```python
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
import torch
from torch.utils.data import Dataset, DataLoader
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```python
df = pd.read_csv('data_cleaned.csv')
df['platform_order_time_date']=pd.to_datetime(df['platform_order_time_date'])
df=df.sort_values(by='platform_order_time_date')
```

```python
time_interval = '5T'  # 'T' 代表分钟df['platform_order_time_date']
df['order_time_interval']=df['platform_order_time_date'].dt.floor(time_interval)
```

```python
all_intervals = pd.date_range(start=df['order_time_interval'].min().floor('5T'),
                              end=df['order_time_interval'].max().floor('5T'),
                              freq='5T')
all_h3 = df['H3_Index'].unique()

# 创建所有可能的组合
full_index = pd.MultiIndex.from_product(
    [all_intervals, all_h3],
    names=['order_time_interval', 'H3_Index']
)
print(len(all_intervals))
print(len(all_h3))
print(len(full_index ))
```

```
2309
1560
3602040
```

```python
grouped=df.groupby(['order_time_interval','H3_Index']).size()#.agg('count')
grouped = grouped.reindex(full_index , fill_value=0)
df = grouped.reset_index(name='count')
```

```python
le = LabelEncoder()
df['H3_Index_encoded'] = le.fit_transform(df['H3_Index'])
```

```python
class H3TimeSeriesDataset(Dataset):
    def __init__(self, sequences, labels):
        """
        sequences: list of tuples (seq_counts, seq_h3)
        labels: list of tuples (target_count, target_h3)
        """
        self.sequences = sequences
        self.labels = labels

    def __len__(self):
        return len(self.sequences)

    def __getitem__(self, idx):
        seq_counts, seq_h3 = self.sequences[idx]
        target_count= self.labels[idx]
        # 转换为张量
        seq_counts = torch.tensor(seq_counts, dtype=torch.float32).unsqueeze(-1)  # (T, 1)
        seq_h3 = torch.tensor(seq_h3, dtype=torch.long).unsqueeze(-1)  # (T, 1)
        # 合并特征，形状 (T, 2)
        input_seq = torch.cat((seq_counts, seq_h3), dim=1)
        # 目标
        target_count = torch.tensor(target_count, dtype=torch.float32)  # 泊松目标为float
        #target_h3 = torch.tensor(target_h3, dtype=torch.float32)  # 二分类标签
        return input_seq, target_count
```

```python
def create_sequences(df, T):
    sequences = []
    labels = []
    # 按 H3_Index_encoded 分组
    for h3, group in df.groupby('H3_Index_encoded'):
        group = group.sort_values('order_time_interval')
```

```
        counts = group['count'].values
        h3_encoded = group['H3_Index_encoded'].values
        # 创建长度为 T+1 的序列
        for i in range(len(counts) - T-1):
            seq_counts = counts[i:i+T]
            seq_h3 = h3_encoded[i:i+T]
            target_counts = counts[i+1:i+T+1]
            #target_h3 = h3_encoded[i+1:i+T+1]
            sequences.append((seq_counts, seq_h3))
            labels.append(target_counts)
    return sequences, labels
```

```
T = 10  # 序列长度
sequences, labels = create_sequences(df, T)
dataset = H3TimeSeriesDataset(sequences, labels)
batch_size = 512
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

```

class H3LSTMModel(nn.Module):
    def __init__(self, h3_vocab_size, h3_embedding_dim, hidden_dim, num_layers, max_count):
        super(H3LSTMModel, self).__init__()
        self.h3_embedding = nn.Embedding(num_embeddings=h3_vocab_size, embedding_dim=h3_embedding_dim)
        self.lstm = nn.LSTM(
            input_size=1 + h3_embedding_dim,
            hidden_size=hidden_dim,
            num_layers=num_layers,
            batch_first=True
        )
        self.fc_binary = nn.Linear(hidden_dim, 1)  # 输出二分类概率
        self.fc_count = nn.Linear(hidden_dim, max_count + 1)  # 输出Softmax概率分布

    def forward(self, x_counts, x_h3):
        """
        x_counts: (batch_size, T, 1)
        x_h3: (batch_size, T, 1)
        """
        # 嵌入 H3_Index
        x_h3 = x_h3.squeeze(-1)  # (batch_size, T)
        embedded_h3 = self.h3_embedding(x_h3)  # (batch_size, T, h3_embedding_dim)

        # 合并 count 和嵌入后的 H3_Index
        x = torch.cat((x_counts, embedded_h3), dim=2)  # (batch_size, T, 1 + h3_embedding_dim)

        # LSTM
        lstm_out, _ = self.lstm(x)  # lstm_out: (batch_size, T, hidden_dim)

        # 计算二分类概率
        binary_logits = self.fc_binary(lstm_out).squeeze(-1)  # (batch_size, T)
        binary_probs = torch.sigmoid(binary_logits)  # (batch_size, T)

        # 计算Softmax概率分布
        count_logits = self.fc_count(lstm_out)  # (batch_size, T, max_count +1)
        count_probs = F.softmax(count_logits, dim=-1)  # (batch_size, T, max_count +1)

        return binary_probs, count_probs

```

```
import torch
from torch.distributions import Poisson

def poisson_log_prob( lambda_,k):
    """
    计算泊松分布的对数概率。

    Args:
        k (torch.Tensor): 事件发生次数，形状可以是任意的非负整数张量。
        lambda_ (torch.Tensor): 平均发生次数，形状与 k 相同，且所有元素均为正数。

    Returns:
        torch.Tensor: 对数概率，形状与 k 相同。
    """
    # 创建泊松分布对象
    poisson_dist = Poisson(rate=lambda_)

    # 计算对数概率
```

```
        log_prob = poisson_dist.log_prob(k)

        return log_prob
```

```
device='cpu'
# model
model=H3LSTMModel(h3_vocab_size=len(all_h3), h3_embedding_dim=3, hidden_dim=32, num_layers=1, max_count=df['count'].max()).to(device)
# 二分类损失
criterion_binary = nn.BCELoss()

# 分类交叉熵损失
criterion_ce = nn.CrossEntropyLoss()

#mse loss
criterion_mse = nn.MSELoss()
# 优化器
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

```

```
import torch
print(torch.cuda.is_available())
```

```
False
```

```
def softargmax(count_probs, max_count):
    """
    计算 Softargmax，返回期望值作为连续预测。

    Args:
        count_probs (torch.Tensor): Softmax 输出概率，形状为 (batch_size, T, max_count + 1)。
        max_count (int): 最大计数 k 值。

    Returns:
        torch.Tensor: 预测的连续计数，形状为 (batch_size, T)。
    """
    # 创建一个包含所有计数类别的张量 k = [0, 1, 2, ..., max_count]
    k = torch.arange(0, max_count + 1, dtype=count_probs.dtype, device=count_probs.device).view(1, 1, -1)

    # 计算 Softargmax: sum_k P(k) * k
    soft_argmax = (count_probs * k).sum(dim=-1)

    return soft_argmax
```

```
# 训练模型
num_epochs = 100
max_count = df['count'].max()  # 从0到8

for epoch in range(num_epochs):
    model.train()
    epoch_loss = 0
    i=0
    for batch_inputs, batch_targets in dataloader:
        # 分离输入特征
        batch_counts = batch_inputs[:, :, 0].unsqueeze(-1).to(device)  # (batch_size, T, 1)
        batch_h3 = batch_inputs[:, :, 1].long().unsqueeze(-1).to(device)  # (batch_size, T, 1)

        # 目标
        y = batch_targets  # y_count: (batch_size,), y_binary: (batch_size,)
        y_count = y.to(device)  # (batch_size, T)
        y_binary = ((y==1)*1.0).to(device)  # (batch_size, T)
        #print( y_binary.shape)
        # 前向传播
        optimizer.zero_grad()
        binary_probs, count_probs = model(batch_counts, batch_h3)  # binary_probs: (batch_size, T), count_probs: (batc

        # 计算二分类损失
        loss_binary = criterion_binary(binary_probs, y_binary)

        soft_argmax=softargmax(count_probs, max_count)


        ##计算mse loss
        loss_mse=criterion_mse(soft_argmax,y_count)
        #$print(soft_argmax.shape)
        # 计算分类交叉熵损失
```

```
        # CrossEntropyLoss expects input of shape (N, C) and target of shape (N)
        # 这里将每个时间步视为一个独立的样本
        #count_probs_reshaped = count_probs.view(-1, max_count +1)  # (batch_size * T, max_count +1)
        #y_count_reshaped = y_count.view(-1)  # (batch_size * T,)
        #loss_ce = criterion_ce(count_probs_reshaped, y_count_reshaped)

        # 计算KL散度损失
        log_possion=-poisson_log_prob(soft_argmax,y_count).mean()
        #kl_div = kl_divergence_softmax_poisson(count_probs, y_count, max_count)  # (batch_size, T)
        #loss_kl = kl_div.mean()

        # 总损失
        loss = loss_binary + loss_mse + log_possion
        i+=1
        # 反向传播和优化

        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
        if i>=1000:
            break
    # 每10个epoch打印一次
    #if (epoch + 1) % 10 == 0:
    avg_loss = epoch_loss / len(dataloader)
    print(f"Epoch {epoch}, Loss: {avg_loss:.4f}")

```

```
Epoch [0], Loss: 3.9421
Epoch [1], Loss: 0.1952
Epoch [2], Loss: 0.1600
```

```
len(dataloader)
```

```
14004
```