面向 D-STGT 模型的 AIS 数据集分析、特征工程与实现方案

本报告由 AI 助手生成

2025 年 7 月 3 日

摘要

本报告旨在为“动态时空图 Transformer (D-STGT)”科研方案提供一份详尽的数据集分析、特征工程方法论及可执行的实现方案。报告将聚焦于方案中选定的美国国家海洋和大气管理局 (NOAA) AIS 数据集，系统性地阐述从原始数据获取到模型可用输入的全流程，并提供完整的 Python 代码实现。报告内容遵循学术严谨性，旨在将理论方案转化为具体、可复现的技术实践。

1 NOAA AIS 数据集深度分析

本章节将对作为模型基础的 NOAA AIS 数据集进行深入剖析，涵盖其数据来源、核心字段及其对轨迹建模的意义，并识别其中固有的数据质量挑战。

1.1 数据来源与访问机制

D-STGT 模型将采用公开的大规模 AIS 数据集进行训练与评估，首选数据源为 NOAA 的 Marine Cadastre 项目[2]。该项目是 NOAA、美国海岸警卫队 (USCG) 及美国海洋能源管理局 (BOEM) 的合作成果，旨在提供权威的海洋空间信息[4]。

- **数据来源:** 数据由 USCG 的全国自动识别系统 (Nationwide Automatic Identification System, NAIS) 收集，这是一个由陆基接收器组成的网络，覆盖美国沿海水域[2]。
- **数据格式与频率:** 自 2015 年起，数据以逗号分隔值 (CSV) 格式提供，按天组织成压缩文件。关键在于，这些数据经过了预处理，被统一过滤为一分钟的采样率[2]。
- **访问方式:** 数据可通过多种方式获取。研究者可以通过 AccessAIS 工具进行自定义的地理和时间范围查询[7]，或直接从其数据服务器上通过 HTML 目录批量下载每日数据文件[9]。

一分钟的采样率是该数据集的一个核心特性。对于在开阔水域航行的大型船舶，此频率足以追踪其基本轨迹。然而，在港口、狭窄水道等交通密集、交互复杂的场景中，船舶可能在短短一分钟内完成关键的避碰机动。因此，这一采样率对于捕捉此类细粒度行为而言可能过于稀疏。这直接引出了后续数据预处理流程中一个关键的技术选型：不能简单地依赖线性插值来填补数据点间的空白，而必须采用更高级的状态估计算法（如卡尔曼滤波），以便在比原始数据更高的频率上对船舶的动态状态进行推理，从而为模型提供更平滑、更物理真实的数据基础。

1.2 轨迹建模的核心数据字段

为了构建 D-STGT 模型，需要从 AIS 消息中提取一系列关键的动态和静态信息。根据 NOAA 提供的官方数据字典[10]，以下字段对于轨迹预测任务至关重要。

表 1: NOAA AIS 数据集核心字段释义

字段名称	**描述**	**示例/单位/格式**	**在模型中的作用**
MMSI	海上移动服务识别码	477220100 (9 位数字)	唯一的船舶标识符，用于将数据点分组为轨迹。
BaseDateTime	完整的 UTC 日期和时间	2017-02-01T20:05:07 (ISO 8601)	用于对轨迹点进行时序排序的时间戳。
LAT	纬度	42.35137 (十进制度)	原始 Y 坐标，用于坐标系投影的输入。
LON	经度	-71.04182 (十进制度)	原始 X 坐标，用于坐标系投影的输入。
SOG	对地航速	5.9 (节)	用于推导速度向量大小的输入。
COG	对地航向	47.5 (度)	用于推导速度向量方向的输入。
Heading	真实航向	45.1 (度)	区分船首指向与实际运动方向（COG）。
VesselType	船舶类型代码	70 (整数代码)	关键的静态特征，用于节点嵌入。

字段名称	描述	示例/单位/格式	在模型中的作用
Length	船舶总长度	71.0 (米)	关键的静态特征，用于节点嵌入和碰撞定义。
Width	船舶总宽度	12.0 (米)	关键的静态特征，用于节点嵌入和碰撞定义。

在这些字段中，COG（Course Over Ground，对地航向）和 Heading（船首向）之间的区别值得关注[11]。COG 表示船舶相对于地球的实际运动方向，而 Heading 表示船头所指向的方向。当船舶受到强烈的侧向风或水流影响时，这两个值会有显著差异。对于轨迹预测任务，COG 是推导速度向量的关键，因为它直接反映了船舶的实际位移路径。

此外，一个至关重要的观察是，原始科研方案中定义的状态向量 $s_{t_i} = (x_{t_i}, y_{t_i}, v_{x_i}, v_{y_i})$ 仅包含了动态运动学信息。然而，AIS 数据提供了丰富的静态物理属性，如 `VesselType`、`Length` 和 `Width`[10]。船舶的动力学特性，如惯性、转弯半径和响应时间，与其物理尺寸和类型密切相关。例如，一艘 300 米长的满载油轮与一艘 30 米长的渔船在机动性上有着天壤之别。若模型仅从轨迹数据中隐式学习这些物理约束，不仅数据效率低下，而且泛化能力可能受限。

一个更强大、更符合物理逻辑的方法是将这些静态特征作为模型的直接输入。具体而言，可以将节点特征嵌入函数 f_{embed} 的输入从单纯的动态状态 s_{t_i} 扩展为一个拼接向量，该向量同时包含动态状态 s_{t_i} 和静态特征向量 $s_{\text{static}_i} = (\text{VesselType}, \text{Length}, \text{Width})$。这种设计为模型注入了强烈的归纳偏置，使其能够更好地理解不同船舶的内在动力学差异，从而有望提升预测的准确性、鲁棒性和样本效率。本报告强烈建议对原模型进行此项改进。

1.3 固有的数据质量挑战

原始 AIS 数据并非为科研目的而设计，其质量存在诸多固有挑战，这在原方案中已有提及。这些挑战主要包括：

- **数据缺失与噪声：** 由于 VHF 信号传播受限、信道拥堵或设备故障，AIS 数据流中常出现数据点丢失（信号中断）和异常跳点（位置突变）[6]。
- **信息不一致：** 静态信息（如船型、尺寸）依赖船员手动输入，常有错误或更新不及时的情况[12]。

- **恶意行为:** AIS 系统存在被恶意利用的风险，包括发送虚假位置信息的“AIS 欺骗”(Spoofing)和为隐藏行踪而故意关闭 AIS 发射器的“暗黑航运”(Dark Shipping)[13]。

这些质量问题对模型的训练和评估构成了严重威胁。一个充满噪声和异常的训练集会导致模型学习到错误的关联，而一个未经清洗的测试集则会使评估指标失真。因此，在进行任何特征工程之前，必须建立一个全面、严格的数据预处理和质量保证流程。

2 数据预处理与质量保证流程

本章节将详细阐述一个多阶段的数据预处理流程，旨在将原始、嘈杂的 AIS 数据转化为干净、连续且可信的船舶轨迹，为后续的特征工程奠定坚实基础。

2.1 初始数据加载与轨迹分段

流程的第一步是从 NOAA 提供的每日 CSV 文件中加载数据。考虑到数据量巨大，使用 pandas 库进行高效的 I/O 和数据操作。

```
import pandas as pd
import numpy as np
from pathlib import Path
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

def load_ais_data(data_dir: Path, year: int) -> pd.DataFrame:
    """
    加载指定年份的所有AIS CSV文件，并进行初步清洗。
    """
    csv_files = list(data_dir.glob(f"AIS_{year}_*.csv"))
    if not csv_files:
        logging.warning(f"在目录 {data_dir} 中未找到 {year} 年的AIS数据。")
        return pd.DataFrame()

    # 定义核心列和数据类型以优化内存
    core_columns = {
        'MMSI': 'int32', 'BaseDateTime': 'str', 'LAT': 'float32',
        'LON': 'float32', 'SOG': 'float32', 'COG': 'float32',
        'Heading': 'float32', 'VesselType': 'float32', # 读取为float以便处理NaN
        'Length': 'float32', 'Width': 'float32',
    }
```

```
24
25      df_list = []
26      for file in sorted(csv_files):
27          try:
28              df = pd.read_csv(
29                  file,
30                  usecols=core_columns.keys(),
31                  dtype={k: v for k, v in core_columns.items() if k != '
    BaseDateTime'}
32              )
33              df_list.append(df)
34          except Exception as e:
35              logging.error(f"读取文件 {file} 时出错: {e}")
36              continue
37
38      if not df_list:
39          return pd.DataFrame()
40
41      full_df = pd.concat(df_list, ignore_index=True)
42
43      # --- 初步清洗 ---
44      # 转换时间戳
45      full_df['BaseDateTime'] = pd.to_datetime(full_df['BaseDateTime'],
    errors='coerce')
46
47      # 删除关键信息缺失的行
48      full_df.dropna(subset=['BaseDateTime', 'MMSI', 'LAT', 'LON'], inplace=
    True)
49
50      # 按MMSI和时间排序，这是轨迹处理的基础
51      full_df.sort_values(by=['MMSI', 'BaseDateTime'], inplace=True)
52      full_df.reset_index(drop=True, inplace=True)
53
54      logging.info(f"成功加载并初步处理了 {len(full_df)} 条AIS记录。")
55      return full_df
56
57  # 示例用法
58  # data_directory = Path("./ais_data")
59  # ais_df = load_ais_data(data_directory, 2023)
```

Listing 1: 初始数据加载与初步清洗

此代码段完成了数据加载、基本类型转换和排序，将原始的扁平数据表组织成以 MMSI 标识的、按时间排序的轨迹序列，为后续的单船处理做好了准备。

2.2 基于卡尔曼滤波的先进轨迹平滑

原方案中提出的简单插值方法虽然能填补数据空白，但其在物理上是幼稚的。它无法体现船舶运动的高惯性、慢响应特性，可能产生不符合动力学约束的轨迹。为了解决这一问题，引入卡尔曼滤波器进行状态估计是一种更为先进和物理真实的方法[15]。

卡尔曼滤波器是一个强大的贝叶斯滤波框架，它通过一个预测-更新的循环过程，将系统动力学模型与带噪声的观测数据进行最优融合。

预测 基于上一时刻的状态和物理模型（例如，匀速运动模型），预测当前时刻的状态。

更新 使用当前时刻的实际观测值（带噪声的 AIS 数据）来修正预测值，得到一个更精确的后验估计。

这种方法不仅能有效地平滑噪声，还能在短时间的信号丢失期间提供合理的内插估计，其输出的轨迹在物理上更为连贯和可信。这对于后续模型学习真实的交互模式至关重要。以下是使用 `filterpy` 库[17] 实现的卡尔曼滤波器封装类，用于平滑单条船舶轨迹。

```python
from filterpy.kalman import KalmanFilter
from filterpy.common import Q_discrete_white_noise
from pyproj import Transformer, CRS

def get_local_projection(center_lat: float, center_lon: float) -> Transformer:
    """为给定的中心点创建一个局部切面投影 (Transverse Mercator)。"""
    crs_wgs84 = CRS("EPSG:4326")
    proj_string = f"+proj=tmerc +lat_0={center_lat} +lon_0={center_lon} +k=1 +x_0=0 +y_0=0 +ellps=WGS84 +datum=WGS84 +units=m +no_defs"
    crs_local = CRS.from_proj4(proj_string)
    return Transformer.from_crs(crs_wgs84, crs_local, always_xy=True)

class TrajectorySmoother:
    def __init__(self, dt=60.0, process_noise=1.0, measurement_noise=10.0):
        """初始化卡尔曼滤波器。状态向量 x = [x, y, vx, vy]^T"""
        self.dt = dt
        self.kf = KalmanFilter(dim_x=4, dim_z=2)

        # 状态转移矩阵 F (匀速模型)
        self.kf.F = np.array([[1, 0, dt, 0], [0, 1, 0, dt],
                              [0, 0, 1, 0], [0, 0, 0, 1]])
        # 观测矩阵 H
        self.kf.H = np.array([[1, 0, 0, 0], [0, 1, 0, 0]])
        # 过程噪声协方差 Q
        self.kf.Q = Q_discrete_white_noise(dim=4, dt=dt, var=process_noise)
        # 观测噪声协方差 R
        self.kf.R = np.eye(2) * measurement_noise
```

```
27
28      def smooth_trajectory(self, traj_df: pd.DataFrame) -> pd.DataFrame:
29          """对单条船舶轨迹进行平滑处理。"""
30          if len(traj_df) < 2:
31              return None
32
33          # 1. 坐标转换
34          center_lon, center_lat = traj_df['LON'].mean(), traj_df['LAT'].mean
    ()
35          transformer = get_local_projection(center_lat, center_lon)
36          x, y = transformer.transform(traj_df['LON'].values, traj_df['LAT'].
    values)
37          measurements = np.vstack((x, y)).T
38
39          # 2. 初始化滤波器状态
40          self.kf.x = np.array([measurements[0, 0], measurements[0, 1], 0.,
    0.])
41          if len(measurements) > 1:
42              self.kf.x[2] = (measurements[1, 0] - measurements[0, 0]) / self
    .dt
43              self.kf.x[3] = (measurements[1, 1] - measurements[0, 1]) / self
    .dt
44          self.kf.P = np.eye(4) * 500. # 初始不确定性较大
45
46          # 3. 滤波与平滑 (RTS Smoother)
47          (means, covs) = self.kf.filter(measurements)
48          (smoothed_means, smoothed_covs) = self.kf.rts_smoother(means, covs)
49
50          smoothed_df = traj_df.copy()
51          smoothed_df['x'] = smoothed_means[:, 0]
52          smoothed_df['y'] = smoothed_means[:, 1]
53          smoothed_df['vx'] = smoothed_means[:, 2]
54          smoothed_df['vy'] = smoothed_means[:, 3]
55          return smoothed_df
```

Listing 2: 基于卡尔曼滤波的轨迹平滑

2.3 多层次的异常与离群点检测

一个单一的异常检测方法是脆弱的，因为它可能只对特定类型的异常敏感。为了构建一个鲁棒的质量保证体系，本报告提出一个三层防御策略，结合了规则、数据驱动和模式匹配的方法。

第一层：基于规则的运动学过滤 这是最直接的防御。利用卡尔曼滤波后得到的平滑状态（尤其是速度），可以设定符合物理现实的阈值来剔除不可能的数据点。

第二层：基于自编码器的行为异常检测 有些轨迹在运动学上是可能的，但在行为上是异常的。深度学习自编码器（Autoencoder）是解决此类问题的有力工具[18]。其原理是：在一个大规模的“正常”轨迹片段数据集上训练一个自编码器。当模型遇到一个行为异常的轨迹时，其重构效果会很差，导致较高的“重构误差”。

第三层：基于时间间隔的“暗黑航运”检测 此层专门针对故意关闭 AIS 发射器的行为[13]。这种异常表现为数据流中出现长时间的、不合理的空白。

通过这三层过滤，可以确保输入到 D-STGT 模型的数据不仅平滑、干净，而且代表了真实、善意的航行行为。

表 3: 数据预处理与质量保证流程总结

步骤	阶段	方法	目的/关键参数
1	轨迹分组	Pandas `groupby('MMSI')`	从批量数据中分离出单个船舶的轨迹。
2	坐标转换	pyproj (局部 tmerc 投影)	将经纬度转换为局部米制坐标 (X, Y)。
3	状态估计与平滑	卡尔曼滤波器 (`filterpy`)	平滑噪声，估计状态 (x, y, vx, vy)，填补空白。 参数：过程/测量噪声协方差
4	异常检测 (运动学)	基于规则的阈值法	移除不符合物理规律的状态（如超速）。 参数：最大速度, 最大加速度
5	异常检测 (行为)	自编码器重构误差 (`tensorflow/keras`)	移除具有异常行为模式的整个轨迹。 参数：误差阈值 (如 99 百分位)
6	异常检测 (间隙)	时间差阈值法	标记潜在的“暗黑航运”事件。 参数：最大间隙时长

3 核心特征工程与坐标系变换

本章节将阐述如何将清洗后的轨迹数据转化为 D-STGT 模型所需的精确数学对象。

3.1 转换为局部笛卡尔坐标系

模型中的所有几何计算都必须在笛卡尔坐标系（即米制坐标系）下进行。`pyproj` 库是进行坐标参考系统 (CRS) 转换的标准工具[19]。

考虑到 D-STGT 模型处理的是一个个地理范围相对较小的“场景”，一个更精确、更严谨的方法是为每个场景动态创建一个自定义的局部投影。具体做法是：对于每个待处理的场景，首先计算该场景内所有轨迹点的地理中心（平均经纬度）。然后，使用这个中心点作为原点，创建一个横轴墨卡托投影 (`+proj=tmerc`)[21]。这个投影在

场景的中心区域畸变最小。这种方法确保了模型输入的几何信息具有最高的保真度。‘get_local_projection‘ 函数已在 2.2 节的代码中提供。

3.2 运动学状态向量的推导

模型需要一个四维的状态向量 $s_t = (x_t, y_t, v_x, v_y)$。

1. **单位转换:** 将 SOG 从节 (knots) 转换为米每秒 (m/s)。1 节 ≈ 0.514444 m/s。
2. **向量分解:** 将极坐标形式的速度分解为笛卡尔坐标系下的速度分量 (v_x, v_y)。正确的转换公式为:

$$v_x = \text{SOG}_{\text{m/s}} \times \sin(\text{COG}_{\text{rad}})$$
$$v_y = \text{SOG}_{\text{m/s}} \times \cos(\text{COG}_{\text{rad}})$$

其中，COG_{rad} 是转换为弧度的 COG 值。

最终，所有船舶在所有观测时间步的状态将被组织成一个形状为 $(N, T_{\text{obs}}, 4)$ 的张量。

表 4: 工程化特征规范

特征名称	推导过程	类型/形状	在模型中的作用
`history_states`	平滑投影后的 (x, y, v_x, v_y)	`torch.Tensor(N, T_obs, 4)`	MA-GAT 编码
`future_states`	真实的未来 (x, y, v_x, v_y)	`torch.Tensor(N, T_pred, 4)`	重构损失 L_{reco}
`static_features`	独热编码/归一化特征	`torch.Tensor(N, D_static)`	增强节点嵌入的
`dynamic_adj`	TCPA/DCPA 推导的邻接矩阵	`torch.Tensor(T_obs, N, N)`	定义图结构的
`scene_centroid`	场景平均 (lat, lon)	`tuple(float, float)`	定义局部投影的
`vessel_ids`	场景中的 MMSI 列表	`list[str]`	用于追踪和可视

4 动态海事安全图的工程实现

本章节将实现原方案中的核心创新：一个由海事安全规则驱动的动态交互图。

4.1 高效的 TCPA/DCPA 计算模块

TCPA/DCPA 计算是构建风险图的基础。当船舶数量 N 较大时，$\mathcal{O}(N^2)$ 的 Python 循环会成为性能瓶颈。必须采用 NumPy 的广播（broadcasting）机制进行向量化计算，其速度比纯 Python 循环快几个数量级。

```python
def calculate_pairwise_tcpa_dcpa(states: np.ndarray) -> tuple[np.ndarray, np.ndarray]:
    """
    使用向量化操作高效计算成对的TCPA和DCPA。
    参数: states (np.ndarray): 形状为 (N, 4) 的状态数组 [x, y, vx, vy]。
    返回: tuple[np.ndarray, np.ndarray]: TCPA矩阵和DCPA矩阵, 形状均为 (N, N)。
    """
    # 扩展维度以利用广播: p_i (N,1,2), p_j (1,N,2), etc.
    p_i = states[:, np.newaxis, :2]; v_i = states[:, np.newaxis, 2:]
    p_j = states[np.newaxis, :, :2]; v_j = states[np.newaxis, :, 2:]

    # 计算相对位置和速度向量 (广播后形状为 (N, N, 2))
    p_rel = p_j - p_i
    v_rel = v_j - v_i

    # 计算相对速度的范数平方
    v_rel_norm_sq = np.sum(v_rel**2, axis=2)
    v_rel_norm_sq[v_rel_norm_sq == 0] = 1e-9 # 避免除以零

    # --- 计算 TCPA = -(p_rel · v_rel) / ||v_rel||^2 ---
    p_rel_dot_v_rel = np.sum(p_rel * v_rel, axis=2)
    tcpa = -p_rel_dot_v_rel / v_rel_norm_sq

    # --- 计算 DCPA = ||p_rel x v_rel|| / ||v_rel|| (2D cross product) ---
    p_rel_cross_v_rel = p_rel[..., 0] * v_rel[..., 1] - p_rel[..., 1] * v_rel[..., 0]
    dcpa = np.abs(p_rel_cross_v_rel) / np.sqrt(v_rel_norm_sq)

    # 将对角线(自身与自身)的值设为无穷大
    np.fill_diagonal(tcpa, np.inf)
    np.fill_diagonal(dcpa, np.inf)

    return tcpa, dcpa
```

Listing 3: 向量化的 TCPA/DCPA 计算

4.2 动态图结构的构建

- **边的生成:** 船舶 i, j 间存在边，当且仅当 $0 < \text{TCPA}_{ij} \leq T_{\text{safe}}$ 且 $\text{DCPA}_{ij} \leq D_{\text{safe}}$。
- **权重的计算:** 采用指数衰减函数作为边权重:

$$w_{ij,t} = \exp\left(-\alpha \frac{\text{DCPA}_{ij}}{D_{\text{safe}}} - \beta \frac{\text{TCPA}_{ij}}{T_{\text{safe}}}\right) \tag{1}$$

其中，α, β 是可调超参数。

```python
def build_risk_graph(tcpa_matrix: np.ndarray, dcpa_matrix: np.ndarray,
                     t_safe: float, d_safe: float,
                     alpha: float = 1.0, beta: float = 1.0) -> np.ndarray:
    """
    根据TCPA和DCPA矩阵构建加权邻接矩阵。
    """
    # 1. 创建布尔掩码，确定边的存在性
    adjacency_mask = (tcpa_matrix > 0) & (tcpa_matrix <= t_safe) & (
    dcpa_matrix <= d_safe)

    # 2. 计算归一化 TCPA 和 DCPA
    norm_dcpa = dcpa_matrix / d_safe
    norm_tcpa = tcpa_matrix / t_safe

    # 3. 计算指数衰减权重
    weights = np.exp(-alpha * norm_dcpa - beta * norm_tcpa)

    # 4. 应用掩码，只保留符合条件的边的权重
    adjacency_matrix = np.zeros_like(tcpa_matrix)
    adjacency_matrix[adjacency_mask] = weights[adjacency_mask]

    return adjacency_matrix
```

Listing 4: 构建风险图邻接矩阵

5 场景组装与 PyTorch 数据加载器实现

5.1 轨迹切片与场景聚合

将清洗后的单船轨迹切片并组合成“场景”。一个场景由在同一时空窗口内共同出现的 N 艘船的轨迹片段（T_{obs} 历史 + T_{pred} 未来）组成。

5.2 “即时处理”的 PyTorch 数据集类

为提升实验灵活性，推荐采用“即时处理”（Just-In-Time, JIT）设计模式。大部分计算（如滤波、建图）被放在 PyTorch `Dataset` 类的 `__getitem__` 方法中，并在 `DataLoader` 的多进程 (`num_workers > 0`) 中并行执行。

```python
import torch
from torch.utils.data import Dataset, DataLoader

class AISSceneDataset(Dataset):
```

```
    def __init__(self, scene_definitions: list, raw_traj_data: dict, config: dict):
        self.scene_definitions = scene_definitions
        self.raw_traj_data = raw_traj_data
        self.config = config
        self.smoother = TrajectorySmoother()

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

    def __getitem__(self, idx):
        scene_info = self.scene_definitions[idx]

        # --- 1. 加载、切片并平滑轨迹 ---
        # (此处省略复杂的索引和切片逻辑)
        # 伪代码:
        # for mmsi in scene_info['mmsis']:
        #     traj_slice = get_slice(self.raw_traj_data[mmsi], scene_info['time'])
        #     smoothed_slice = self.smoother.smooth_trajectory(traj_slice)
        #     history_states, future_states, static_features = extract_features(smoothed_slice)
        #     ... append to lists ...

        # 假设已填充好列表并转换为张量
        history_states = torch.randn(len(scene_info['mmsis']), self.config['T_obs'], 4)
        future_states = torch.randn(len(scene_info['mmsis']), self.config['T_pred'], 4)
        static_features = torch.randn(len(scene_info['mmsis']), 10)

        # --- 2. 构建动态邻接矩阵 ---
        adj_matrices = []
        for t in range(self.config['T_obs']):
            current_states = history_states[:, t, :].numpy()
            tcpa, dcpa = calculate_pairwise_tcpa_dcpa(current_states)
            adj_matrix = build_risk_graph(
                tcpa, dcpa, self.config['t_safe'], self.config['d_safe']
            )
            adj_matrices.append(adj_matrix)
        dynamic_adjacency = torch.tensor(np.array(adj_matrices), dtype=torch.float32)

        # --- 3. 组装最终样本 ---
        return {
```

```
            'history_states': history_states, 'future_states':
    future_states,
            'static_features': static_features, 'dynamic_adjacency_matrices
    ': dynamic_adjacency,
            'vessel_ids': scene_info['mmsis']
        }
```

Listing 5: PyTorch Dataset 类框架

6 结论与建议

本报告为 D-STGT 科研方案提供了一套从数据到模型的完整、详细且可执行的技术路线图。

- **数据质量是基础:** 强调了基于卡尔曼滤波的多层次质量保证流程的必要性，以取代简单的插值。
- **特征工程决定模型上限:** 提出了两项关键优化：(1) 将船舶物理属性（类型、长度、宽度）作为静态特征输入模型；(2) 为每个交互场景动态创建局部坐标系，保证几何信息的精确性。
- **实现效率是关键:** 所有计算密集型任务均采用向量化实现。推荐的“即时处理”数据加载器设计，在保证高性能的同时，为科研探索提供了最大的灵活性。

遵循本报告提出的方法论和代码实现，研究者可以构建一个高质量的数据基础，从而更有效地验证和发挥 D-STGT 模型的潜力。