# 几何特征

## 几何编码矩阵的建立：

对每个实体建立一个边框，统计重叠数量，考虑10类实体，一张图就是10个节点，每个节点算一个实体类，每个节点维度：1（该实体类，剩余不重叠的数量）+10（该实体类和其它实体类重叠的数量，自身类与自身类重叠数量为0）组成，即该节点对于所有其它实体的重叠数量和该实体类不重叠的实体数量=>几何编码矩阵体现了实体间的位置，实体类之间的关联强度，实体类的数量（并不是直接这个实体类有多少的数量，而是重叠与不重叠的数量）

## 邻接矩阵的建立：

若某类实体重叠某类实体发生重叠，则建立一条边

## 重叠数量的计算：

- **边界框计算**：对每个实体建立一个边界框（先简化处理，获得上、下、左、右四个坐标）。
  - 对于：POLYLINE，ELLIPS，我们的数据集没有这两种，SPLINE只有极少数有1个，只考虑下述10种，（需要更精细的处理）
    - INSERT （块引用，是可复用的实体组合，还是很重要的，SOLID都是位于INSERT内）：遍历块中的实体，计算其边界框（需不需）
    - LINE：将起点和终点作为边界框的两个极端点
    - TEXT：文本的插入位置 `(x, y)` ，文本的高度和宽度，默认是横向或竖向，如果是斜的，就不精确
    - MTEXT
    - HATCH：遍历填充实体的边界路径，取边界路径的关键点
    - LWPOLYLINE：将所有顶点添加到 `points` 列表中。通过计算 `points` 中的最小和最大坐标，得到边界框。
    - LEADER：获取引线的顶点
    - CIRCLE：圆的边界框是其外接正方形
    - DIMENSION：（计算重叠不够精确）收集标注的关键点：（需不需要？）
      - `def_point` ：定义点，通常是标注的基准点。
      - `text_midpoint` ：标注文字的中点。
      - `dim_line_point` ：标注线的位置。

    - ARC：采样圆弧上的点，通过 `points` 列表中的点，计算最小和最大坐标，得到边界框
- 使用**空间索引**：**R 树（R-tree）**，来减少需要进行重叠检查的实体对数量。
- **归一化**：对于特征矩阵的每一行，将自身类对应的自身类放在第一列，后续列为固定顺序的其它实体类，对第一列的所有行进行列归一化（利用起不同实体类，数量不同），对后续列的每一行进行行归一化，有少数只有一两个实体重叠，则归一化后为0，这样就无需在这两类之间建立边

归一化的特征矩阵：

| 实体类型 | SELF | ARC | TEXT | MTEXT | LWPOLYLINE | INSERT | DIMENSION | LEADER | CIRCLE | HATCH | LINE |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ARC | 0.1751 | 0.0 | 0.0 | 0.0 | 0.2162 | 0.0 | 0.3784 | 0.0 | 0.0 | 0.0811 | 0.3243 |
| TEXT | 0.035 | 0.0 | 0.0 | 0.0 | 0.0303 | 0.0 | 0.8182 | 0.0 | 0.0 | 0.0606 | 0.0909 |
| MTEXT | 0.0078 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| LWPOLYLINE | 0.0156 | 0.3137 | 0.0196 | 0.0 | 0.0 | 0.0 | 0.5686 | 0.0 | 0.0 | 0.0196 | 0.0784 |
| INSERT | 0.0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| DIMENSION | 0.1673 | 0.1407 | 0.1357 | 0.0 | 0.1457 | 0.0 | 0.0 | 0.005 | 0.0 | 0.3166 | 0.2563 |
| LEADER | 0.0233 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| CIRCLE | 0.0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| HATCH | 0.0233 | 0.0732 | 0.0244 | 0.0 | 0.0122 | 0.0 | 0.7683 | 0.0 | 0.0 | 0.0 | 0.122 |
| LINE | 0.5525 | 0.2609 | 0.0326 | 0.0 | 0.0435 | 0.0 | 0.5543 | 0.0 | 0.0 | 0.1087 | 0.0 |

- **特殊说明：**
    - 计算没考虑自定义块内的实体（仅仅考虑模型空间中的实体），自定义块只有需要插入模型空间中，才能通过INSERT引用，没插入模型空间时，只作为预定义的形式存在
    - 后面尝试下把INSERT内部的实体一起考虑进来，看看效果
    - 后续看看分割DXF文件需不需优化
    - ```
      块信息（实体类别及数量）：
      *Model_Space: {'LINE': 72, 'TEXT': 4, 'CIRCLE': 18, 'INSERT': 7, 'DIMENSION': 22, 'LEADER'
      *Paper_Space: {'VIEWPORT': 2}
      lead00: {}
      FR: {'SOLID': 4, 'LINE': 40, 'TEXT': 31, 'LWPOLYLINE': 5, 'CIRCLE': 2}
      M0: {'LINE': 10, 'ARC': 26}
      slit: {'LINE': 25, 'ARC': 8}
      *D4: {'LINE': 1, 'MTEXT': 1, 'POINT': 3}
      *D5: {'LINE': 1, 'MTEXT': 1, 'POINT': 3}
      *D6: {'LINE': 1, 'MTEXT': 1, 'POINT': 3}
      .......
      *D203: {'LINE': 1, 'MTEXT': 1, 'POINT': 3}
      *D204: {'LINE': 1, 'MTEXT': 1, 'POINT': 3}
      *D205: {'LINE': 3, 'MTEXT': 1, 'POINT': 3}
      *D206: {'LINE': 5, 'SOLID': 2, 'MTEXT': 1, 'POINT': 3}
      ```
      ```

# 丰富数据集

## 原因:数据太少

- MGMN需要同一类多个数据，CGMN不需要，因为它只取MGMN的部分模块并且改了损失函数，变成对比学习的方式，损失函数就不需要标签了，但是我们还是采用自然语言处理中的dropout技术，对特征进行随机遮蔽的做法，随机舍弃一些节点的与其它节点的重叠情况，这样一份dxf文件，得到不同的节点间的重叠情况，得到不同的邻接矩阵，但它们是相似的，归为同一类，我们取6次随机遮蔽，这样数据集从1份变成6份，从89->6*89

## 具体做法:

- 对第一种归一化方法的提取编码矩阵除第一列的后续列，形成新的矩阵TEMP，对TEMP进行6次随机遮盖特征，形成若干个TEMP1，TEMP2，TEMP3，TEMP4, TEMP5, TEMP6
- 把TEMP1，TEMP2，TEMP3， TEMP4, TEMP5, TEMP6大于0的元素视为有边，进一步转化为邻接表adj1,adj2,adj3,adj4, adj5,adj6
- 把这些TEMP1，TEMP2，TEMP3， TEMP4, TEMP5, TEMP6与原先的第一列拼接，形成f1，f2，f3, f4, f5, f6作为新的编码矩阵，这些编码矩阵都属于来自同一份dxf文件
- 随机遮盖特征的做法：dropout取0.1=> f1，f2，f3, f4, f5, f6

```
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.        ]
[0.16731518 0.14070352 0.13567839 0.         0.14572864 0.
 0.         0.00502513 0.         0.31658291 0.25628141]
[0.0233463  0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.        ]
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.        ]
[0.0233463  0.07317073 0.02439024 0.         0.01219512 0.
 0.76829268 0.         0.         0.         0.12195122]
[0.55252918 0.26086957 0.0326087  0.         0.04347826 0.
 0.55434783 0.         0.         0.10869565 0.        ]]
adj1 (邻接表):
{0: [3, 5, 8, 9], 1: [3, 5, 9], 2: [], 3: [0, 1, 5, 8, 9], 4: [], 5: [0, 1, 3, 6, 8, 9], 6: [], 7: [], 8: [0, 1, 3, 5, 9], 9: [0, 1, 3, 5, 8]}
f2 (特征矩阵):
[[0.17509728 0.         0.         0.         0.21621622 0.
  0.37837838 0.         0.         0.08108108 0.32432432]
 [0.03501946 0.         0.         0.         0.03030303 0.
  0.81818182 0.         0.         0.06060606 0.09090909]
```

# CGMN模块的结果

## 我们的相关指标：

```
/mnt/share/CGMN/CGMN/data/CFG/OpenSSL_11ACFG_min10_max10/acfgSSL_11 Train: 210 graphs,
/mnt/share/CGMN/CGMN/data/CFG/OpenSSL_11ACFG_min10_max10/acfgSSL_11 Dev   : 162 graphs,
/mnt/share/CGMN/CGMN/data/CFG/OpenSSL_11ACFG_min10_max10/acfgSSL_11 Test  : 162 graphs,
Before Training: the val AUC score is: 0.8322473708276178
Before Training: the lr AUC score is: 0.7435897435897436
Before Training: the test AUC score is: 0.7919143423258 65
Before Training: the test lrAUC score is: 0.7413003663003663
```

```
Final Testing: the cosine AUC score is: 0.8727137631458619
Final testing: the lr AUC score is: 0.7803873722739058
Best_valid_auc =  0.8446121018137478
```

## 原始论文相关指标：

| Methods | OS [50, 200] | OS [20, 200] | OS [3, 200] | FF [50, 200] | FF [20, 200] | FF [3, 200] |
|---|---|---|---|---|---|---|
| GCN | 67.24±1.14 | 68.09±1.01 | 73.51±0.72 | 78.41±0.49 | 79.47±0.08 | 80.88±0.18 |
| GIN | 66.60±0.10 | 63.85±0.56 | 75.65±0.30 | 78.38±0.20 | 81.25±0.57 | **81.82±0.25** |
| DGI | 67.55±2.76 | 63.58±1.96 | 72.58±2.36 | 86.10±0.66 | 80.82±2.22 | 66.28±0.30 |
| GRACE | 68.84±2.45 | 67.01±0.49 | 69.86±0.29 | 85.44±0.27 | 75.05±0.73 | 66.95±2.78 |
| ScGSLC | 67.43±0.82 | 61.46±0.33 | 63.28±0.09 | **87.57±0.82** | 83.27±0.71 | 69.80±1.22 |
| CGMN | **80.89±0.20** | **78.15±0.85** | **75.94±1.86** | 86.11±0.98 | **86.76±0.85** | 77.98±2.69 |

Table 3: Experimental results on the BSD datasets in terms of AUC scores (%).

## 下面仅仅是简单的测试，测试用的DXF文件，模型在训练，验证，测试时是没有见过的

## 使用扩展后的数据集测量相似度

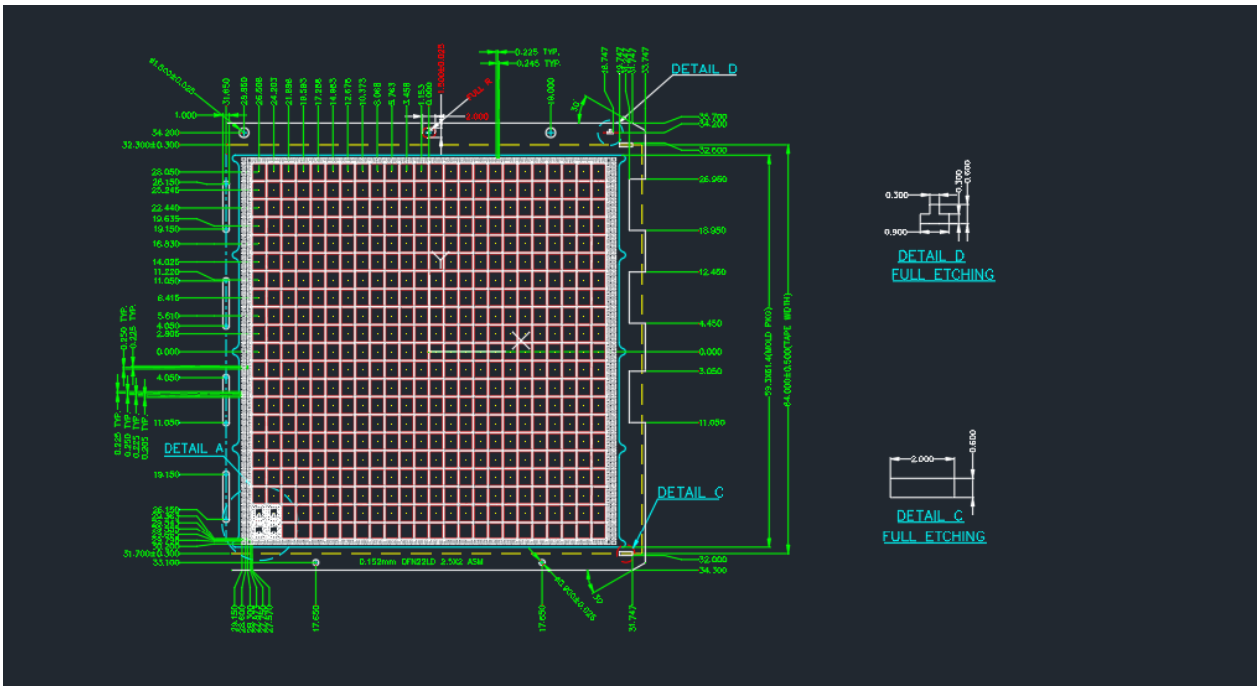
同一类6份dxf之间的相似度（以QFN22LD(Cu)-532Rev1_2.dxf为基准）

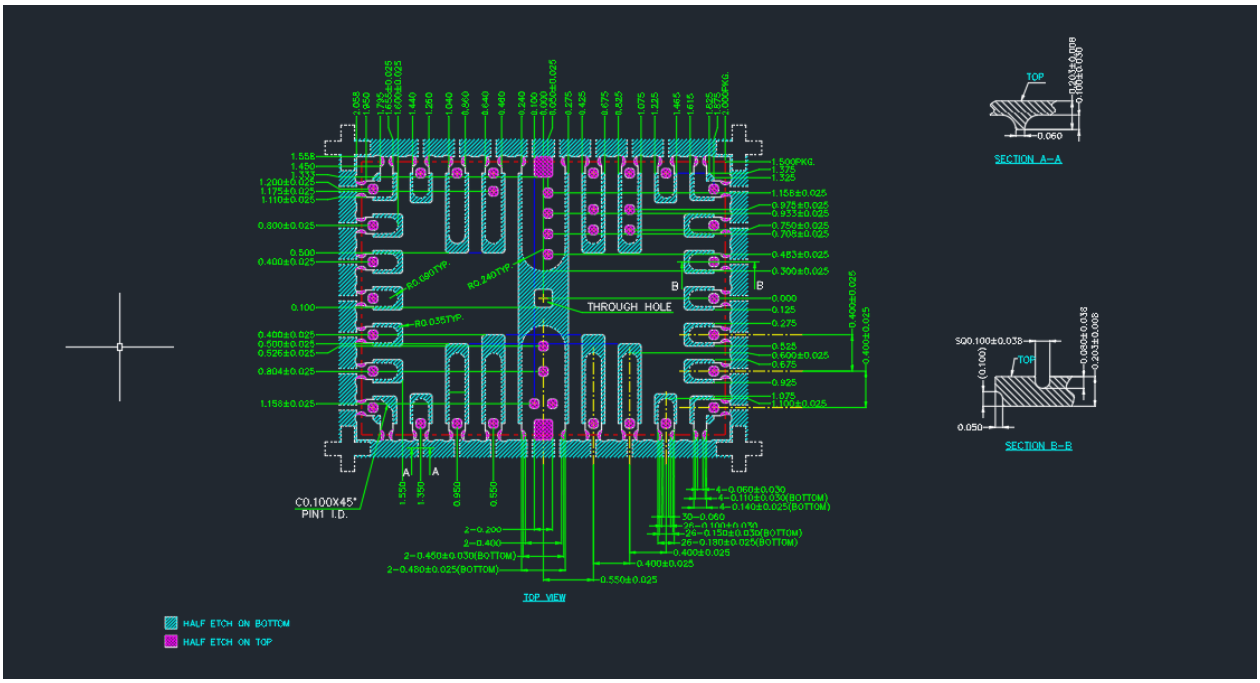
```
root@csip-090:/mnt/share/CGMN/CGMN/src# python Calculate_similarity.py
/mnt/share/CGMN/CGMN/CFGLogs/同一类之间/2_2.json
/mnt/share/CGMN/CGMN/CFGLogs/同一类之间/2_1.json
classification task
图对的相似度为： 1.0
root@csip-090:/mnt/share/CGMN/CGMN/src# python Calculate_similarity.py
/mnt/share/CGMN/CGMN/CFGLogs/同一类之间/2_2.json
/mnt/share/CGMN/CGMN/CFGLogs/同一类之间/2_3.json
classification task
图对的相似度为： 0.9859582781791687
root@csip-090:/mnt/share/CGMN/CGMN/src# python Calculate_similarity.py
/mnt/share/CGMN/CGMN/CFGLogs/同一类之间/2_2.json
/mnt/share/CGMN/CGMN/CFGLogs/同一类之间/2_2.json
classification task
图对的相似度为： 1.0
root@csip-090:/mnt/share/CGMN/CGMN/src# python Calculate_similarity.py
/mnt/share/CGMN/CGMN/CFGLogs/同一类之间/2_2.json
/mnt/share/CGMN/CGMN/CFGLogs/同一类之间/2_4.json
classification task
图对的相似度为： 1.0
root@csip-090:/mnt/share/CGMN/CGMN/src# python Calculate_similarity.py
/mnt/share/CGMN/CGMN/CFGLogs/同一类之间/2_2.json
/mnt/share/CGMN/CGMN/CFGLogs/同一类之间/2_5.json
classification task
图对的相似度为： 0.9859582781791687
root@csip-090:/mnt/share/CGMN/CGMN/src# python Calculate_similarity.py
/mnt/share/CGMN/CGMN/CFGLogs/同一类之间/2_2.json
/mnt/share/CGMN/CGMN/CFGLogs/同一类之间/2_6.json
```

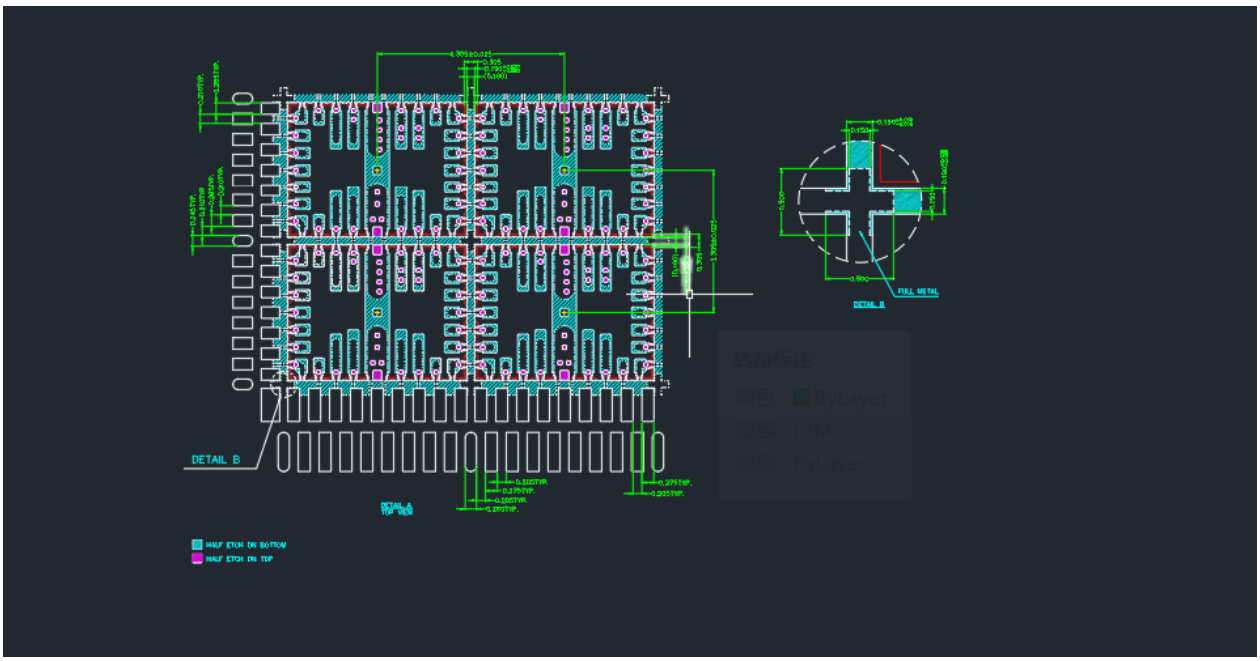
我们认为同一张DXF文件扩展利用随机去掉一些边和节点得到的6份DXF数据，是属于同一类的，结果也证实了这一点的合理性

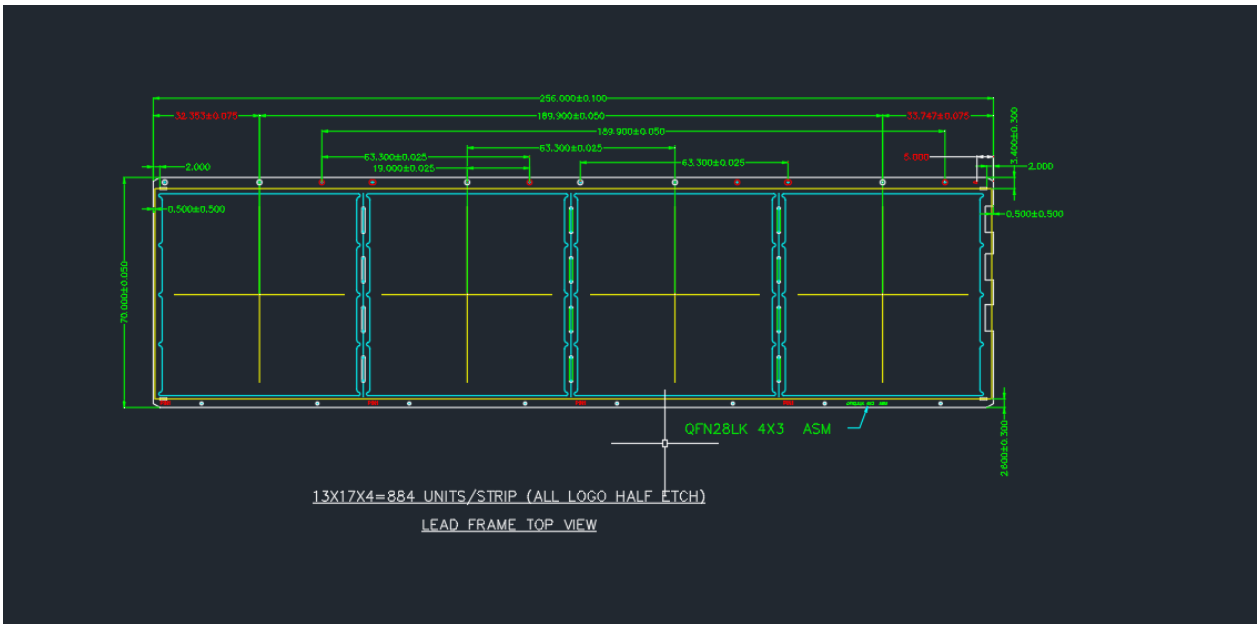
# 不同类之间的相似度（以"QFN22LD(Cu)-532Rev1_3.dxf为基准)

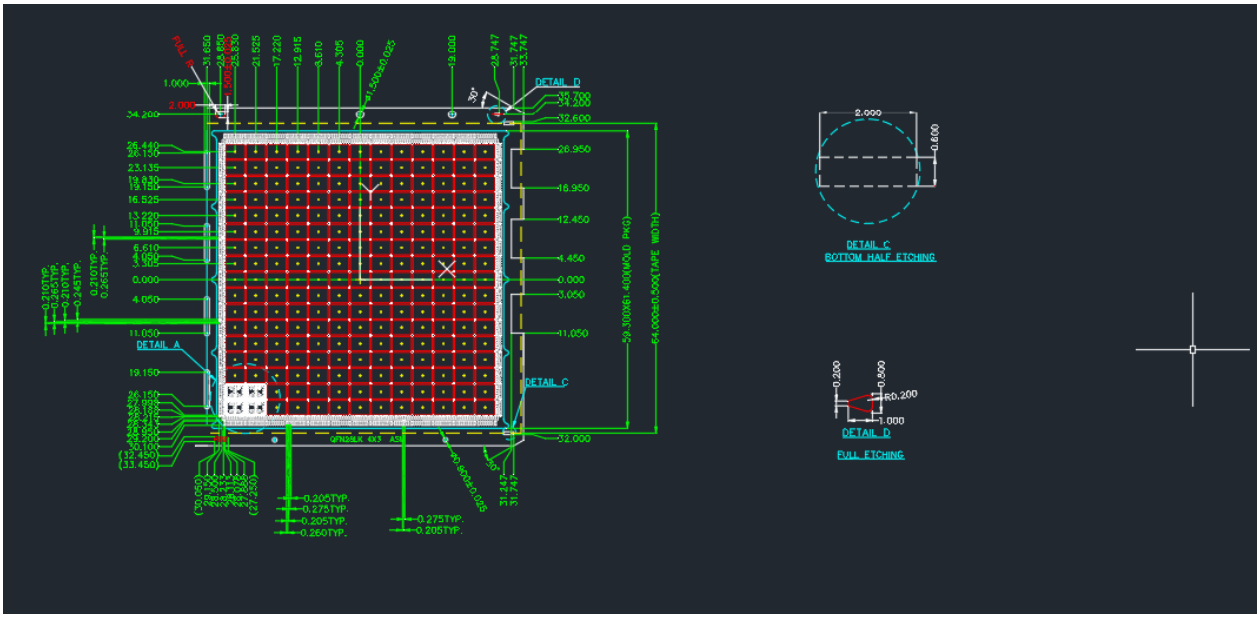
```
root@csip-090:/mnt/share/CGMN/CGMN/src# python Calculate_similarity.py
/mnt/share/CGMN/CGMN/CFGLogs/不同类之间/2_3.json
/mnt/share/CGMN/CGMN/CFGLogs/不同类之间/1_1.json
classification task
图对的相似度为： 0.900559663772583
root@csip-090:/mnt/share/CGMN/CGMN/src# python Calculate_similarity.py
/mnt/share/CGMN/CGMN/CFGLogs/不同类之间/2_3.json
/mnt/share/CGMN/CGMN/CFGLogs/不同类之间/1_2.json
classification task
图对的相似度为： 0.7371960878372192
root@csip-090:/mnt/share/CGMN/CGMN/src# python Calculate_similarity.py
/mnt/share/CGMN/CGMN/CFGLogs/不同类之间/2_3.json
/mnt/share/CGMN/CGMN/CFGLogs/不同类之间/1_3.json
classification task
图对的相似度为： 0.9239799380302429
root@csip-090:/mnt/share/CGMN/CGMN/src# python Calculate_similarity.py
/mnt/share/CGMN/CGMN/CFGLogs/不同类之间/2_3.json
/mnt/share/CGMN/CGMN/CFGLogs/不同类之间/1_4.json
classification task
图对的相似度为： 0.9504803419113159
```


DETAIL_D
DETAIL_A
DETAIL_C
DETAIL_D
FULL_ETCHING
DETAIL_C
FULL_ETCHING

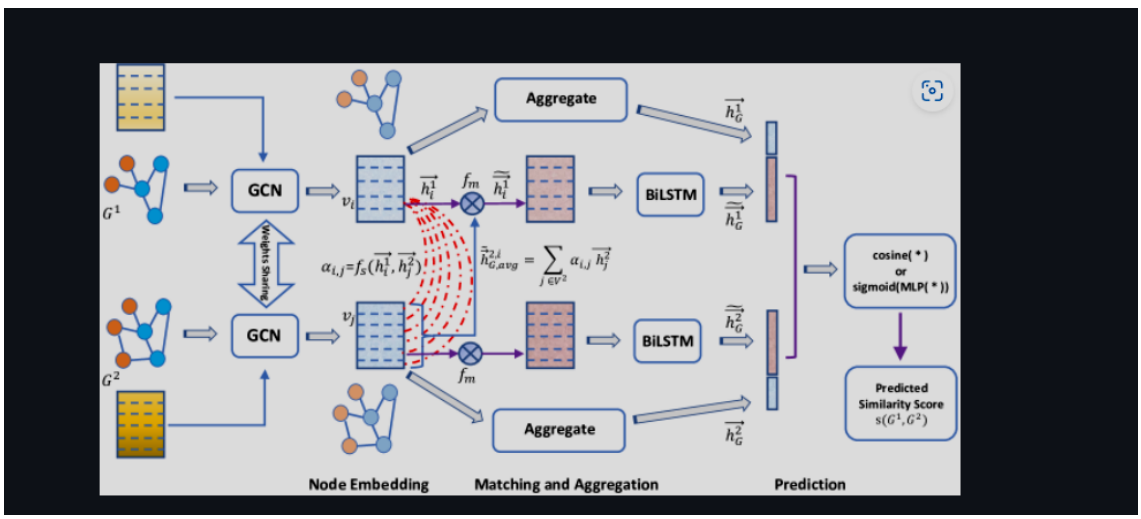

THROUGH_HOLE
C0.100X45°
PIN1 I.D.
TOP_VIEW
SECTION_A-A
SECTION_B-B
HALF ETCH ON BOTTOM
HALF ETCH ON TOP


DETAIL_B
DETAIL_B
HALF ETCH ON BOTTOM
HALF ETCH ON TOP




# 网络结构

- MGMN
  - 

  - SimGNN

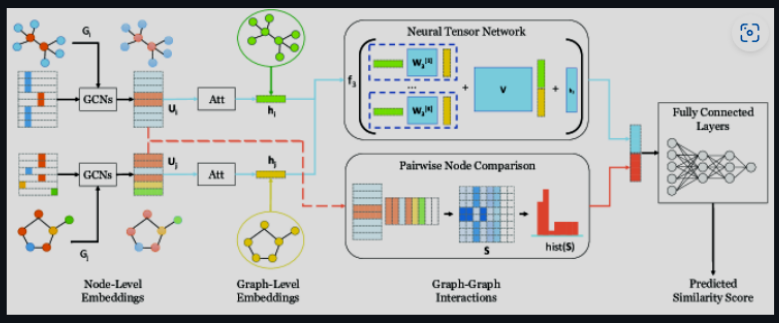
- 


  Figure 3. An overview illustration of SimGNN. The blue arrows denote the data flow for Strategy 1, which is based on graph-level embeddings. The red arrows denote the data flow for Strategy 2, which is based on pairwise node comparison.

  图 3. SimGNN 的概述图示。蓝色箭头表示策略 1 的数据流，该策略基于图形级嵌入。红色箭头表示策略 2 的数据流，它基于成对节点比较。

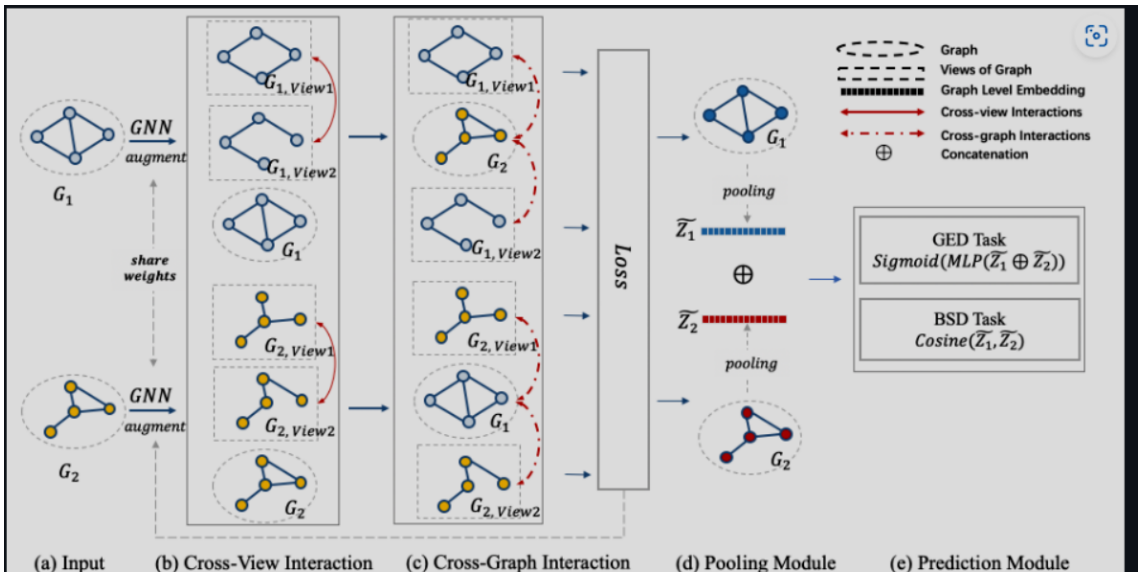
- CGMN
  - CGMN将MGMN的loss改动就变成了自监督
- $$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(\tilde{y} - y)^2$$
- $$loss(h_u^*, h_v^*) = -\log \frac{\text{sim}(h_u^*, h_v^*)}{\text{sim}(h_u^*, h_v^*) + \sum_{k=1}^{N} \text{sim}(h_u^*, h_k^*)},$$

  air of positive samples and $N$ is the number of negative sample
  d, then the total loss can be defined as their average result:

  本，是 $N$ 负样本的数量。节点 $u$ 和节点 $v$ 都需要计算，那么总损失

  $$\mathcal{L} = \frac{1}{2}\left[loss(h_u^*, h_v^*) + loss(h_v^*, h_u^*)\right].$$

- 添加了视图（图增强）的跨级特征提取
  - 每一个视图的节点具有另外一个视图全部节点的信息以及另一张图全部节点的信息




- CGMN的正负对:同一份DXF文件，生成两个视图，对这两个视图的正样本对（在一个视图中某个节点，正样本对就是同一份DXF产生的另外一个视图的对应位置的节点），相似度最大化，负样本对相似度最小化（负样本对定义为，一个视图中某一个点，它的负样本对就是它和这个视图的其它节点（不包括自身和自身），以及同一份DXF产生的另外一个视图的所有节点）

# 序列特征

## 句柄

HATCH 1703BF
1508287
TEXT 1703C0
1508288
MTEXT 1E19DB
1972699
LWPOLYLINE 1E1C1B
1973275
MTEXT 1E2DB1
1977777
HATCH 1EB8C3
2013379
HATCH 1EB8C4
2013380
LWPOLYLINE 1EB8C5

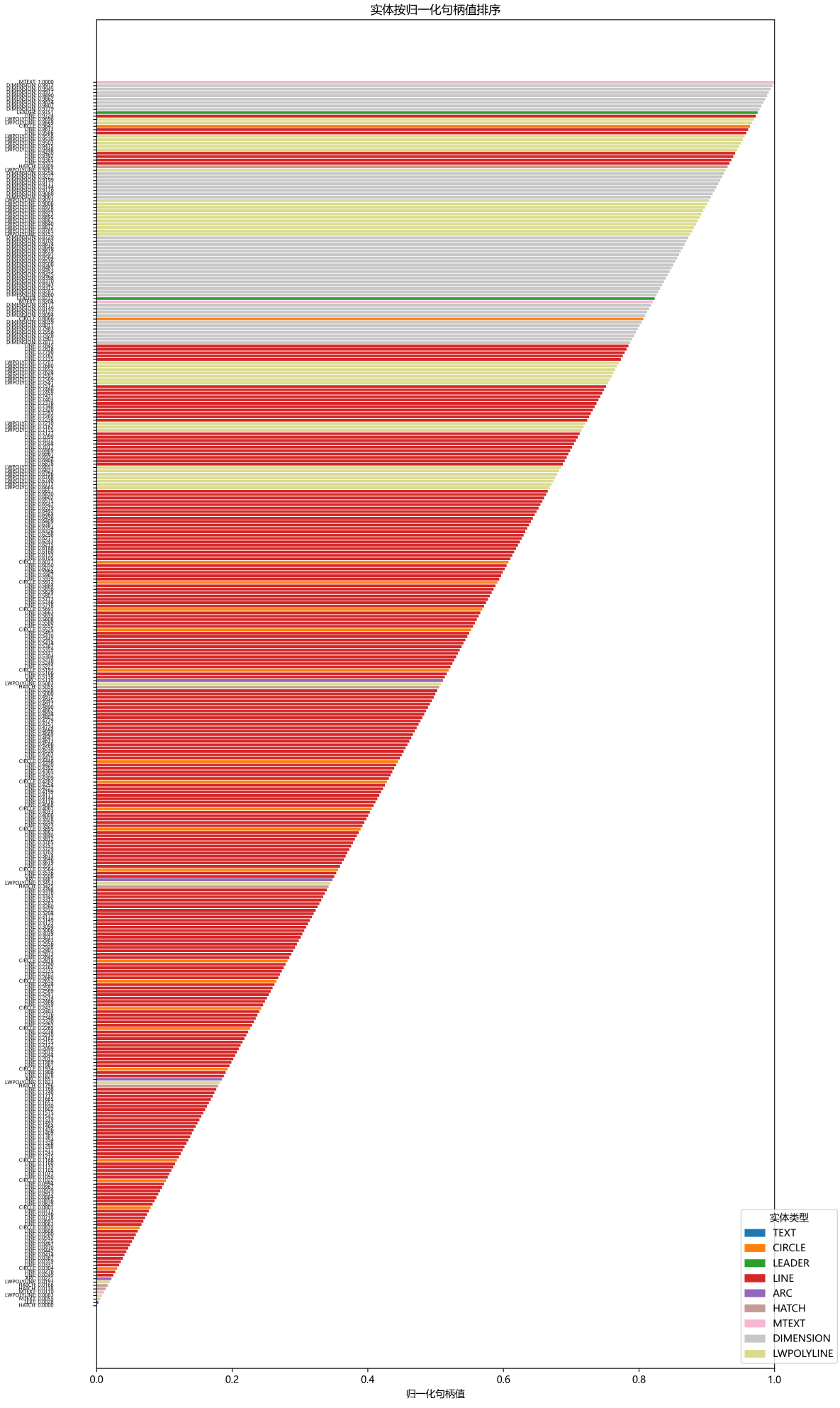
## 原因：

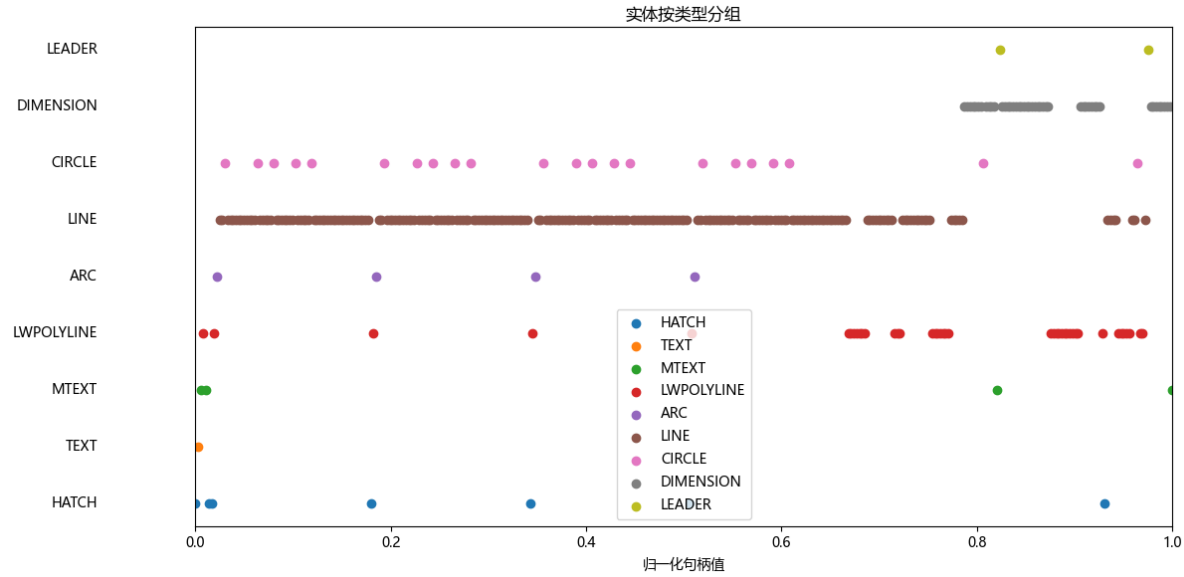
DXF文件的实体句柄的值从小到大表示该实体添加入DXF文件的顺序，反映了设计意图和因果关系，某些实体的存在可能是其他实体存在的前提条件

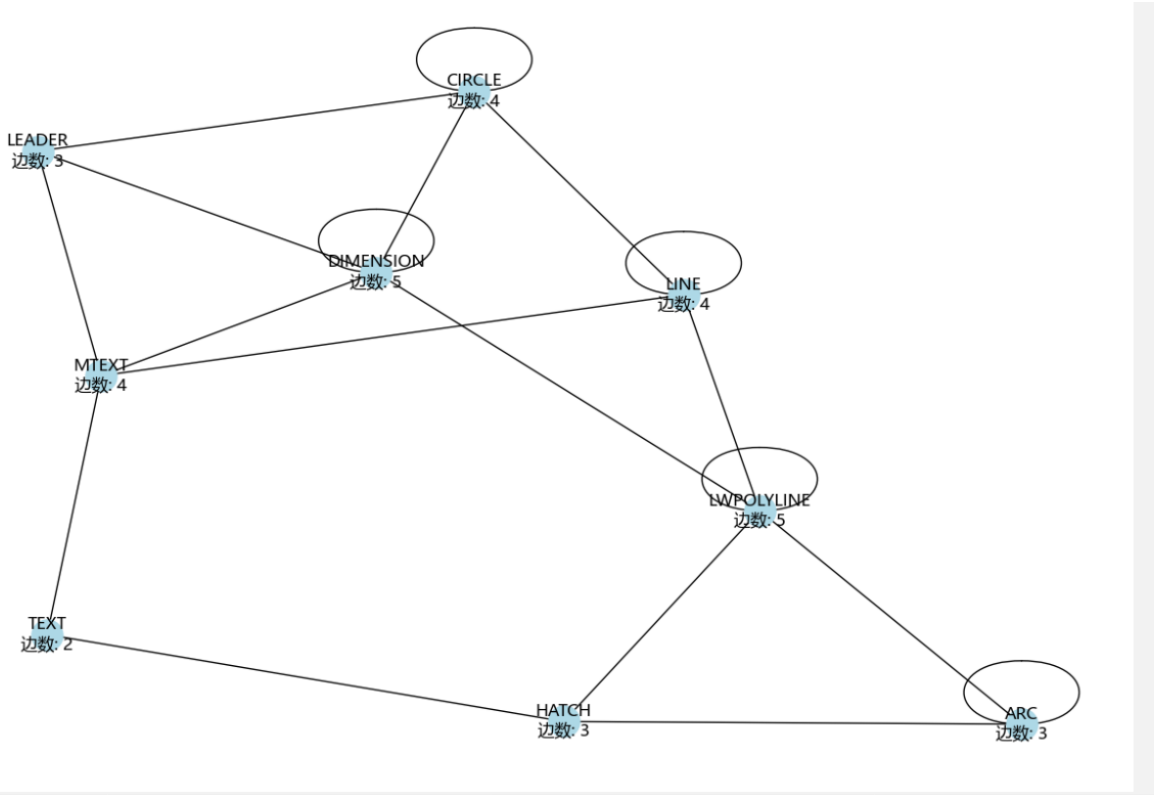
## 处理方式：

1. 预处理：因为我们输入的DXF文件并不是完整的一份DXF文件，而是分割后的DXF文件，这样造成句柄值并不是连续的，例如：MTEXT的句柄值为C0DB5，而下一个实体LEADER句柄值为D13E3，我们需要将下一个实体LEADER句柄值处理为C0DB6
2. 数值化：将十六进制的句柄值转换为十进制数。
3. 归一化：使用min-max归一化将句柄值映射到[0, 1]区间。

# 可视化结果


实体按归一化句柄值排序
归一化句柄值
0.0
0.2
0.4
0.6
0.8
1.0
实体类型
TEXT
CIRCLE
LEADER
LINE
ARC
HATCH
MTEXT
DIMENSION
LWPOLYLINE




# 融合方法

## 最直接的方式

一个实体类视为一个节点，每个节点考虑下述特征：聚类数量（该实体类型的聚类数量），平均聚类密度（所有聚类的平均密度），聚类密度方差（反映密度分布的均匀程度），最大聚类间距（最远两个聚类中心之间的距离），最小聚类间距（最近两个聚类中心之间的距离），聚类间距方差（反映聚类分布的均匀程度），首次出现时间，最后出现时间.....

把句柄得到的序列编码矩阵拼接到之前的几何编码矩阵之后

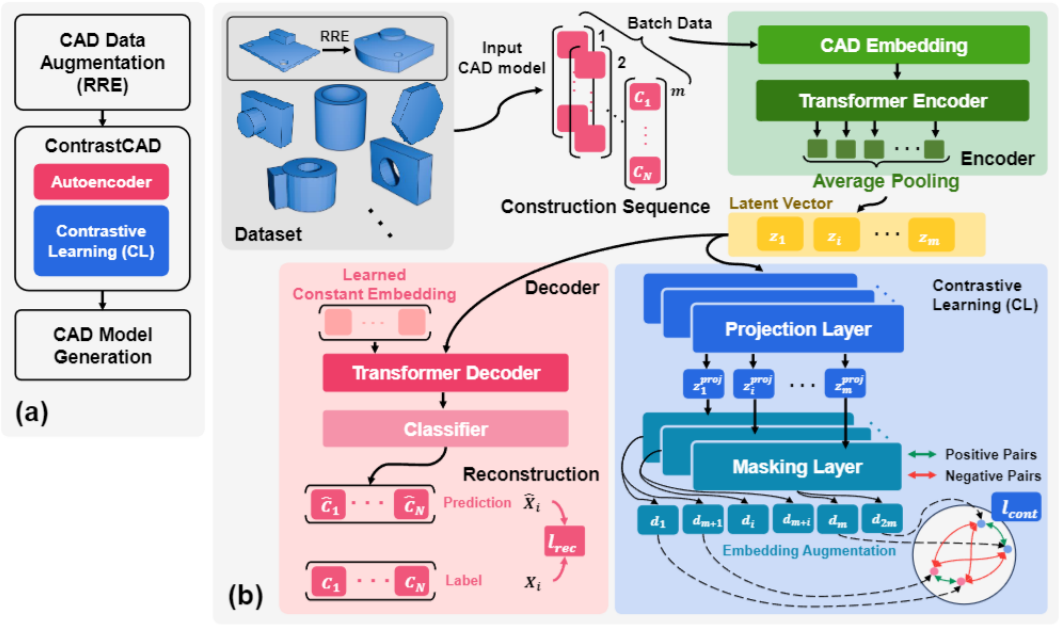
# 邻接矩阵的第二种建立方式：

边的条数和之前会比之前那种方法建立边的条数会更少一点（差不多10个点，10~20条）先利用相关模型检测出异常，然后剔除，最后如果一个实体前是另一种实体，或者一个实体后是另一种实体，则在这两个实体间建立一条边

| Tasks 任务 | Datasets 数据 | Sub-Datasets 子数据集 | # of Graphs # 图形 | # of Functions # 函数 | AVG # of Nodes 平均节点数 # | AVG # of Edges 平均 # 个边数 | Initial Feature Dimensions 初始特征尺寸 |
|---|---|---|---|---|---|---|---|
| Graph-Graph 图形 Classification Task 分类任务 | FFmpeg | [3, 200] | 83,008 | 10,376 | 18.83 | 27.02 | 6 |
| | | [20, 200] | 31,696 | 7,668 | 51.02 | 75.88 | |
| | | [50, 200] | 10,824 | 3,178 | 90.93 | 136.83 | |
| | OpenSSL 开放SSL | [3, 200] | 73,953 | 4,249 | 15.73 | 21.97 | 6 |
| | | [20, 200] | 15,800 | 1,073 | 44.89 | 67.15 | |
| | | [50, 200] | 4,308 | 338 | 83.68 | 127.75 | |

# ContrastCAD

ContrastCAD: Contrastive Learning-based Representation Learning for Computer-Aided Design Models (arxiv.org)，202404 IEEE ACCESS




# 目的：

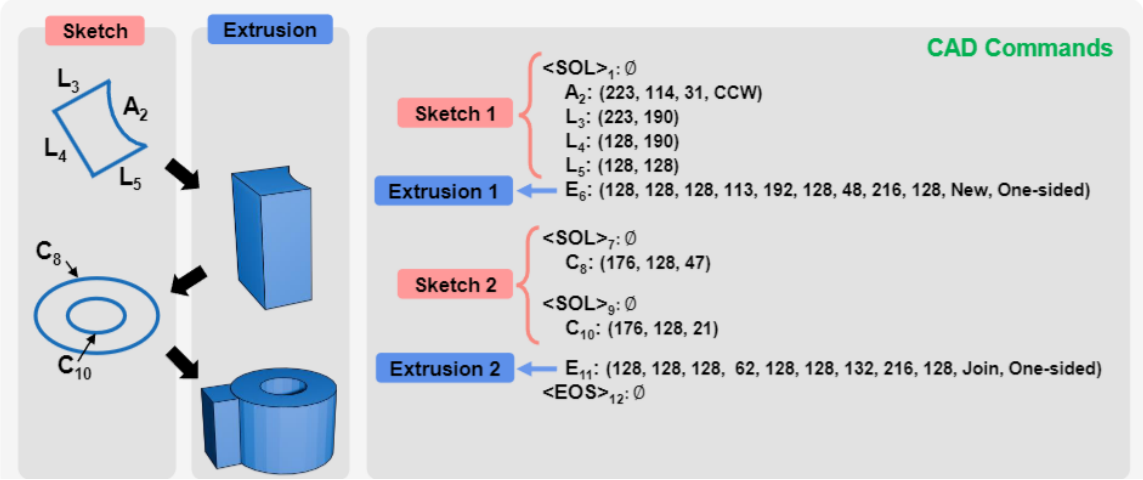
生成多样化，全新的CAD（利用latent-GAN生成（输入标准噪声数据到模型（训练好的基于Transform的自编码器）中生成）

# 相似之处：

“相同的 CAD 模型可以使用不同的 CAD 构建顺序来表示“这和我们的句柄数据差不多，不同的句柄数据确实可以表示相同的CAD

# 特点：

**输入序列的处理方式**：




一份CAD命令序列由若干条命令组成，论文中代码命令条数设置最大为60条，每一条命令，包含命令编码（用数字表示），参数编码：将所有命令的设计的参数全部调出来，在这个命令中有就填，没有就0，位置编码：这条命令在整个序列中的位置

**Encoder. 编码器。**

The encoder of ContrastCAD consists of a CAD embedding layer, a Transformer encoder, and an average pooling layer. First, the CAD embedding layer takes the input $X_i = [C_1, ..., C_N]$ and outputs CAD embeddings $[e_1, ..., e_N]$. Unlike natural language processing, $C_k$ is separated into $t_k$ that represents the type of command (i.e., line, arc, circle, extrusion, ⟨SOL⟩, ⟨EOS⟩) and $p_k \in \mathbb{R}^{1X16}$ that represents the parameters of the command. As explained in Section 3, there are 16 types of parameters (i.e., $p_k = [x, y, \theta, c, r, \alpha, \beta, \gamma, o_x, o_y, o_z, s, \delta_1, \delta_2, b, w]$). Therefore, embeddings need to be performed separately for $t_k$ and $p_k$. Let $d_E$ denotes the embedding dimension, the CAD embedding $e_k \in \mathbb{R}^{1\times d_E)}$ for $C_k$ is calculated as follows by adding a learned positional encoding $e^{pos}$ to the embedding of $t_k$ and $p_k$:

ContrastCAD的编码器由CAD嵌入层、Transformer编码器和平均池化层组成。首先，CAD嵌入层获取输入 $X_i = [C_1, ..., C_N]$ 并输出 CAD 嵌入 $[e_1, ..., e_N]$。与自然语言处理不同，$C_k$ 被分成 $t_k$ 表示命令的类型（即直线、圆弧、圆、挤压、⟨SOL⟩，⟨EOS⟩）和 $p_k \in \mathbb{R}^{1X16}$ 代表命令的参数。正如第 3 节中所解释的，有 16 种类型的参数（即 $p_k = [x, y, \theta, c, r, \alpha, \beta, \gamma, o_x, o_y, o_z, s, \delta_1, \delta_2, b, w]$）。因此，需要单独执行嵌入 $t_k$ 和 $p_k$。让 $d_E$ 表示嵌入尺寸，CAD嵌入 $e_k \in \mathbb{R}^{1\times d_E)}$ 为了 $C_k$ 通过添加学习的位置编码计算如下 $e^{pos}$ 到嵌入 $t_k$ 和 $p_k$：

$$e_k = t_k W^{cmd} + p_k W^{param} + e^{pos}, \quad (1)$$

where $W^{cmd} \in \mathbb{R}^{1\times d_E}$ and $W^{param} \in \mathbb{R}^{16\times d_E}$ are weight matrices.

在哪里 $W^{cmd} \in \mathbb{R}^{1\times d_E}$ 和 $W^{param} \in \mathbb{R}^{16\times d_E}$ 是权重矩阵。

Next, the Transformer encoder takes $[e_1, ..., e_N]$ as input, performs self-attention and feed-forward computations, and outputs $[h_1, ..., h_N]$. The Transformer encoder consists of $L$ stacked layers of self-attention and feed-forward layers, following the conventional Transformer encoder architecture. Finally, after passing through an average pooling layer, the latent vector $z_i$ $(i = 1, ..., m)$ of $X_i$ is produced as output, where the dimension of $z_i$ is $d_E$.

接下来，Transformer 编码器采用 $[e_1, ..., e_N]$ 作为输入，执行自注意力和前馈计算，并输出 $[h_1, ..., h_N]$。Transformer 编码器包括 $L$ 自注意力层和前馈层的堆叠层，遵循传统的 Transformer 编码器架构。最后，经过平均池化层后，潜在向量 $z_i$（$i = 1, ..., m$）的 $X_i$ 产生为输出，其中的维度 $z_i$ 是 $d_E$。

All experiments were implemented using PyTorch and trained on an RTX A6000 GPU. Following [5], we fixed $N$ to 60 for all construction sequences, and shorter sequences were padded with the ⟨EOS⟩ command. We set $d_E$ to 256, $L$ to 4, feed-forward dimension to 512, and the number of attention heads to 4. The dropout rate was set to 0.1. During training, we utilized the Adam optimizer [32] with an initial learning rate of 0.001, along with linear warmup for 2,000 steps and gradient clipping at 1.0. We trained for 1,000 epochs with a batch size ($m$) of 1,024. The $\lambda$ and $\kappa$ values for the loss function were set to 2, and $\tau$ was set to 0.07. The $\eta$ value in Equation (8) was set to 3.

所有实验均使用 PyTorch 实施，并在 RTX A6000 GPU 上进行训练。按照[ 5 ]，我们修复了 $N$ 所有构建序列为 60，较短的序列用 ⟨EOS⟩ 命令。我们设定 $d_E$ 至 256， $L$ 为 4，前馈维度为 512，注意力头数量为 4。dropout 率设置为 0.1。在训练过程中，我们使用 Adam 优化器[ 32 ]，初始学习率为 0.001，以及 2,000 步的线性预热和 1.0 的梯度裁剪。我们训练了 1,000 个 epoch，批量大小为（$m$）共 1,024 个。这 $\lambda$ 和 $\kappa$ 损失函数的值设置为 2，并且 $\tau$ 被设置为0.07。这 $\eta$ 式( 8 )中的值设置为3。

基于 **Transformer 的自编码器模型**：将序列映射到具有语义信息的特征空间

**对比学习**：在嵌入空间中创建正对（相同文件，但应用了不同的掩膜）和负对（不同的文件）来实现对比学习。CGMN的正负对是在节点层面的，ContrastCAD是在整个文件上的

**损失函数：**

$$l_{ContrastCAD} = l_{rec} + \kappa l_{cont}, \tag{2}$$

where $\kappa$ is a balancing hyperparameter between two terms.

其中 $\kappa$ 是两个术语之间的平衡超参数。

As each command in the construction sequence is divided into command type and parameters, it is necessary to learn representations for command types as well as command parameters. Therefore, $l_{rec}$ is calculated as the sum of cross-entropy loss between ($t_k$ and $\hat{t}_k$) and ($p_k$ and $\hat{p}_k$) and denoted as:

由于构造序列中的每个命令都分为命令类型和参数，因此有必要学习命令类型和命令参数的表示。因此， $l_{rec}$ 计算为（$t_k$ 和 $\hat{t}_k$） 和 （$p_k$ 和 $\hat{p}_k$） 之间的交叉熵损失之和，表示为：

$$l_{rec} = CE(t_k, \hat{t}_k) + \lambda CE(p_k, \hat{p}_k), \tag{3}$$

**判别指标**：使用聚类作为判别指标

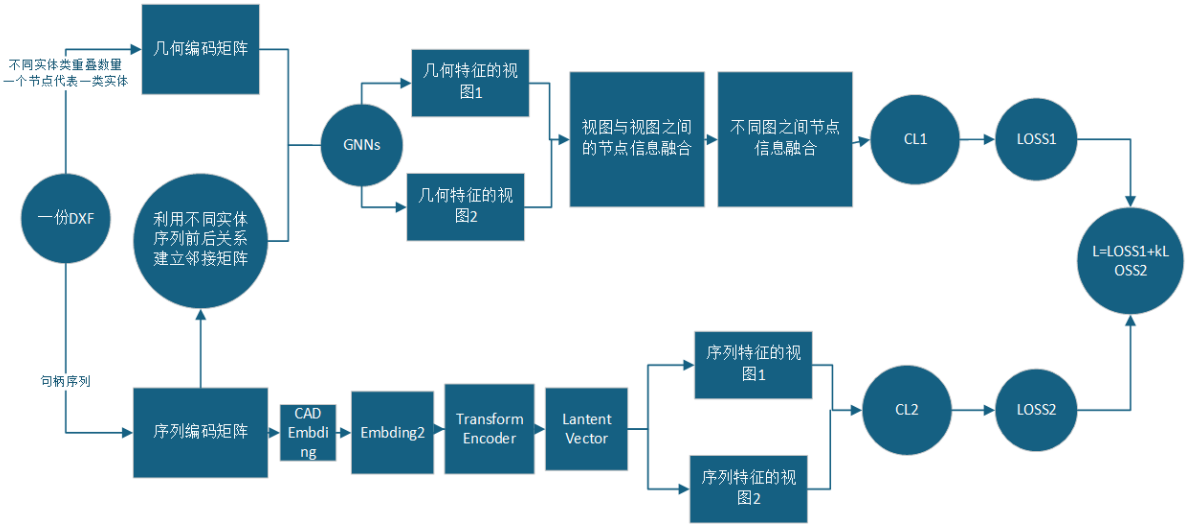
# 拟处理方式

1.先处理掉异常的（找一找相关方法）

2.我们的数据大多是是两三百个实体，有出现1700的，先定一个固定值，把长度大于固定值的删掉，然后先看看直接设置512行不行，如何行，就不需要改动

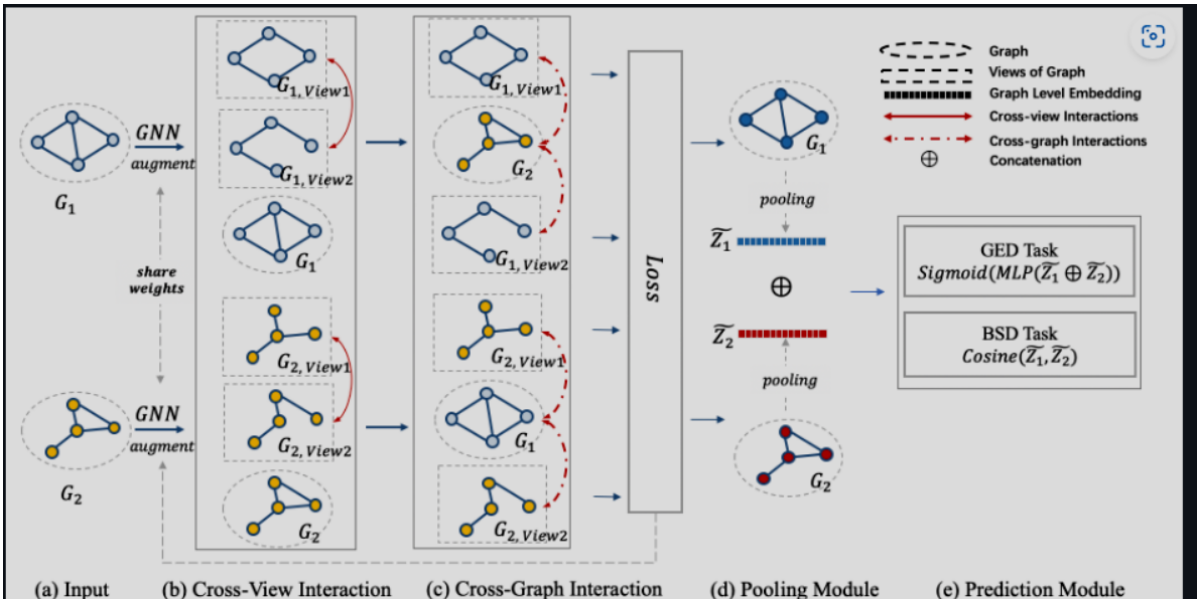
否则我们考虑分层，下面那层（小窗口）使用（CAD Embedding）,在CAD Embedding和Transformer Encoder之间添加一个模块，作为上层，比如我们要对整个句柄序列分40个小窗口，一个窗口大小就是（总长/40），对一个窗口内使用（CAD Embedding），然后整个窗口用MLP处理到一个特征向量，每个小窗口拼起来就是Transformer Encoder的输入

# 模型粗稿

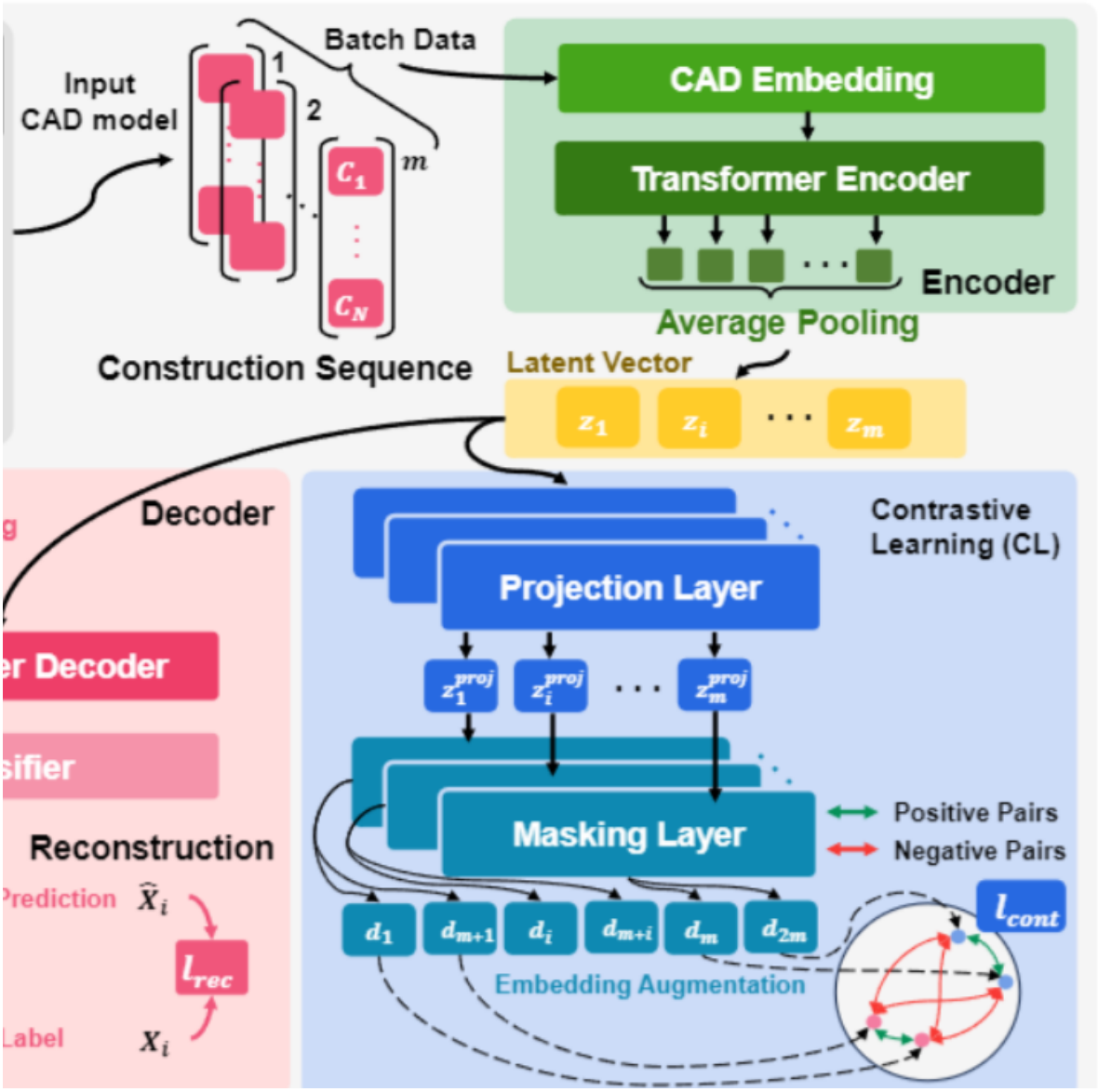



## 来源：

几何特征矩阵




序列特征矩阵




# 计划

1.先处理掉异常的（找一找相关方法）

## 2.句柄序列编码

### 2.1 实体参数

HATCH 特征:

{'pattern_name': 'ANSI31', 'solid_fill': 0, 'associative': 0, 'boundary_paths': 8}
{'pattern_name': 'ANSI31', 'solid_fill': 0, 'associative': 1, 'boundary_paths': 4}

pattern_name: 填充图案名称

solid_fill: 是否为实体填充， 0 表示非实体填充,即使用图案填充

associative: 是否为关联填充，0 表示非关联填充,1 表示关联填充。关联填充意味着填充边界与其他实体（如线、圆等）相关联。当这些边界实体发生变化时，填充会自动更新以匹配新的边界。

'boundary_paths'：边界路径数量指的是定义填充区域的封闭轮廓的数量

TEXT 特征:
{'text': 'HALF ETCH ON BOTTOM', 'insert_point': Vec3(396.0818019561805, 139.6943445264609, 0.0), 'height': 0.08, 'rotation': 0}

text: 文本内容

insert_point: 插入点坐标

height: 文字高度

rotation: 旋转角度

MTEXT 特征:
{'text': 'FULL METAL', 'insert_point': Vec3(404.1361941913078, 142.9183805374713, 0.0), 'char_height': 0.06666666, 'width': 0.6219441976340474, 'rotation': 0}

text: 多行文本内容

insert_point: 插入点坐标

char_height: 字符高度

width: 文本框宽度

LWPOLYLINE 特征:
{'closed': True, 'points': [(395.819853154926, 139.6625137883657, 0.0, 0.0, 0.0), (395.9806541764283, 139.6625137883657, 0.0, 0.0, 0.0), (395.9806541764283, 139.8233148098681, 0.0, 0.0, 0.0), (395.819853154926, 139.8233148098681, 0.0, 0.0, 0.0)], 'count': 4}

closed: 是否闭合

points: 多段线的点列表

count: 点的数量

ARC 特征:
{'center': Vec3(398.8221825223083, 143.4364333052549, 0.0), 'radius': 0.11875, 'start_angle': 180.0, 'end_angle': 90.0}

center: 圆心坐标

radius: 半径

start_angle: 起始角度

end_angle: 结束角度

LINE 特征:
{'start_point': Vec3(398.9971825223084, 143.5551833052549, 0.0), 'end_point': Vec3(398.5440575223073, 143.5551833052551, 0.0)}

start_point: 线段的起点坐标

end_point: 线段的终点坐标

CIRCLE 特征:
{'center': Vec3(398.8221825223083, 142.9364333052549, 0.0), 'radius': 0.11875}

center: 圆心坐标

radius: 半径

DIMENSION 特征:
{'defpoint': Vec3(400.3284325223083, 146.1966954184052, 0.0), 'text_midpoint': Vec3(399.7003075223083, 146.1966954184052, 0.0), 'dim_type': 0}

defpoint: 定义点坐标

text_midpoint: 文本中点坐标

dim_type: 标注类型

LEADER 特征:
{'vertices': [(398.2190951127764, 141.8598343414721, 0.0), (396.7597151407759, 140.0412732757189, 0.0), (395.7713643418502, 140.0412732757189, 0.0)], 'annotation_type': 3}

vertices: 引线的顶点列表

annotation_type: 注释类型

INSERT 特征:
{'name': '*U121', 'insert_point': Vec3(384.5059270744619, 143.5915590496204, 0.0), 'scale': (1, 1, 1), 'rotation': 0}

name: 插入块的名称

insert_point: 插入点的坐标

scale: X、Y、Z方向的缩放比例

rotation: 旋转角度

3.ContrastCAD模块的测试或者添加Embding

4.将几何特征和序列特征两个方向改成完整的一个模型