以我们认为，由于训练集和测试集都有join的存在，所以使用q-error来作为误差可能更加合适。

训练时，我们通过各种调参和尝试，确定了batch_size为1024，一次处理1024个训练querry作为一个批次，使用Adam进行网络内部参数的调优，隐藏单元的数量设置为256，迭代了50轮，使用pytorch自带的train模式，每一轮训练之后进行batch normalization以及drop一些单元，从而尽可能地避免过拟合现象发生。

3.2 实现步骤

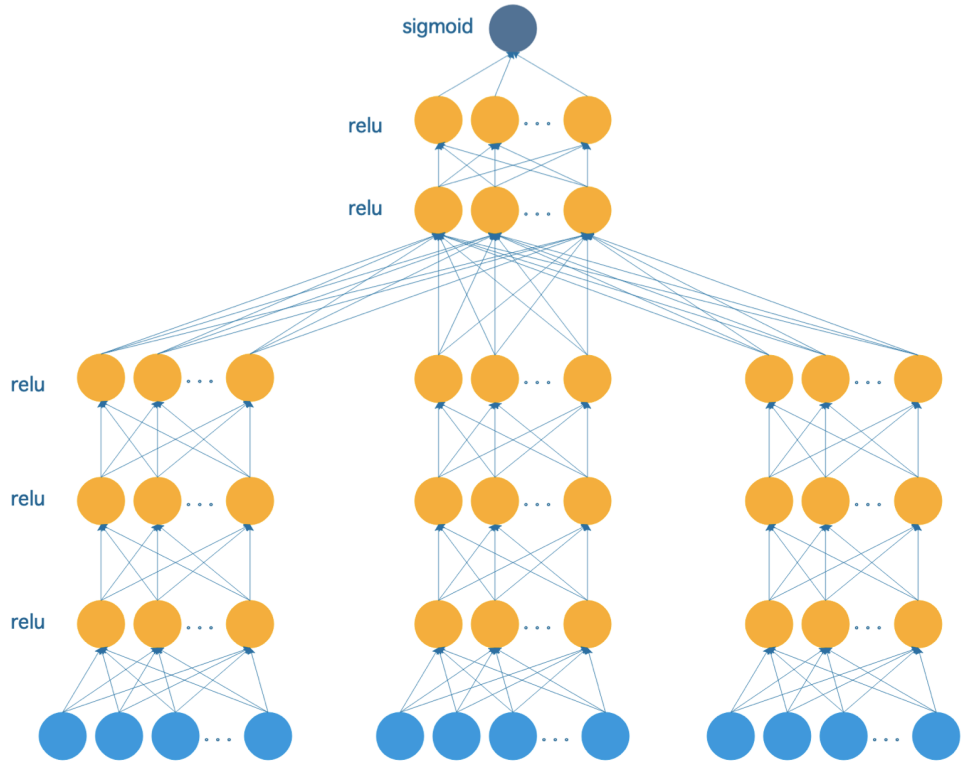
将table-set、predicates、join-set抽取独热特征后，组成三个特征向量，每个特征向量连接一组hidden_units，隐藏层节点数量为超参数；然后再次连接一组相同数量的hidden_units，这些hidden_units与另一组相同数量的hidden_units相连。

依此法生成的三个特征向量对应的三组hidden_units与另一组hidden_units相连，与前文所述相同，这一组hidden_units连接数量相同的一组hidden_units，最后的这一组hidden_units连接到一个输出层节点上。

我们进行数据预处理的核心代码如下，从csv文件之中读取SQL查询语句，经过编码，归一化和填充、张量化之后得到每一个SQL语句的特征向量：

```
def make_dataset(samples, predicates, joins, labels, max_num_joins,
max_num_predicates):
    """Add zero-padding and wrap as tensor dataset."""

    sample_masks = []
    sample_tensors = []
    for sample in samples:
        sample_tensor = np.vstack(sample)
        num_pad = max_num_joins + 1 - sample_tensor.shape[0]
        sample_mask = np.ones_like(sample_tensor).mean(1, keepdims=True)
        sample_tensor = np.pad(sample_tensor, ((0, num_pad), (0, 0)),
'constant')
        sample_mask = np.pad(sample_mask, ((0, num_pad), (0, 0)), 'constant')
        sample_tensors.append(np.expand_dims(sample_tensor, 0))
        sample_masks.append(np.expand_dims(sample_mask, 0))
    sample_tensors = np.vstack(sample_tensors)
    sample_tensors = torch.FloatTensor(sample_tensors)
    sample_masks = np.vstack(sample_masks)
    sample_masks = torch.FloatTensor(sample_masks)

    predicate_masks = []
    predicate_tensors = []
    for predicate in predicates:
        predicate_tensor = np.vstack(predicate)
        num_pad = max_num_predicates - predicate_tensor.shape[0]
        predicate_mask = np.ones_like(predicate_tensor).mean(1, keepdims=True)
        predicate_tensor = np.pad(predicate_tensor, ((0, num_pad), (0, 0)),
'constant')
        predicate_mask = np.pad(predicate_mask, ((0, num_pad), (0, 0)),
'constant')
        predicate_tensors.append(np.expand_dims(predicate_tensor, 0))
        predicate_masks.append(np.expand_dims(predicate_mask, 0))
    predicate_tensors = np.vstack(predicate_tensors)
    predicate_tensors = torch.FloatTensor(predicate_tensors)
    predicate_masks = np.vstack(predicate_masks)
    predicate_masks = torch.FloatTensor(predicate_masks)

    join_masks = []
    join_tensors = []
    for join in joins:
        join_tensor = np.vstack(join)
        num_pad = max_num_joins - join_tensor.shape[0]
        join_mask = np.ones_like(join_tensor).mean(1, keepdims=True)
        join_tensor = np.pad(join_tensor, ((0, num_pad), (0, 0)), 'constant')
        join_mask = np.pad(join_mask, ((0, num_pad), (0, 0)), 'constant')
        join_tensors.append(np.expand_dims(join_tensor, 0))
        join_masks.append(np.expand_dims(join_mask, 0))
    join_tensors = np.vstack(join_tensors)
    join_tensors = torch.FloatTensor(join_tensors)
    join_masks = np.vstack(join_masks)
    join_masks = torch.FloatTensor(join_masks)

    target_tensor = torch.FloatTensor(labels)

    return dataset.TensorDataset(sample_tensors, predicate_tensors,
join_tensors, target_tensor, sample_masks,
                                 predicate_masks, join_masks)
```

网络结构示意图如下所示：



这样，每次在train_set中划分的30%大小的validation_set中得到的误差，都会在该网络反向传播，更新每个权重。在一定次数的迭代后，将训练好的模型代入test_set，并提交查看MSLE。

```
class SetConv(nn.Module):
    def __init__(self, sample_feats, predicate_feats, join_feats, hid_units):
        super(SetConv, self).__init__()
        self.sample_mlp1 = nn.Linear(sample_feats, hid_units)
        self.sample_mlp2 = nn.Linear(hid_units, hid_units)
        self.sample_mlp3 = nn.Linear(hid_units, hid_units)####1.7
        self.predicate_mlp1 = nn.Linear(predicate_feats, hid_units)
        self.predicate_mlp2 = nn.Linear(hid_units, hid_units)
        self.predicate_mlp3 = nn.Linear(hid_units, hid_units)####1.7
        self.join_mlp1 = nn.Linear(join_feats, hid_units)
        self.join_mlp2 = nn.Linear(hid_units, hid_units)
        self.join_mlp3 = nn.Linear(hid_units, hid_units)####1.7
        self.out_mlp1 = nn.Linear(hid_units * 3, hid_units)
        #self.out_mlp2 = nn.Linear(hid_units, 1)
        self.out_mlp2 = nn.Linear(hid_units, hid_units)####1.7
        self.out_mlp3 = nn.Linear(hid_units, 1)####1.7

    def forward(self, samples, predicates, joins, sample_mask, predicate_mask,
join_mask):
        hid_sample = F.relu(self.sample_mlp1(samples))
        hid_sample = F.relu(self.sample_mlp2(hid_sample))
        hid_sample = F.relu(self.sample_mlp3(hid_sample))####1.7
        hid_sample = hid_sample * sample_mask  # Mask
        hid_sample = torch.sum(hid_sample, dim=1, keepdim=False)
        sample_norm = sample_mask.sum(1, keepdim=False)
        hid_sample = hid_sample / sample_norm

        hid_predicate = F.relu(self.predicate_mlp1(predicates))
```

```
        hid_predicate = F.relu(self.predicate_mlp2(hid_predicate))
        hid_predicate = F.relu(self.predicate_mlp3(hid_predicate))####1.7
        hid_predicate = hid_predicate * predicate_mask
        hid_predicate = torch.sum(hid_predicate, dim=1, keepdim=False)
        predicate_norm = predicate_mask.sum(1, keepdim=False)
        hid_predicate = hid_predicate / predicate_norm

        hid_join = F.relu(self.join_mlp1(joins))
        hid_join = F.relu(self.join_mlp2(hid_join))
        hid_join = F.relu(self.join_mlp3(hid_join))####1.7
        hid_join = hid_join * join_mask
        hid_join = torch.sum(hid_join, dim=1, keepdim=False)
        join_norm = join_mask.sum(1, keepdim=False)
        hid_join = hid_join / join_norm

        hid = torch.cat((hid_sample, hid_predicate, hid_join), 1)
        hid = F.relu(self.out_mlp1(hid))
        hid = F.relu(self.out_mlp2(hid))####1.7
        #out = torch.sigmoid(self.out_mlp2(hid))
        out = torch.sigmoid(self.out_mlp3(hid))####1.7
        return out
```

通过调节epoch循环次数，num_units隐藏层节点个数，得到的测试结果如下：

MSLE	128	300	256	512	num_units
15	2.028				
30		1.4528	1.7498		
60			1.1502	1.422	
10			1.4444		
100			1.284		
50			1.3378		
70			1.7		
61			1.5498		
40			1.5057		
num_epochs					

在设置60次循环，每层隐藏层256个节点的超参数后，得到的MSLE最小，为1.15

4. 方法二： XGBoost Regression

4.1 实现思路

首先对数据进行预处理，其中最主要也最为关键的一步就是将SQL查询语句进行向量化表达，以作为模型的输入。通过观察column_min_max_vals.csv文件可以发现，本次任务的所有查询语句总共涉及到6张表的20个属性，这20个属性中，有16个属性可以看作是数值型数据，有4个属性可以看作是类别型数据，如t.kind_id，mc.company_type，ci.role_id，mi_idx.info_type_id。同时观察所给的SQL语句可以发现，所有的SQL语句都遵循模式：

```
SELECT * FROM <tableset> WHERE <join> AND <predicates>
```

因此，我们的向量化工作是针对table set, join 和 predicates 三个部分分别进行向量化，然后三个向量进行连接，最终得到整个SQL语句的向量化表达。

对于table set的向量化工作相对比较简单，因为总共只有六张表，所以可以使用一个长度为6的01向量来表达$[table_1, table_2, .., table_6]$，如果第$i$张表出现在FROM后面的tableset的话，就将对应的$table_i$设置为1，否则就设置为0。

对于join条件的表达同样比较简单，由于本次任务规定最多只能够出现1个join连接条件，所以只需要搜索找出所有在数据集之中出现过的连接情况，对这些连接类型按照01独热表达进行编码即可，并且在最终的向量中最多只会有一个1出现。

重点在于对predicates条件的编码和向量化，这是最为复杂的部分，需要分数值型的属性和类别型的属性两种情况进行讨论。

- 对于每一个数值型的属性k，使用一个区间$[lb_k, ub_k]$来表示一个SQL查询所要求的该属性的上界和下界，并满足条件$lb_k \geq min_k$以及$ub_k \leq max_k$，其中min_k和max_k分别表示属性k在数据库中存在的最小值和最大值。如果该SQL语句并没有对这个属性进行限制，那么使用最大值和最小值分别作为上界和下界，也就是表示成$[min_k, max_k]$.
- 对于每一个类别型的属性k，假设该属性总共具有N个取值，使用独热表达$[catg_1^k, catg_2^k, catg_3^k, \dots, catg_N^k]$来表示一个SQL查询语句对该属性的要求，如果允许这个类型属性取到第i个值，那么$catg_i^k = 1$，否则就有$catg_i^k = 0$。如果这个SQL查询语句并没有对这个属性进行限制，那么就有$catg_i^k = 1, i = 1, 2, 3, \dots, N$，也就是一个全部是1的序列。
- 对于所有总共20个属性，逐个讨论每一个属性是否在predicates之中有约束限制，并根据其类型进行编码，每一个数值型变量的向量长度是2，每一个类别型变量的向量长度是其类别总数，将每一个属性的向量表达进行拼接操作，就得到整个predicates部分的

按照上述方法，按照table set，join和predicates三个类别分别进行编码，得到三个向量。将这三个向量进行拼接之后就可以得到最终整个SQL语句的向量化的结果。

之后，我们为了可以有效利用每一个SQL语句的query plan，我们从每一个query plan之中的第一行抽取了rows的值作为其中的一个特征，因此，对于每一个SQL查询语句，我们最终得到了以下四个特征：

- 表编码
- 连接条件编码
- 选择条件编码
- query plan之中预测值的对数

得到向量化结果之后，我们发现，对于数值型的属性，他们的取值范围的差异非常之大，导致整个向量化表达是“不均衡的”，因此我们使用了一个常规技巧——归一化。利用column_min_max_vals.csv之中的各个属性的最大值和最小值，将上界和下界进行归一化，具体而言就是进行可逆变换：

$$lb'_k = \frac{lb_k - min_k}{max_k - min_k}$$
$$ub'_k = \frac{ub_k - min_k}{max_k - min_k}$$

变换之后，所有的上下界全部都在0和1之间。此外，我们借鉴论文之中的做法，对标签进行对数化，也就是

$$label' = log(label + 1)$$

这样做是因为，原有的label也就是查询规模变化范围非常大，难以预测；评价指标是MSLE，同样需要取对数运算，更加贴近评价标准。

之后，我们使用XGBoost作为回归拟合的工具，进行回归。XGBoost使用的超参数为$\{tree_method : hist, grow_policy : lossguide\}$.回归得到的是预测值加一之后取对数的值，然后通过变换得到原本的预测结果。

4.2 实现步骤

首先，我们需要提取query plan之中对最终查询结果的元组数量的估计值。这个步骤我们是通过pyhon中的正则表达式相关函数来完成的，相关代码如下：

```
pattern=r'rows=(\d*)'
planrows=[]
for i in range(100000):
    with open(Config.train_queryplan_dir+str(i)+'.txt',mode='r') as f:
        line=f.readline()
        match=re.search(pattern,line)
        if match:
            planrows.append(int(match.group(1)))
            else:
                planrows.append(-1)
```

正则表达式提取模式为：'rows=(\d*)'.通过这个模式可以精确匹配到query plan第一行之中的行数，并将所有query plan中的行数记录在中间文件之中，以备之后使用。

接下来，我们着手对SQL语句中接在FROM之后的表名，以及两表连接条件和选择约束条件进行编码。其中对和的编码是较为简单和直接的，因为无需对他们进行进一步的解析，只需要按照在SQL语句中的字面值进行处理。为了统一地进行这种对有限大小的集合中元素进行编码的工作，尽可能地减少代码复写率，我们专门编写了一组函数idx_to_onehot()和set_encode()以实现此功能：

```
def idx_to_onehot(idx, num_elements):
    onehot=[0]*num_elements
    onehot[idx] = 1.
    return onehot

def set_encode(source_set, onehot=True):
    num_elements = len(source_set)
    source_list = list(source_set)
    source_list.sort()
    thing2idx = {s: i for i, s in enumerate(source_list)}
    idx2thing = [s for i, s in enumerate(source_list)]
    if onehot:
        thing2vec = {s: idx_to_onehot(i, num_elements) for i, s in
enumerate(source_list)}
        return thing2vec, idx2thing
    return thing2idx, idx2thing
```

对集合的编码方式统一为独热编码。对和编码的一点小区别是，我们的题目要求限制了最多只有两表连接，最多只有一个连接条件，所以要么为空（这种情况下会被编码为全零），要么为一个等式的连接条件。而则最多会有两个，因此会对这两个表分别进行编码，并且编码所得到的向量会进行向量加法，而不是级联，以保证最终所有SQL语句的向量表达长度是相同的。

对于的编码是最为复杂的，因为不能简单地对字面值进行编码，需要对条件进行解析，正如上文之中所提到的，编码的目的是对20个属性中分成数值型和类别型分别得到其允许的取值范围。

我们首先构造初始向量，也就是在一条SQL语句没有对属性做任何限制的情况下得到的编码，数值型变量的上下界全部用最大值和最小值暂时替代，类别型变量的所有类别全部取值为1，代码如下所示：

```
def vector_init():
    min_max=pd.read_csv(Config.minmax_path)
    min_val=list(min_max['min'])
    max_val=list(min_max['max'])
    column=list(min_max['name'])
    col2i=dict()
    j=0
    for i in range(len(column)):
        if column[i] not in Domain.excludelist:
            col2i[column[i]]=j*2
            j+=1
    j*=2
    col2i['t.kind_id']=j
    j+=6
    col2i['mc.company_type_id']=j
    j+=2
    col2i['ci.role_id']=j
    j+=11
    col2i['mi_idx.info_type_id']=j
    j+=5
    orig=[0]*56
    for i in range(len(column)):
        if column[i] not in Domain.excludelist:
            orig[col2i[column[i]]]=min_val[i]
            orig[col2i[column[i]]+1]=max_val[i]
    for j in range(len(orig)):
        if orig[j]==0:
            orig[j]=1
    return column, col2i, orig, min_val, max_val
```

返回值中，col2i是一个字典，将一个属性名映射到这个这个属性在向量编码中开始的第一位的下标，之后解析每条语句的选择条件时需要使用。

对于每一个SQL语句，使用python中自带的split函数，将其部分以逗号作为切割符进行字符串切割，得到一个个单独的'语素'。由于每一个条件都具有 的形式，所以将零散的'语速'每三个为一组构成一个语义完整的、以列表形式呈现的子条件。每一个子条件都是一个列表，列表中有三项，分别是和分组使用的函数chunk()如下：

```
def chunk(l, n):
    for i in range(0, len(l), n):
        yield l[i:i + n]
```

然后针对每一个子条件，抽取字段，判断其是否为数值型的变量，如果是，利用col2i找到这个属性在编码向量中应当对应的位置，跟据 是大于、小于和等于，使用 来更新下界或者上界；如果不是，跟据事先确定的valist找到这个类别在编码上的每一位对应的哪一个类别值，依次判断这个类别值是否符合当前子条件的要求，具体代码如下。

```
    condition_encodes=[]
    for i in range(len(conditions)):
        temp=orig.copy()
        onesql_condition=conditions[i]
        if onesql_condition==[['']]:
            condition_encodes.append(temp)
            continue
        for condition in onesql_condition:
            op=condition[1]
            val=int(condition[2])
            if condition[0] not in Domain.excludelist:
                minval_index=col2i[condition[0]]
                maxval_index=minval_index+1
                if op =='>':
                    temp[minval_index]=val
                if op=='<':
                    temp[maxval_index]=val
                if op=='=':
                    temp[minval_index]=val
                    temp[maxval_index]=val
            elif condition[0] in Domain.excludelist:
                start_index=col2i[condition[0]]
                valist=Domain.valists[condition[0]]
                if op=='>':
                    for j in range(len(valist)):
                        if valist[j]<=val:
                            temp[start_index+j]=0
                if op=='<':
                    for j in range(len(valist)):
                        if valist[j]>=val:
                            temp[start_index+j]=0
                if op=='=':
                    for j in range(len(valist)):
                        if valist[j]!=val:
                            temp[start_index+j]=0
        condition_encodes.append(temp)
```

最后需要对于每一个数值型的上下界跟据其最大值和最小值进行归一化。

向量的四个部分完成计算后级联，使用函数train_predict()进行训练和预测。首先使用math.log对训练集之中所有的label进行取对数操作，训练了300个轮次，并将预测完成的结果进行指数运算，得到真正的预测结果，并将最终结果写入到指定目标文件之中。

```
def train_predict(train_sqlvec, test_sqlvec, labels):
    loglabels=[]
    for i in range(len(labels)):
        loglabels.append(math.log(labels[i]+1))
    dtrain = xgb.DMatrix(np.array(train_sqlvec), label=loglabels)
    param={'tree_method':'hist', 'grow_policy':'lossguide'}
    num_round=300
```

```
    bst = xgb.train(param, dtrain, num_round)
    dtest = xgb.DMatrix(test_sqlvec)
    logpred = bst.predict(dtest)
    pred=np.exp(logpred)-1
    with open(Config.result_path,mode='w') as f:
        f.write('Query ID'+','+'Predicted Cardinality'+'\n')
        for i in range(len(pred)):
            f.write(str(i) + ',' + str(int(pred[i])) + '\n')
    return pred
```

在迭代了300轮之后，得到的最终测试集评测结果MSLE为0.77