# PRML期末Project实验报告

> PRML课程期末Project
>
> 小组：聂希瑞(学号16307130133)
>
> 金辰哲(学号18307130112)

这份文档是我们小组关于期末项目的实验报告，本次实验采用了分工合作的方式，对目标问题分别利用传统方法和图网络方法进行了尝试和训练，对不同方法的结果进行了记录；并对不同方法的结果进行横向比较。最终得到的结果具有较好的参考价值。

# 目录

# 一、背景介绍

## 1. SMILES 字符串

SMILES全称Simplified Molecular Input Line Entry Specification，即简化分子线性输入规范，是一种用ASCII字符串明确描述分子结构的规范。形如：`O=[N+]([O-])C(Br)(CO)CO`

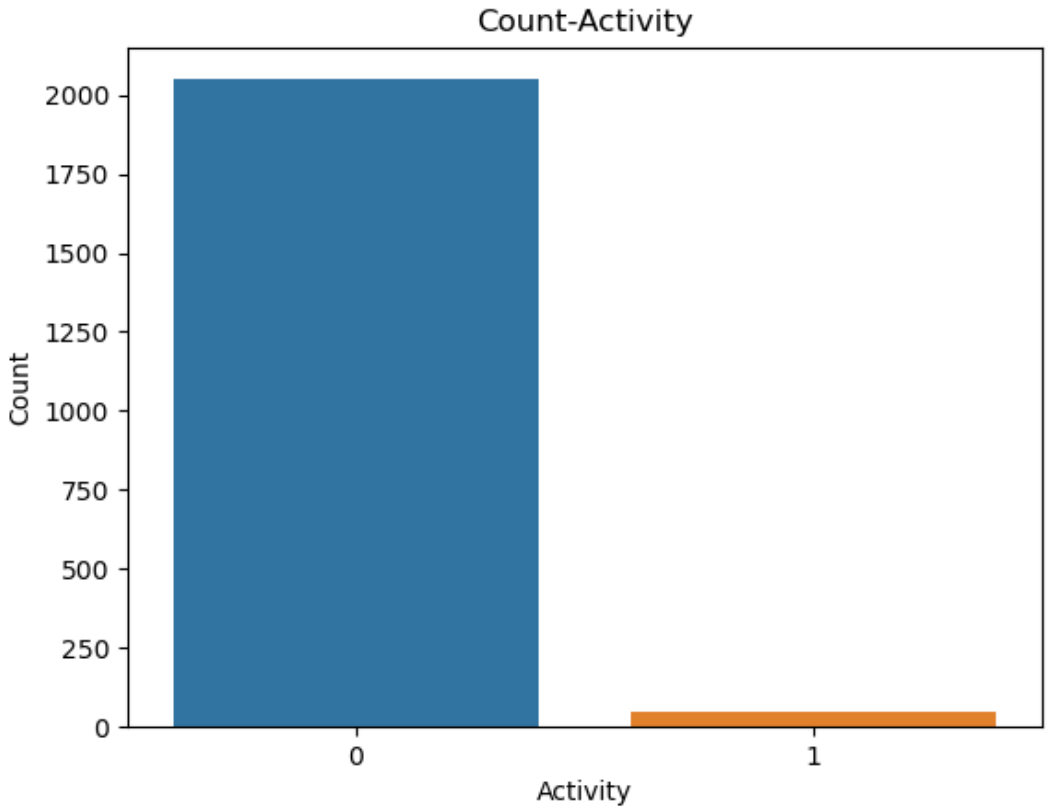
## 2. 任务描述

输入是若干化学分子的SMILES字符串和对应的活性值（0/1），通过建立模型预测化合物分子的抗性概率，输出结果。

## 3. 数据集介绍

任务给定的数据集有一个总的train.csv(带标签)和一个test.csv(未标注)的数据文件，与此同时，还有10个10折交叉验证集的文件夹(fold_0~fold_9)，每个文件夹下的数据被分为train, dev, test三部分。10折交叉验证数据集用于训练，最终对test.csv(未标注)进行标注。

值得一提的是，总的数据集中活性数据的比例非常低，如下图所示：




## 4. 评价指标

任务主要的评价指标是PRC-AUC和ROC-AUC，这两个指标的含义在此不作赘述；在本地训练的时候可以得到10折数据集的ROC-AUC和PRC-AUC得分，但是网站的排名主要依据是未标注的数据集的PRC-AUC得分。

# 二、成员分工

我们的分工如下：聂希瑞负责传统方法的探索和实验，包括写完整的代码、必要的文档（见文档第二部分），金辰哲负责图网络方法的部分，包括写代码、必要的文档（见文档第三部分）。最终的文档整合由聂希瑞完成。

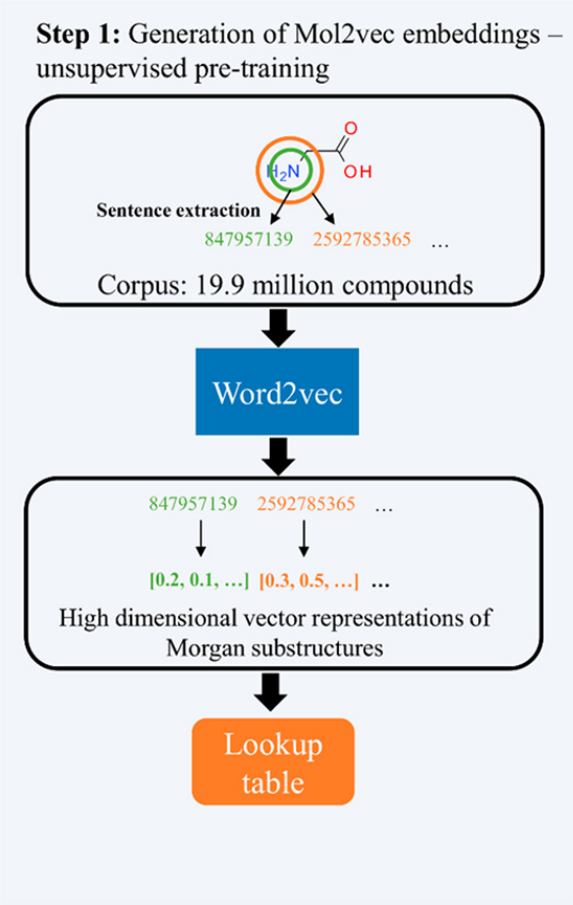
# 三、传统方法

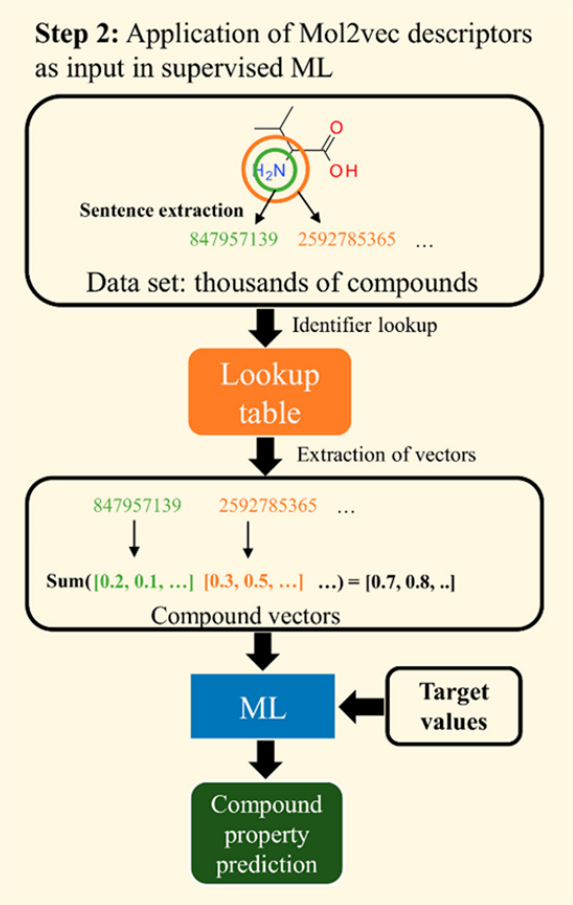
在此次实验中，我们对传统的一些方法进行了尝试，在这里我仅仅写出比较成功的两次尝试的结果，分别是mol2vec加上逻辑斯蒂回归和随机森林的模型。

## 1. mol2vec+LogisticRegression

### 1.1 mol2vec

mol2vec的大致思想是把分子的部分构造当成单词，把一个化合物当作一整个句子。利用NLP对句子处理的经验，在化合物的文本集上得到部分构造的高维嵌入形式。




mol2vec的步骤可以分为两步：

- 无监督学习和预训练部分

  使用Zinc和Chembl两个数据库，合并以后去掉重复的构造留下能被rdkit读出来的化合物，设定了一系列的筛选条件之后得到了预训练的数据集。这两个数据集的特征是化合物相对比较多元。接下里就是摩根分子指纹的编码和fit上mol2vec模型

- 使用部分：

  在之前的字典里找之前用无监督方式计算出来的各个部分构造的向量，然后整个分子可以用这些个向量的和来表示，作为x，之后就可以用监督学习的方式再去构建QSPR / QSAR的模型了。

对于一些药物活性分类的任务上mol2vec的表现，在同时用随机森林的时候，mol2vec比分子指纹稍低。在Tox21，相对于分子图使用CNN，mol2vec使用随机森林就超过了分子图。

### 1.2 Logistic Regression

逻辑斯蒂回归是一个经典的线性回归中的判别式分类模型，具体内容不必在此赘述。主要是通过sigmoid函数将结果映射到0/1，然后计算每个元素属于各个类别的概率值，以最大概率值决定其所属类别。在训练过程中，通过最大似然估计进行参数求解并且修改权重和偏置参数。数学公式在此不表。

### 1.3 API

关于mol2vec需要手动安装github上的仓库，并且加载其训练好的模型对SMILES字符串进行处理，当然也可以按照文档进行重新训练，但是这里是直接使用预训练好的模型进行后续的处理。

逻辑回归模型比较简单，作业1的时候就手动实现过该模型，但此处为了方便，选择了sklearn带有的LogisticRegression模型。

## 2. mol2vec+RandomForest

### 2.1 Random Forest

这一部分只需要简单介绍一下随机森林模型即可。随机森林就是通过集成学习的思想将多棵树集成的一种算法，它的基本单元是决策树，而它的本质属于机器学习的一大分支——集成学习（Ensemble Learning）方法。随机森林能够有效地运行在大数据集你上，处理具有高维特征的输入样本，不需要进行降维。它依靠于决策树的投票选择来决定最后的分类结果。

## 3. 代码实现

传统方法的两个实现结果见文件夹`niexirui`下main.py文件中相关部分，代码比较容易理解。

## 4. 运行

安装好相应的依赖后，只需要`python main.py`即可做完所有训练和测试，前提是数据文件夹被放置在合适的位置且对应的路径变量正确。

运行成功以后，控制台会输出各个数据文件的训练结果：PRC-AUC和ROC-AUC得分，与此同时，对应文件夹下会生成若干图片，分别对应每个fold的PRC和ROC图以及每个fold的Count-Activity图。

## 5. 结果

在这里，我们分别对10折交叉训练集和完整的训练集(测试集占20%)进行了训练和测试；评价指标选取为ROC-AUC和PRC-AUC，得到的结果如下：

### 5.1 10-fold

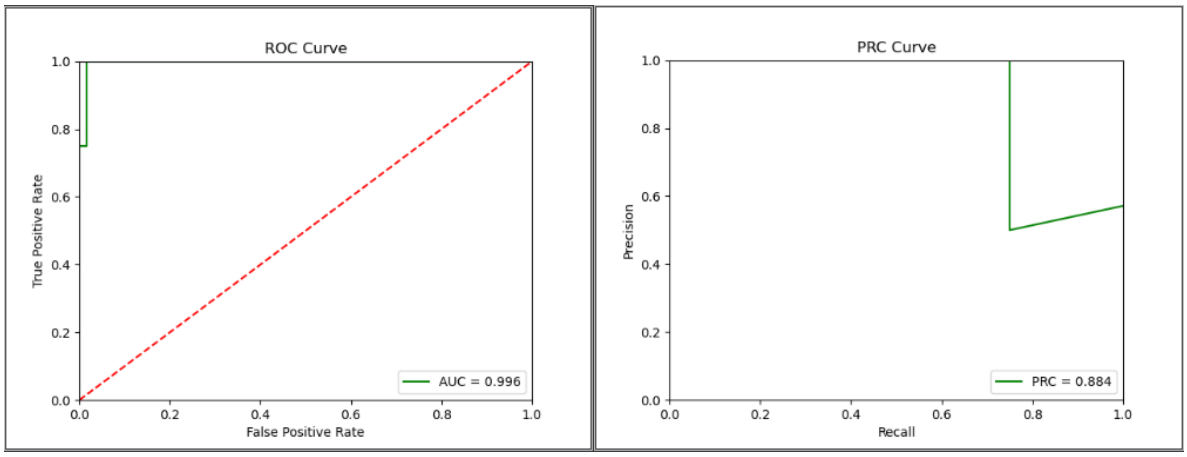
| model / fold | mol2vec+LogisticRegression | | mol2vec+RandomForest | |
|---|---|---|---|---|
| | ROC-AUC | PRC-AUC | ROC-AUC | PRC-AUC |
| 0 | 0.6439 | 0.4173 | 0.7061 | 0.281 |
| 1 | 0.8788 | 0.6252 | 0.9215 | 0.6237 |
| 2 | 0.8135 | 0.7558 | 0.7906 | 0.1702 |
| 3 | 0.5253 | 0.2643 | 0.7014 | 0.0469 |
| 4 | 0.8573 | 0.8386 | 0.9675 | 0.6039 |
| 5 | 0.8906 | 0.1362 | 0.947 | 0.4102 |
| 6 | 0.4891 | 0.0297 | 0.5543 | 0.0346 |
| 7 | 0.7391 | 0.0091 | 0.7488 | 0.0089 |
| 8 | 0.9962 | 0.8839 | 0.9937 | 0.8524 |
| 9 | 0.8693 | 0.1832 | 0.6993 | 0.0231 |
| TOTAL | 0.77031 | 0.41433 | 0.80302 | 0.30549 |

**结论：**

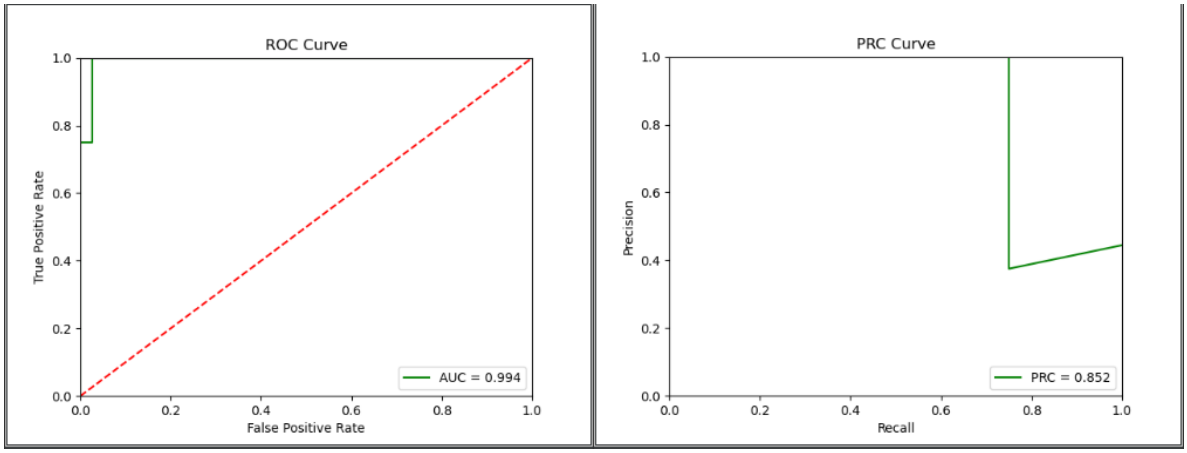
- 可以看到，在ROC-AUC的表现上，随机森林的效果要好于逻辑斯蒂回归；而对于主要的评价指标PRC-AUC来说，逻辑斯蒂回归明显表现更好。
- 总体的结果来说，这两种方法的表现都还算不错。

对表现比较好的fold，最终的ROC-AUC和PRC-AUC图如下：

- mol2vec+logisticregression




- mol2vec+randomforest




## 5.2 完整训练集

在完整训练集train.csv中，我们随机选取了80%作为训练集，20%作为测试集，得到的测试结果如下：

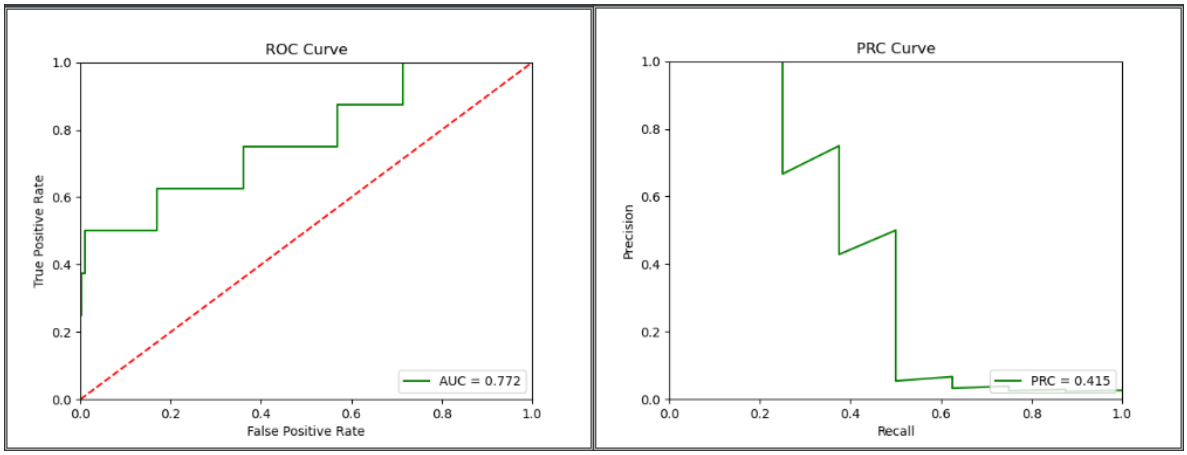
| model / fold | mol2vec+LogisticRegression | | mol2vec+RandomForest | |
|---|---|---|---|---|
| | ROC-AUC | PRC-AUC | ROC-AUC | PRC-AUC |
| train.csv | 0.7718 | 0.4151 | 0.813 | 0.3322 |

**结论：**

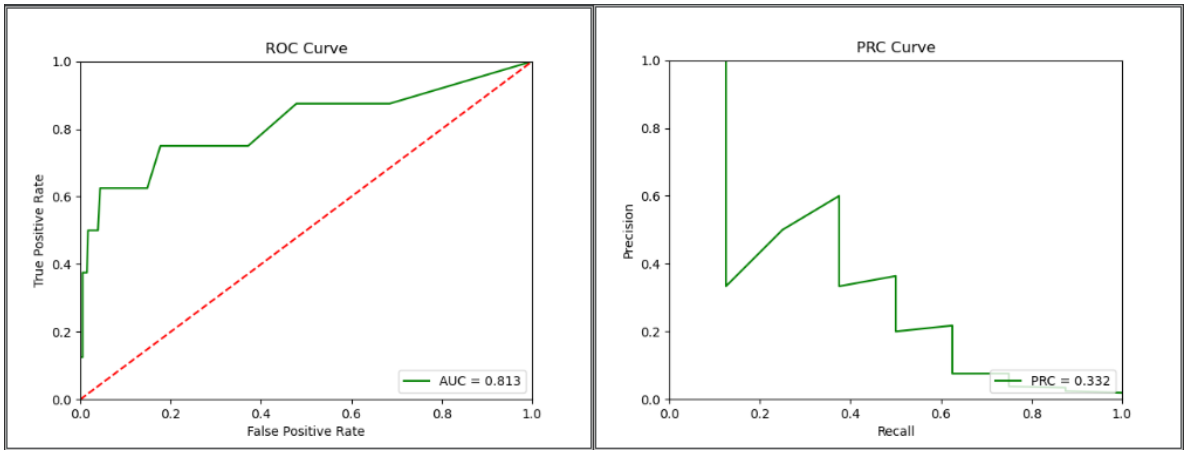
- 可以看到，对完整训练集的测试结果中，mol2vec+randomforest在ROC-AUC表现更好，而逻辑斯蒂回归啧在PRC-AUC表现更好。

对完整数据集的测试结果图示如下：

- mol2vec+logisticregression




- mol2vec+randomforest




# 四、图卷积网络方法 (GCNs)

## 1. Convolutional Layers

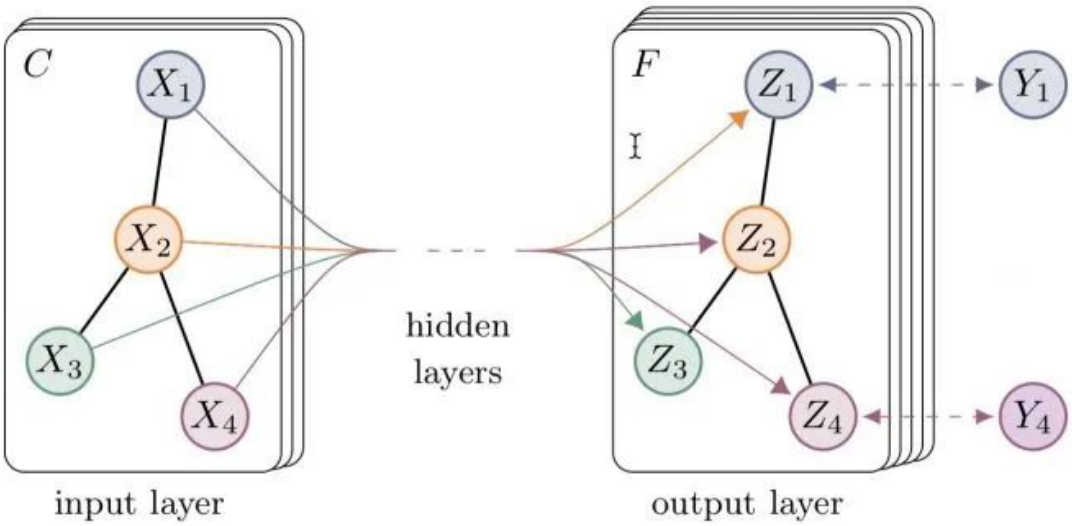
在卷积网络层方法中，我们使用了GCN, GAT, GIN, SAGECov四种模型进行实验。

### 1.1 GCN (Graph Convolutional Network)

- 原理

图卷积神经网络的算子可以表示为：$h_i^{l+1} = \sigma(\sum_{j \in N_i} \frac{1}{c_{ij}} h_j^l w_{R_j}^l)$；其中$h_i^l$表示节点$i$在第$l$层的特征表达，$c_{ij}$是归一化因子；$N_i$表示节点$i$的邻居，包括自身；$R_i$表示节点$i$的类型；$w_{R_i}$表示$R_i$类型的节点的变换权重参数。权重使用Glorot均匀初始化，偏差被初始化为0。

图卷积算法的过程主要为发射，接收和变换三步，具体含义在此不做赘述。总之，GCN能够同时对节点特征信息与结构信息进行端对端学习，在节点分类与边预测等任务上，在公开数据集上的效果要远优于其他方法。




(a) Graph Convolutional Network

- API

```
dgl.nn.pytorch.conv.GraphConv(in_feats, out_feats, norm='both', weight=True,
bias=True, activation=None)
forward(graph, feat, weight=None)
```

## 1.2 GAT (Graph Attention Network)

- 原理

注意力机制是通过注意力分数来实现的，它可以作用在RNN的每个序列上使其具有不同关注度，每个注意力分树代表当前项所被分配的注意力权重。GAT正是在GCN的基础之上引入了注意力机制，从而克服了先前GCN的一些短板。比起GCN，GAT具有更高效的运行效率，可以并行运算；针对不同degree的节点，可以运用不同的权重与之对应；并且可以进行归纳学习，不需要重新训练模型。

公式描述是：$h_i^{(l+1)} = \sum_{j \in Ni} \alpha_{i,j} w_{R_i}^l h_j^l$；其中$\alpha_{i,j}$是节点$i, j$之间的attention score，$\alpha_{i,j} = softmax_i(e_{i,j}^l), e_{i,j}^l = LeakyReLU(\overrightarrow{a}^T [wh_i || wh_j])$。

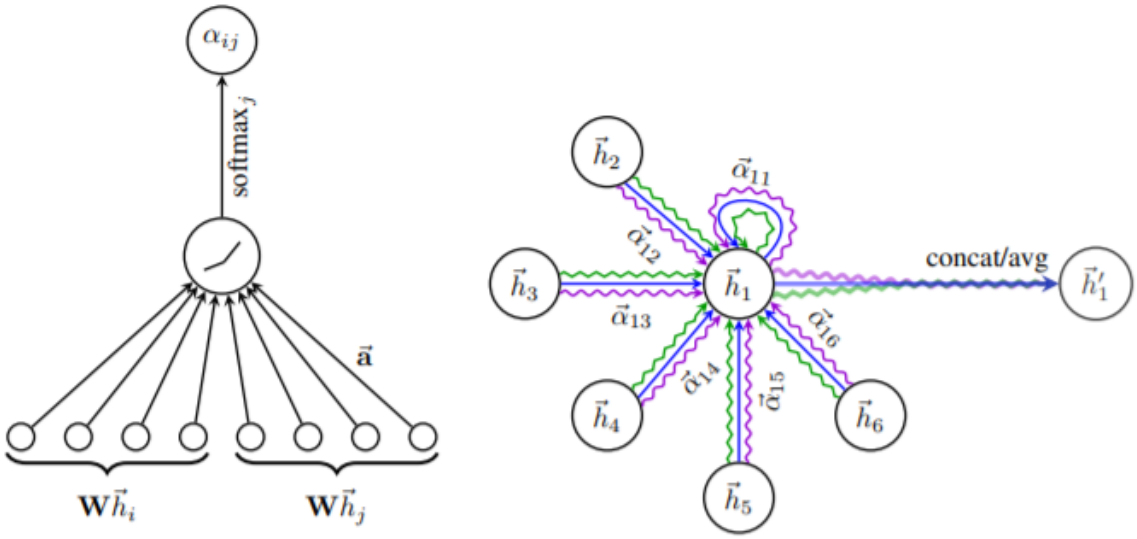



Figure 1: **Left:** The attention mechanism $a(\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j)$ employed by our model, parametrized by a weight vector $\vec{\mathbf{a}} \in \mathbb{R}^{2F'}$, applying a LeakyReLU activation. **Right:** An illustration of multi-head attention (with $K = 3$ heads) by node 1 on its neighborhood. Different arrow styles and colors denote independent attention computations. The aggregated features from each head are concatenated or averaged to obtain $\vec{h}'_1$.

- API

```
dgl.nn.pytorch.conv.GATConv(in_feats, out_feats, num_heads, feat_drop=0.0,
attn_drop=0.0, negative_slope=0.2, residual=False, activation=None)
forward(graph, feat)
```

## 1.3 GIN (Graph Isomorphism Network)

- 原理

图同构网络是GNNs中鉴别能力最强的一种网络，公式描述为：
$h_i^{l+1} = f_\Theta((1+\epsilon)h_i^l + aggregate(\{h_j^l, j \in N_i\})$，其中aggregate_type = sum, max, mean; 关于GIN的更多详细介绍在此不做赘述。

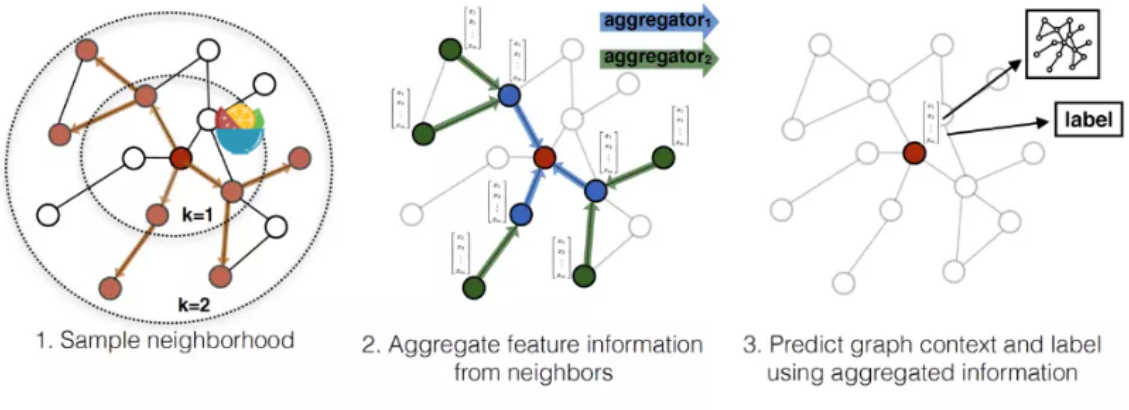
- API

```
dgl.nn.pytorch.conv.GINConv(apply_func, aggregator_type, init_eps=0,
learn_eps=False)
forward(graph, feat)
```

## 1.4 SageGCN

- 原理

SageGCN的基本思想是去学习一个节点的信息是怎么通过其邻居节点的特征聚合而来的。 学习到了这样的“聚合函数”，加上本身就已知各个节点的特征和邻居关系，就可以很方便地得到一个新节点的表示。SageGCN进行embedding的时候使用的是前向传播算法，聚合K次，每次把上一层的各个node的特征再聚合一次。




SageGCN的过程可以描述为

1. 先对邻居随机采样，降低计算复杂度（图中一跳邻居采样数=3，二跳邻居采样数=5）
2. 生成目标节点emebedding：先聚合2跳邻居特征，生成一跳邻居embedding，再聚合一跳邻居embedding，生成目标节点embedding，从而获得二跳邻居信息。（后面具体会讲）。
3. 将embedding作为全连接层的输入，预测目标节点的标签。

在上述过程中，使用的聚合函数为：

$$h_{N_i}^{l+1} = aggregate(\{h_j^l, \forall j \in N_i\})$$
$$h_i^{l+1} = \sigma(W \cdot concat(h_i^l, h_{N_i}^{l+1} + b))$$
$$h_i^{l+1} = norm(h|i^l)$$

- API

```
dgl.nn.pytorch.conv.SAGEConv(in_feats, out_feats, aggregator_type,
feat_drop=0.0, bias=True, norm=None, activation=None)
forward(graph, feat)
```

# 2. Dense Convolutional Layers

在Dense Convolutional Layers这组模型中，我们选取了Dense GCN, Dense SageGCN, Dense ChebGCN三种模型进行实验。 Conv Layers与Dense Conv Layers的区别在于forward时第一个参量是DGL图还是应用图卷积的图的邻接矩阵。在构建模型时两者(model.py中class Model与models.py中class Models)的区别在于对smiles字符串进行处理的process函数，其中一个是构图另一个是构建图卷积的邻接矩阵。

## 2.1 Dense GCN

- 原理

DenseGCN为每层的图卷积衔接了先前所有中间层的信息，从而可以有效融合多级别的特征，为梯度的流动提供了良好的通道，进一步促进了特征复用，缓解了梯度消失问题。图卷积网络层的图结构由邻接矩阵给出。

$$\begin{aligned}\mathcal{G}_{l+1} &= \mathcal{H}(\mathcal{G}_l, \mathcal{W}_l) \\ &= \mathcal{T}(\mathcal{F}(\mathcal{G}_l, \mathcal{W}_l), \mathcal{G}_l) \\ &= \mathcal{T}(\mathcal{F}(\mathcal{G}_l, \mathcal{W}_l), ..., \mathcal{F}(\mathcal{G}_0, \mathcal{W}_0), \mathcal{G}_0).\end{aligned}$$

- API

```
dgl.nn.pytorch.conv.DenseGraphConv(in_feats, out_feats, norm='both', bias=True,
activation=None)
forward(adj, feat)
```

## 2.2 Dense SageGCN

- 原理

  同理，相比于SageGCN模型，Dense加强了梯度的流动，使得网络可以更好地融合多级别的特征。

- API

```
dgl.nn.pytorch.conv.DenseSAGEConv(in_feats, out_feats, feat_drop=0.0, bias=True,
norm=None, activation=None)
forward(adj, feat)
```

## 2.3 Dense Cheb

- 原理

契比雪夫频谱卷积网络是通过图的拉普拉斯矩阵的特征值所构成的对角矩阵的契比雪夫多项式所组成的,其卷积滤波器的定义为

$$\mathbf{g}_\theta = \sum_{k=0}^{K-1} \theta_k T_k(\hat{\mathbf{\Lambda}})$$

其中，$\hat{\mathbf{\Lambda}} = 2\mathbf{\Lambda}/\lambda_{\max} - \mathbf{I}_N$ .契比雪夫多项式可递归的定义为

$$T_k(\mathbf{x}) = 2\mathbf{x}T_{k-1}(\mathbf{x}) - T_{k-1}(\mathbf{x}), \quad T_0(\mathbf{x}) = 1, T_1(\mathbf{x}) = \mathbf{x}.$$

从而,在上述卷积滤波器的定义下,图信号 $\mathbf{x}$ 的图卷积可定义为

$$\begin{aligned}\mathbf{x} * \mathbf{g}_\theta &= \mathbf{U}(\sum_{k=0}^{K-1} \theta_k T_k(\hat{\mathbf{\Lambda}}))\mathbf{U}^T\mathbf{x} \\ &= \sum_{k=0}^{K-1} \theta_k T_k(\hat{\boldsymbol{L}})\mathbf{x}\end{aligned}$$

其中，$\hat{\boldsymbol{L}} = 2\mathbf{L}/\lambda_{max} - \mathbf{I}_N$. 从上面的表达式我们可以看出,ChebNet不需要计算拉普拉斯特征矩阵的傅里叶基矩阵,这样可以降低计算的复杂度.并且由于卷积操作只针对单个结点,因而其是一个局部卷积操作.

- API

```
dgl.nn.pytorch.conv.DenseChebConv(in_feats, out_feats, k, bias=True)
forward(adj, feat, lambda_max=None)
```

# 3. 代码实现

提交`*.py`代码部分包含main, model, dense_model, train, evaluate五个`*.py`文件。`main.py`中load_data函数为读入fold_x文件夹中三个`*.csv`文件，process_fold函数为执行10个fold_x,对于每个fold in range(10)，先用load_data函数读取train, dev, test三组数据，并将smiles字符串转化为DGL图，与对应的activity值一起转化为程序调用中的data数据。sampler为通过Sampler和Randomsampler从train_data数据中选取的部分符合设定activity=1比例的数据集，并将其作为训练数据集，利用train_model函数对每个fold最多进行max_iteration次训练，每次训练抽取batch_per_epoch组数据，累加所有随机抽取的数据组的loss，由此对模型优化，训练的结果利用evaluate_model 函数对测试集进行测试，得到ROC-AUC / PRC-AUC。model与dense_model里分别包含GIN, GCN, GAT, SAGE, DenseGraph, DenseSAGE, DenseCheb两种七类模型，两种model的区别在于图构建（process函数）方式不同。GNN模型中包含GNN模型初始化，构建smiles字符串对应化学式的DGL图和每个原子的特征向量（利用atom_feature函数），forward以及predict部分。其中forward部分对每组batch中的数据的图邻接矩阵利用预先构建的 GNN 模块获得标准的两层GNN模型，获得的结果用criterion（CrossEntropyLoss）预测并统计loss。

输入为main.py目录下train_cv中10个fold，输出为main.py目录下output.txt文件。输出文件每个cycle在输出文件中显示loss, tn, tp, fn, fp，每个fold训练结束后对测试集评估，并输出ROC-AUC和PRC-AUC的值，之后为对十组ROC-AUC/PRC-AUC的分析。输出文件最后统计每个fold_x运行的时间和程序运行总时间。

# 4. 运行

在main.py 函数最后一行调用process_fold函数处修改想要选择的模型(“gcn”, ”gat”, “gin”, “sage”,“dense_gcn”, “dense_sage”, “dense_cheb”)。在命令行执行python main.py或者使用PyCharm运行main.py即可。需预装torch, dgl, rdkit, sklearn等库。

# 5. 结果

在这部分，我们只拿出了10-fold的测试结果进行对比，对比的指标有三个：ROC-AUC, PRC-AUC和运行时间。以下是详细的结果：

## 5.1 模型间对比

| model / fold | GCN | | | DenseGCN | | |
|---|---|---|---|---|---|---|
| | ROC-AUC | PRC-AUC | Runtime(s) | ROC-AUC | PRC-AUC | Runtime(s) |
| 0 | 0.791 | 0.354 | 533 | 0.702 | 0.157 | 133 |
| 1 | 0.891 | 0.406 | 725 | 0.918 | 0.707 | 90 |
| 2 | 0.792 | 0.215 | 356 | 0.835 | 0.515 | 84 |
| 3 | 0.521 | 0.022 | 746 | 0.718 | 0.274 | 90 |
| 4 | 0.863 | 0.711 | 790 | 0.936 | 0.689 | 222 |
| 5 | 0.542 | 0.343 | 234 | 0.877 | 0.289 | 86 |
| 6 | 0.427 | 0.02 | 777 | 0.21 | 0.008 | 223 |
| 7 | 0.454 | 0.004 | 644 | 0.106 | 0.003 | 234 |
| 8 | 0.981 | 0.513 | 661 | 0.981 | 0.531 | 228 |
| 9 | 0.882 | 0.369 | 508 | 0.812 | 0.37 | 90 |
| TOTAL | 0.715±0.195 | 0.296±0.22 | 5974 | 0.709±0.289 | 0.354±0.242 | 1479 |

| model / fold | SAGE | | | DenseSAGE | | |
|---|---|---|---|---|---|---|
| | ROC-AUC | PRC-AUC | Runtime(s) | ROC-AUC | PRC-AUC | Runtime(s) |
| 0 | 0.791 | 0.311 | 683 | 0.541 | 0.267 | 71 |
| 1 | 0.804 | 0.412 | 750 | 0.916 | 0.676 | 80 |
| 2 | 0.727 | 0.524 | 495 | 0.868 | 0.282 | 164 |
| 3 | 0.648 | 0.06 | 446 | 0.739 | 0.276 | 87 |
| 4 | 0.938 | 0.735 | 765 | 0.939 | 0.698 | 221 |
| 5 | 0.611 | 0.351 | 751 | 0.736 | 0.672 | 90 |
| 6 | 0.604 | 0.075 | 717 | 0.279 | 0.009 | 218 |
| 7 | 0.826 | 0.014 | 506 | 0.121 | 0.003 | 229 |
| 8 | 0.766 | 0.431 | 1880 | 0.997 | 0.908 | 207 |
| 9 | 0.888 | 0.095 | 1152 | 0.853 | 0.069 | 85 |
| TOTAL | 0.760±0.107 | 0.301±0.225 | 8144 | 0.699±0.280 | 0.386±0.310 | 1451 |

| model / fold | GAT | | | GIN | | | DenseCheb | | |
|---|---|---|---|---|---|---|---|---|---|
| | ROC-AUC | PRC-AUC | Runtime(s) | ROC-AUC | PRC-AUC | Runtime(s) | ROC-AUC | PRC-AUC | Runtime(s) |
| 0 | 0.63 | 0.184 | 860 | 0.79 | 0.479 | 456 | 0.614 | 0.183 | 469 |
| 1 | 0.91 | 0.521 | 2180 | 0.927 | 0.635 | 537 | 0.915 | 0.745 | 438 |
| 2 | 0.66 | 0.219 | 2501 | 0.766 | 0.197 | 539 | 0.701 | 0.512 | 444 |
| 3 | 0.403 | 0.033 | 2519 | 0.657 | 0.103 | 539 | 0.793 | 0.059 | 418 |
| 4 | 0.921 | 0.468 | 1594 | 0.857 | 0.728 | 535 | 0.95 | 0.848 | 409 |
| 5 | 0.584 | 0.344 | 2624 | 0.505 | 0.342 | 578 | 0.781 | 0.392 | 362 |
| 6 | 0.845 | 0.044 | 2475 | 0.714 | 0.025 | 538 | 0.29 | 0.009 | 312 |
| 7 | 0.884 | 0.02 | 2534 | 0.908 | 0.025 | 560 | 0.319 | 0.004 | 326 |
| 8 | 0.891 | 0.76 | 2579 | 0.978 | 0.497 | 562 | 0.989 | 0.656 | 324 |
| 9 | 0.948 | 0.118 | 4142 | 0.861 | 0.073 | 560 | 0.645 | 0.019 | 328 |
| TOTAL | 0.767±0.176 | 0.271±0.235 | 24008 | 0.796±0.135 | 0.310±0.249 | 5403 | 0.700±0.230 | 0.343±0.313 | 3829 |

**结论：**

- 运行时间比较：DenseSAGE < DenseGraph < DenseCheb < GCN ≈ GIN< SAGE << GAT
- ROC-AUC 数值比较：GIN优于GAT, SAGE优于DenseSAGE优于GCN,DenseGraph,DenseCheb，误差GCN < DenseCheb < DenseGraph
- PRC-AUC数值比较：DenseSAGE优于DenseCheb,DenseGraph优于GIN, SAGE, GCN优于GAT，误差DenseGraph < DenseCheb
- 整体比较来看，Conv Layers计算的ROC-AUC 和 PRC-AUC的值比Dense Conv Layers更高，且误差较小，但用时更长。

## 5.2 模型内对比

### 5.2.1 GIN

| model / fold | GIN(aggregator_type=sum) | | | GIN(aggregator_type=mean) | | | GIN(aggregator_type=max) | | |
|---|---|---|---|---|---|---|---|---|---|
| | ROC-AUC | PRC-AUC | Runtime(s) | ROC-AUC | PRC-AUC | Runtime(s) | ROC-AUC | PRC-AUC | Runtime(s) |
| 0 | 0.79 | 0.479 | 456 | 0.655 | 0.083 | 256 | 0.843 | 0.21 | 801 |
| 1 | 0.927 | 0.635 | 537 | 0.855 | 0.581 | 631 | 0.871 | 0.429 | 659 |
| 2 | 0.766 | 0.197 | 539 | 0.717 | 0.331 | 452 | 0.772 | 0.334 | 1866 |
| 3 | 0.657 | 0.103 | 539 | 0.605 | 0.082 | 517 | 0.778 | 0.065 | 1844 |
| 4 | 0.857 | 0.728 | 535 | 0.926 | 0.279 | 334 | 0.978 | 0.833 | 849 |
| 5 | 0.505 | 0.342 | 578 | 0.562 | 0.345 | 242 | 0.666 | 0.381 | 809 |
| 6 | 0.714 | 0.025 | 538 | 0.601 | 0.055 | 683 | 0.708 | 0.045 | 802 |
| 7 | 0.908 | 0.025 | 560 | 0.507 | 0.005 | 686 | 0.758 | 0.01 | 355 |
| 8 | 0.978 | 0.497 | 562 | 0.789 | 0.643 | 646 | 0.912 | 0.403 | 827 |
| 9 | 0.861 | 0.073 | 560 | 0.846 | 0.067 | 645 | 0.928 | 0.11 | 802 |
| TOTAL | 0.796±0.135 | 0.310±0.249 | 5403 | 0.706±0.135 | 0.247±0.216 | 5091 | 0.821±0.096 | 0.282± 0.237 | 9613 |

**结论：**

- GINConv模型取aggregator_type=mean最优。

### 5.2.2 SageGCN

| model | SAGE(aggregator_type=mean) | | | SAGE(aggregator_type=gcn) | | | SAGE(aggregator_type=pool) | | |
|---|---|---|---|---|---|---|---|---|---|
| | ROC-AUC | PRC-AUC | Runtime(s) | ROC-AUC | PRC-AUC | Runtime(s) | ROC-AUC | PRC-AUC | Runtime(s) |
| 0 | 0.791 | 0.311 | 683 | 0.716 | 0.345 | 318 | 0.796 | 0.355 | 789 |
| 1 | 0.804 | 0.412 | 750 | 0.84 | 0.309 | 222 | 0.921 | 0.74 | 1673 |
| 2 | 0.727 | 0.524 | 495 | 0.736 | 0.148 | 287 | 0.662 | 0.511 | 2141 |
| 3 | 0.648 | 0.06 | 446 | 0.601 | 0.026 | 366 | 0.63 | 0.038 | 1466 |
| 4 | 0.938 | 0.735 | 765 | 0.884 | 0.501 | 1583 | 0.886 | 0.687 | 956 |
| 5 | 0.611 | 0.351 | 751 | 0.64 | 0.347 | 870 | 0.599 | 0.071 | 930 |
| 6 | 0.604 | 0.075 | 717 | 0.578 | 0.075 | 1548 | 0.516 | 0.014 | 1991 |
| 7 | 0.826 | 0.014 | 506 | 0.667 | 0.007 | 552 | 0.696 | 0.008 | 1385 |
| 8 | 0.766 | 0.431 | 1880 | 0.949 | 0.406 | 940 | 0.99 | 0.729 | 839 |
| 9 | 0.888 | 0.095 | 1152 | 0.943 | 0.402 | 309 | 0.826 | 0.053 | 424 |
| TOTAL | 0.760±0.107 | 0.301±0.225 | 8144 | 0.755±0.132 | 0.256±0.168 | 6993 | 0.752±0.147 | 0.321±0.303 | 12596 |

**结论：**

- SAGEConv模型取aggregator_type=gcn最优。

# 五、总结

1. 在经过传统的两种模型和图网络的七种模型之间进行对比之后，我们发现总体上而言**传统模型取得的效果反而要好于这些图卷积网络模型。**
2. 我们的实验仍然存在不少值得改进的地方：比如实际模型训练过程中的一些调参和优化可以继续改进(实际上并未怎么优化)。
3. 受限于数据集的缺陷（正例太少），为了取得更好的训练效果——即提取出更多正例的特征从而优化模型，我们可以尝试对数据集进行一次过滤，提高正例所占据的比例，从而得到一个更好的模型；受限于时间关系，我们并未做完这个尝试。
4. 目前得到的所有结果对比都是基于已经有标注的10折交叉验证集和完整的train.csv训练集，对于网站上的要求（将标注好的test.csv发送至网站进行排名）我们并未满足。不过我们认为在已有标注的训练集上得到的结果仍然具有很高的参考价值，因此我们认为我们得到的结果是可信的。

# 六、参考文献


[1] https://blog.csdn.net/qq_27075943/article/details/106623059 #【GNN】MPNN：消息传递神经网络

[2] https://blog.csdn.net/Frank_LJiang/article/details/95194733 #图神经网络（GNN）总结（GCN）

[3] https://zhuanlan.zhihu.com/p/89503068 #一文读懂图卷积GCN

[4] http://rdkit.chenzhaoqiang.com/basicManual.html #RDKit中文教程

[5] https://blog.csdn.net/qq_33148001/article/details/105192816 #DGL官方教程--API--dgl.nn

[6] https://pubs.acs.org/doi/10.1021/acs.jcim.7b00616Mol2vec: Unsupervised Machine Learning Approach with Chemical Intuition