心跳信号分类预测

1、赛题

赛题以预测心电图心跳信号类别为任务，该数据来自某平台心电图数据记录，总数据量超过20万，主要为1列心跳信号序列数据，其中每个样本的信号序列采样频次一致，长度相等。为了保证比赛的公平性，将会从中抽取10万条作为训练集，2万条作为测试集。

字段表

Field	Description
id	为心跳信号分配的唯一标识
heartbeat_signals	心跳信号序列
label	心跳信号类别（0、1、2、3）

2、评判标准

需提交4种不同心跳信号预测的概率，选手提交结果与实际心跳类型结果进行对比，求预测的概率与真实值差值的绝对值（越小越好）。

3、机器学习

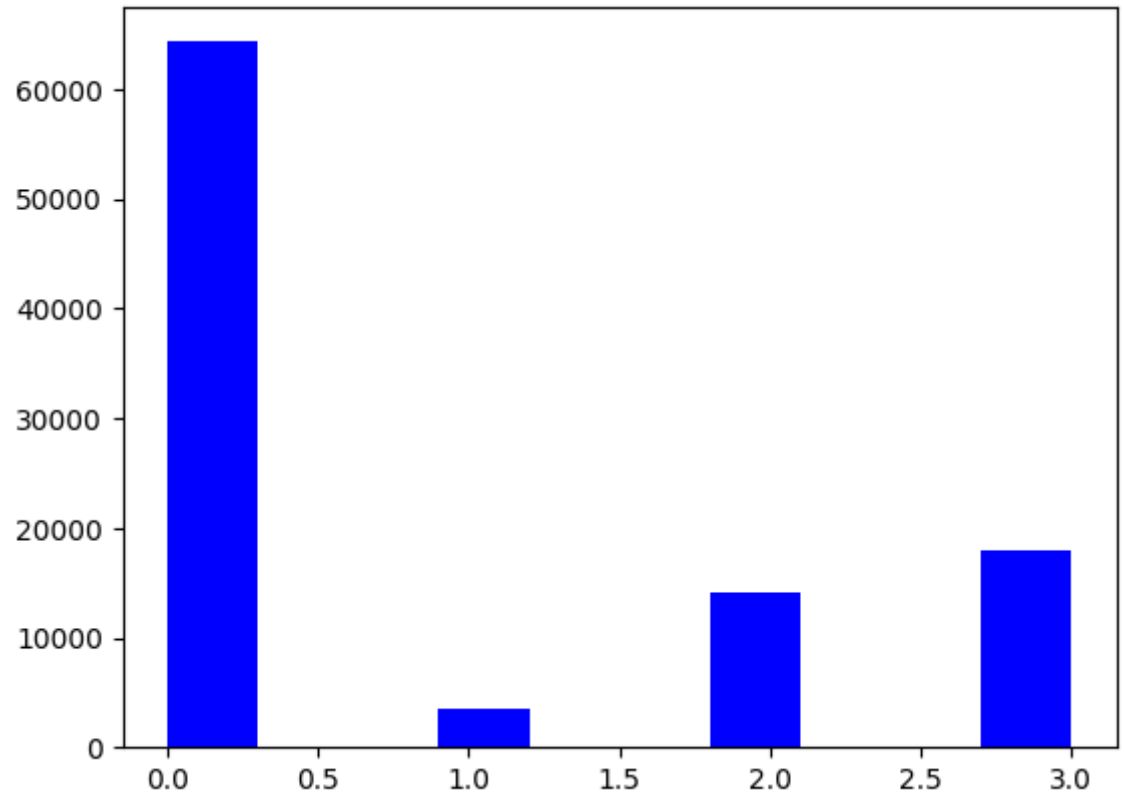
将心跳序列转为205个特征：

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import xgboost as xgb

# 步骤1: 数据预处理
# 分割心跳信号序列为数值列表
df['heartbeat_signals'] = df['heartbeat_signals'].apply(lambda x: [float(i) for
i in x.split(',')])

# 将列表转换为DataFrame的列
max_length = max(df['heartbeat_signals'].apply(len))  # 找到最长序列的长度
signals_df = pd.DataFrame(df['heartbeat_signals'].tolist()).fillna(0)  # 用0填充缺
失值
df = pd.concat([df, signals_df], axis=1)

# 删除原始的heartbeat_signals列
df = df.drop('heartbeat_signals', axis=1)
```

	id	label	0	1	2	3	4	5	6	7	...	195	196	197	198	199	200	201	202	203	204
0	0	0.0	0.991230	0.943533	0.764677	0.618571	0.379632	0.190822	0.040237	0.025995	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	1	0.0	0.971482	0.928969	0.572933	0.178457	0.122962	0.132360	0.094392	0.089575	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	2	2.0	1.000000	0.959149	0.701378	0.231778	0.000000	0.080698	0.128376	0.187448	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	3	0.0	0.975795	0.934088	0.659637	0.249921	0.237116	0.281445	0.249921	0.249921	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	4	2.0	0.000000	0.055816	0.261294	0.359847	0.433143	0.453698	0.499004	0.542796	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 207 columns

类别不平衡（估计1类是正常的）：




```
from imblearn.over_sampling import SMOTE

# 使用SMOTE进行过采样处理类别不平衡
smote = SMOTE()
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)
```

xgboost寻参并预测：

```
from sklearn.model_selection import KFold
from sklearn.metrics import log_loss
import numpy as np
import xgboost as xgb

kf = KFold(n_splits=5, shuffle=True, random_state=42)

abs_sum_scores = []  # 存储每一折的评分

for train_index, test_index in kf.split(X):
    X_resampled, X_test = X.iloc[train_index], X.iloc[test_index]
    y_resampled, y_test = y[train_index], y[test_index]
    # 使用找到的最佳参数设置模型
```

```
    best_params = {
    'colsample_bytree': 0.7692681476866446,
    'learning_rate': 0.0823076398078035,
    'max_depth': 6,
    'min_child_weight': 7,
    'n_estimators': 527,
    'subsample': 0.848553073033381,
    'use_label_encoder': False,
    'eval_metric': 'mlogloss'
   }

    # 初始化XGBoost模型
    model = xgb.XGBClassifier(**best_params)

    model.fit(X_resampled, y_resampled)

    # 预测概率
    y_pred_proba = model.predict_proba(X_test)

    # 计算abs-sum
    # 首先，我们需要将y_test转换为one-hot编码形式，以匹配y_pred_proba的格式
    y_test_one_hot = np.zeros((y_test.size, y_pred_proba.shape[1]))
    y_test_one_hot[np.arange(y_test.size), y_test.astype(int)] = 1

    abs_sum = np.abs(y_test_one_hot - y_pred_proba).sum() / y_test.size
    abs_sum_scores.append(abs_sum)

# 计算平均abs-sum分数
average_abs_sum = np.mean(abs_sum_scores)
print(f"Average Abs-Sum Score: {average_abs_sum}")

# Average Abs-Sum Score: 0.044173902911483334
```

换成LightGBM试试：

```
import lightgbm as lgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import log_loss
# 创建LightGBM数据集
train_data = lgb.Dataset(X_resampled, label=y_resampled)
test_data = lgb.Dataset(X_test, label=y_test, reference=train_data)

# 使用默认参数
params = {
    'objective': 'multiclass',
    'num_class': len(np.unique(y_resampled)),
    'metric': 'multi_logloss',
    'verbosity': -1
}

# 训练模型
gbm = lgb.train(params, train_data, num_boost_round=100, valid_sets=[test_data])

# 预测测试集
y_pred_proba = gbm.predict(X_test, num_iteration=gbm.best_iteration)
```

```
# 计算测试集的对数损失
log_loss_score = log_loss(y_test, y_pred_proba)
print(f"Test Log Loss: {log_loss_score}")

# 首先，将实际标签转换为one-hot编码形式
num_classes = np.unique(y_resampled).shape[0]  # 假设所有类别都出现在y_resampled中
y_test_one_hot = np.zeros((y_test.shape[0], num_classes))
y_test_one_hot[np.arange(y_test.shape[0]), y_test.astype(int)] = 1

# 计算预测概率与实际标签之间的abs-sum
abs_sum = np.sum(np.abs(y_pred_proba - y_test_one_hot)) / y_test.shape[0]

print(f"Average Abs-Sum: {abs_sum}")

# Test Log Loss: 0.02038390054920139
# Average Abs-Sum: 0.035310493539996254
```

4、深度学习（CNN）

```
class DeeperCNNModel(nn.Module):
    def __init__(self, num_classes=4):
        super(DeeperCNNModel, self).__init__()
        # 初始卷积层与之前相同
        self.layer1 = nn.Sequential(
            nn.Conv1d(1, 32, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm1d(32),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2)
        )
        self.layer2 = nn.Sequential(
            nn.Conv1d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm1d(64),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2)
        )
        # 添加额外的卷积层
        self.layer3 = nn.Sequential(
            nn.Conv1d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2)
        )
        self.layer4 = nn.Sequential(
            nn.Conv1d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2)
        )
        self.layer5 = nn.Sequential(
            nn.Conv1d(256, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm1d(512),
            nn.ReLU(),
```

```
            nn.MaxPool1d(kernel_size=2)
        )
        self.dropout = nn.Dropout(0.5)
        self.global_avg_pool = nn.AdaptiveAvgPool1d(1)
        # 调整全连接层以匹配最后一个卷积层的输出
        self.fc = nn.Linear(512, num_classes)

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.layer5(x)
        x = self.global_avg_pool(x)
        x = torch.flatten(x, 1)
        x = self.dropout(x)
        x = self.fc(x)
        return x


# 初始化模型、损失函数、优化器和混合精度缩放器
num_epochs = 30
model = DeeperCNNModel(num_classes=4).to(device)
criterion = nn.CrossEntropyLoss().to(device)
optimizer = optim.Adam(model.parameters(), lr=0.001)
scaler = GradScaler()
```

模型结构

卷积层（Convolutional Layers）

- `layer1`至`layer5`：这些层由一系列卷积层（`nn.Conv1d`）、批量归一化层（`nn.BatchNorm1d`）、ReLU激活函数（`nn.ReLU`），以及最大池化层（`nn.MaxPool1d`）组成。这些层逐步增加通道数（即特征数），从32增加到512，同时通过最大池化层减小特征维度，以提取并压缩特征。每个卷积层后面的批量归一化和ReLU激活函数有助于加速训练过程并提高模型的泛化能力。

全局平均池化层（Global Average Pooling Layer）

- `global_avg_pool`：使用`nn.AdaptiveAvgPool1d(1)`实现全局平均池化，将每个通道的特征映射到一个单一的数值上，以减少参数数量并减轻过拟合风险。

全连接层（Fully Connected Layer）

- `fc`：一个全连接层（`nn.Linear`），将全局平均池化后的特征转换为最终的分类输出。输出维度为`num_classes`，对应于数据集中的类别数。

前向传播（Forward Pass）

- `forward`**方法**：定义了数据通过模型的前向传播路径。数据依次通过五个卷积块（`layer1`至`layer5`）、全局平均池化层（`global_avg_pool`），然后展平（`torch.flatten`），通过Dropout层（减少过拟合风险），最后通过一个全连接层来预测最终的类别。

效果目前和xgboost差不多，但是top方案是用CNN做的。。。。。