NumVerify — 数值验证框架

1. 项目简介 & 用途说明

NumVerify（数值验证框架） 是一个面向移动设备行为建模的实验性框架，旨在基于真实用户数据进行数值验证与场景模拟。其主要目标是验证各类预测模型和 pipeline 的可行性及效果。框架通过读取本地 `./UserModel/data/` 目录下由 dubai 数据库提供的用户设备使用数据（包括电池使用记录、前台应用使用记录等）进行建模和测试。**NumVerify** 框架提供了完整的数据处理、特征构造、聚类建模、耗电预测及误差评估流程，模拟设备使用场景（如电池耗电预测、应用使用预测、散热预测等），从而验证模型算法的有效性并评估整个数据处理与预测流程是否合理可靠。

2. 项目结构

本项目包含两个主要子模块目录：`NumValKit/` 和 `VerifyScenarios/`。

2.1 顶层结构

```
NumVerify
├── NumValKit           # 通用数值建模函数模块库
└── VerifyScenarios     # 可运行的场景脚本与测试逻辑
```

2.2 NumValKit 模块

NumValKit 是核心库，**封装了一系列基础类、加载器、预测模型与工具方法**，结构如下：

```
NumValKit/
├── __init__.py                     # 包初始化文件，标识 NumValKit 包，可用于设置包级别常量或引入子模块
└── numvalkit/
    ├── core                                # Core 模块：提供模块的抽象基类
    │   ├── base_generator.py               # 定义 Generator 抽象基类（行为向量生成器、热会话生成器等继承）
    │   ├── base_loader.py                  # 定义 Loader 抽象基类（各类数据加载器如电池、前台、产品等继承）
    │   ├── base_predictor.py               # 定义 Predictor 抽象基类（各模型预测器统一接口）
    │   └── __init__.py                     # Core 模块初始化文件
    ├── data_loader/                        # 各类数据加载与预处理模块
    │   ├── battery_loader.py               # 负责读取并预处理电池使用日志数据（如放电、充电时间戳）
    │   ├── behave_vector_generator.py      # 从前台应用行为与电池数据中生成行为向量序列
    │   ├── foreground_loader.py            # 读取并预处理前台应用使用记录（如使用时长、应用类别）
    │   ├── __init__.py                     # data_loader 模块初始化文件
    │   ├── product_loader.py               # 读取设备/产品型号信息、手机容量、型号映射等辅助数据
```

```
    │   └── thermal_session_generator.py            # 基于热测会话日志构造热场景序列（用于热预测模块）
    ├── predictors/                              # 各类预测模型实现
    │   ├── battery_ae_predictor.py                  # 【实验性】基于自编码器（AE）实现的电池耗电预测模型
    │   ├── battery_aosp_predictor.py                # 【Baseline】基于 Android 原生算法（AOSP）做电池续航预测实现
    │   ├── battery_dt_predictor.py                  # 【实验性】基于决策树（DT）的电池预测模型
    │   ├── battery_kmeans_predictor_plus_huber.py  # 【实验性】K-Means + Huber 回归超参数版
    │   ├── battery_kmeans_predictor_plus.py         # 【SOTA】K-Means + 特征工程
    │   ├── battery_kmeans_predictor.py              # 【Baseline】基础 K-Means 聚类电池耗电预测模型
    │   ├── battery_statistical_predictor.py         # 【Baseline】基于统计平均模型的电池耗电预测器（baseline）
    │   ├── behave_ggnn_predictor.py                 # 基于行为图（GGNN）的行为模式预测模型
    │   ├── behave_kmeans_predictor.py               # 基于行为向量聚类（K-Means）的行为预测模型
    │   ├── behave_topk_predictor.py                 # 基于 Top-K 相似历史日的行为匹配预测器
    │   ├── __init__.py                              # predictors 模块初始化文件
    │   ├── thermal_lgbm_predictor.py                # 基于 LightGBM 的热预测模型（热场景下的功耗/温度预测）
    │   └── thermal_rf_predictor.py                  # 基于随机森林（RF）的热预测模型
    └── utils/                                   # 工具函数模块
        ├── app_type.csv                             # 应用类别映射表（应用名→类别）
        ├── __init__.py                              # utils 模块初始化文件
        └── utils.py                                 # 包含数据处理、可视化、评价指标等通用函数
```

2.3 VerifyScenarios 模块

该目录下为**用于运行的用户行为预测与评估任务的具体实现场景逻辑脚本**：

```
VerifyScenarios/
├── config.py                   # 用户列表与缓存策略配置
├── discharge_prediction.py     # 用户 APP 耗电预测评估脚本（原始版本）
├── discharge_prediction_.py    # 混合时间片耗电评估脚本（推荐使用）
└── utils.py                    # 评估辅助函数（局部调用）
```

3. 核心模块说明

项目代码主要由以下几个 Python 模块组成，每个模块承担不同功能。

3.1 行为向量生成模块

`behave_vector_generator.py`：行为向量生成模块。负责读取指定用户的前台应用使用日志和电池使用日志，生成**行为向量序列**。该模块将时间线按固定粒度切分（例如每 60 分钟一个片段），在每个时间片段内统计用户使用各类别应用的时长，并结合电池数据计算该时间片段的耗电量（discharge）及是否在充电状态的标记。结果以字典形式返回，键为时间片段起始时间，值为包含该片段**行为向量**（应用类别使用时长分布）、**耗电量**以及**充电标志**的三元组。若开启缓存功能，生成的行为向量序列会缓存在用户数据文件夹中，以便下次直接读取。

3.2 K-Means 聚类电池预测模块

`battery_kmeans_predictor.py / battery_kmeans_predictor_.py`：K-Means 聚类电池预测模块，前者为备份版本，后者为采用效果最好的**混合时间片训练**版本。实现了 **BatteryKmeansPredictor** 类，用于基于用户行为模式进行电池耗电的聚类建模与预测。其核心思想是将一天划分为多个时段（如凌晨 0-9 点、白天 9-15 点、傍晚 15-24 点），对每个时段的用户行为向量进行 K-Means聚类，将这些"类似日间使用模式"合为类。模型在训练阶段会根据聚类结果计算历史平均耗电曲线（排除充电影响）用于指导预测。通过结合最新一次观察到的耗电速率与历史平均模式（由超参数 α 和 β 加权）进行迭代估算，该预测器能够推断给定起始时间和未来一段时间内的电量消耗。本模块还提供方法来预测电池可持续时长或特定时长内的耗电量（分别通过 `predict_battery_life` 和 `predict_discharge` 实现）。BatteryKmeansPredictor 在 NumVerify 中用于模拟通过匹配当前用户行为与历史模式来预测耗电的方案。

3.3 耗电预测评估脚本

`discharge_prediction.py`：耗电预测评估脚本。该模块是一个独立可运行的脚本，用于评估各种电池预测方法在用户数据上的表现。脚本会调用前述的 BehaveVectorGenerator 生成用户的行为向量序列，并利用 BatteryKmeansPredictor 等模型对耗电情况进行预测，比较预测值与真实值的偏差。它针对预设的一系列场景（例如不同的起始时间段和不同长度的预测区间）计算预测误差，并输出评估指标和可视化结果。评估过程中产生的误差数据、统计指标以及图表会输出至项目根目录下的 `/results/` 目录。

3.4 宏配置模块

`config.py`：配置模块。包含项目运行所需的一些全局配置和参数，例如数据目录路径、需要处理的用户列表、缓存刷新选项等。默认情况下，`DATA_DIR` 指向项目中的 `UserModel/data` 路径。`user_list` 定义了参与分析的用户 ID 集合（例如默认包含 user1 至 user1540 等多个用户，并可排除部分数据异常用户），`regenerate_cached_data` 则控制是否强制重算缓存的数据。通过修改 config.py，用户可以调整评估使用的数据集范围和参数设置。

3.5 工具函数模块

`utils.py`：工具函数模块。提供了一些辅助功能函数用于数据分析与结果可视化。例如，`weighted_average_2d()` 用于根据样本权重计算二维误差矩阵的加权平均值；`draw_error_matrix_figure()` 可绘制每个用户在不同起始时间和预测区间下的误差矩阵热力图；`draw_error_abs_cdf()` 则用于绘制误差绝对值的累积分布函数（CDF）曲线图，以评估预测误差分布情况。该模块还包含如 `get_productname_from_user()` 之类的工具函数，用于根据 user ID 获取设备型号等信息（需要数据目录下提供 `phone_type.csv` 映射文件）。这些工具函数在耗电预测评估脚本和其他模拟场景脚本中被调用，用于计算评估指标或生成分析图表。

4. 核心类与方法说明

下面重点介绍项目中的两个核心类及其主要方法：`BehaveVectorGenerator` 和 `BatteryKmeansPredictor`。

4.1 BehaveVectorGenerator 类

BehaveVectorGenerator 负责从原始的前台应用使用日志与电池使用日志中，生成标准化的**行为向量序列（behavior vector sequence）**。生成过程包含数据对齐、时间切片、行为统计与缓存机制等步骤。

```
BehaveVectorGenerator.generate(user_id, vector_granularity, use_appname=False,
reflush=False)
```

其核心流程可分为以下几个部分：

4.1.1 数据加载

- 内部调用 `BatteryLoader` 与 `ForegroundLoader` 分别读取电池事件与前台应用使用日志。
- 输入参数 `user_id` 会自动映射到对应的数据文件夹。

4.1.2 时间切片

- 根据 `vector_granularity`（如 30min / 60min）将整体时间轴均匀切分。
- 为每个时间片建立统计容器，用于累计行为与耗电信息。

4.1.3 统计行为向量

- 对每条前台 App 记录，计算其与当前时间片的时间重叠，并将重叠时长累计到对应类别或应用名。
- 输出的行为向量是一个 `{category → seconds}` 的字典，可根据 `use_appname` 控制维度是应用类别还是应用名。

4.1.4 统计放电与充电状态

- 遍历电池事件，按时间片重叠比例累加放电量，并记录该片段内是否发生充电。
- 每个时间片最终形成：`(behavior_vector, discharge, is_charging)` 的三元组

4.1.5 缓存机制

- 每次生成的行为向量序列会缓存到用户目录下：`vector_{granularity}_{use_appname}`
- 若缓存存在且未设置 `reflush=True`，则优先读取缓存，可显著加快后续运行速度。

4.1.6 调用示例

```
gen = BehaveVectorGenerator("./data")
seq = gen.generate("user123", vector_granularity=30, use_appname=False)
```

输出为：

```
{
    Timestamp(...): (behavior_vector, discharge, is_charging),
```

```
    ...
}
```

4.2 BatteryKmeansPredictorPlus 类

BatteryKmeansPredictorPlus 是在原有 **BatteryKmeansPredictor** 基础上扩展的版本，用于基于用户行为模式进行电池耗电预测。它在整体流程上沿用"按时间段聚合行为 → K-Means 聚类 → 统计历史耗电曲线 → 用 α/β 平滑进行迭代预测"的框架，但新增了若干可配置的特征工程模块，以提升聚类质量和时间建模能力。

4.2.1 初始化与可选特征开关

构造函数形如：

```
BatteryKmeansPredictorPlus(
    n_clusters,
    alpha,
    beta,
    standardize=False,
    use_pca=False,
    pca_var_ratio=0.95,
    weekend_tag=False,
    time_triangle=True,
    time_tri_anchors=(0, 4, 8, 12, 16, 20),
    time_tri_width=6.0,
)
```

- `n_clusters`：KMeans 聚类簇数上限（实际会取 `min(n_clusters, 样本数)`）
- `alpha, beta`：控制预测阶段"最近一次耗电观测值"和"历史平均耗电模式"的加权比例（线性组合）
- `standardize`：若为 True，则对行为特征做**行归一化（转为占比）+ StandardScaler 标准化**
- `use_pca` / `pca_var_ratio`：控制是否对特征做 PCA 降维，以及保留的累计方差比例（如 0.95）
- `weekend_tag`：若为 True，则为每个样本增加一个工作日/周末二值特征
- `time_triangle` / `time_tri_anchors` / `time_tri_width`：控制是否使用**时间三角核特征**，以及三角基函数的锚点和宽度，用于在 24 小时周期上构造平滑的时间编码

同时，类内部会为不同时间窗口（0–9、9–15、15–24）分别保存对应的 StandardScaler 与 PCA 变换器，确保每个窗口使用各自的特征空间。

4.2.2 特征构建：_build_features

`_build_features(df, window_id)` 是该类中最核心的特征工程函数，它接收一个以日期为索引、列为应用类别使用秒数的 DataFrame，输出可供 KMeans 使用的二维特征矩阵。其处理流程为：

1. **工作日/周末标签（可选）** 当 `weekend_tag=True` 时，会根据日期索引，添加一列 `__is_weekend__`（工作日为 0，周末为 1），使 KMeans 能区分工作日与周末的行为模式。

2. **时间三角特征（可选）** 当 `time_triangle=True` 时，函数会根据当前时间窗口的中心时刻（默认：0–9 对应 4.5h，9–15 对应 12h，15–24 对应 19.5h），对给定的一组锚点时间（如 0、4、8、12、16、20 小时）计算**三角核响应值**，并添加若干列形如 `__ttri_0__`、`__ttri_4__` 等的时间特征。

- 每个锚点特征代表在 24h 周期上的“接近程度”，在锚点附近值为 1，随着时间距离增大线性衰减，超过给定宽度后变为 0。
- 在同一时间窗口内的所有日期样本共享相同的三角特征值，相当于给窗口贴上一个平滑的时间编码。

3. **行归一化 + 标准化（可选）** 当 `standardize=True` 时：
   - 对每一行先做 L1 归一化，将“每类使用秒数”转为“占比向量”，消除一天内总使用时长差异的影响；
   - 然后通过 `StandardScaler` 对所有特征做零均值、单位方差标准化，并将该 scaler 缓存到 `_scalers[window_id]` 中，以便将来复用。
4. **PCA 降维（可选）** 当 `use_pca=True` 时：
   - 采用 `PCA(n_components=pca_var_ratio, svd_solver="full")`，自动选择主成分数，使累计方差达到设定比例（例如 95%）；
   - 将降维后的矩阵作为 KMeans 的输入特征，并缓存对应的 PCA 模型到 `_pcas[window_id]`。

若未启用某些模块，则保留原始特征或跳过相应步骤，从而兼容基础版的行为。

4.2.3 模型训练流程：fit

`fit(user_vector_sequence, vector_granularity)` 的整体逻辑与基础版 KMeans 预测器保持一致，但在构建行为特征时使用了上述增强版特征工程。

1. **按时间段聚合行为** 根据 `user_vector_sequence` 中每个时间片的时间信息，将一天划分为三个窗口：
   - 0: `[00:00, 09:00)`
   - 1: `[09:00, 15:00)`
   - 2: `[15:00, 24:00)`

   对每个窗口分别按“日期”聚合行为向量，得到三个 `daily_vectors` 字典，每个字典存放某日期该时间段内的合并行为特征。
2. **构建日级行为 DataFrame** 将每个窗口的 `daily_vectors` 转为 DataFrame：行索引为 date，列为应用类别，自动对缺失类别补零，并按日期排序，得到三个 `daily_df`。
3. **按窗口进行特征工程 + KMeans 聚类** 对每个窗口的 DataFrame：
   - 先 drop 掉 `"off"` 列（若存在），再 fillna(0)；
   - 调用 `_build_features(df_filtered, window_id)` 生成特征矩阵 `X_for_kmeans`；
   - 使用 KMeans （`n_init="auto"`）在这个特征空间中聚类，每个日期被分配一个 `cluster` 标签；
   - 将聚类结果回写到 DataFrame 中，形成带 `cluster` 列的 `behavior_df`（存入 `behavior_list`）。
4. **基于聚类标签统计耗电曲线** 与基础版类似，对每个日期 `day`，遍历三个窗口对应的 `behavior_df` 列表：
   - 取该日期在当前窗口的簇 ID（若当日缺失，则默认簇ID为 0）；
   - 找到属于该簇的所有历史日期，排除当前日期，作为“同类行为日”集合；

- 在 `discharge_df` 中筛选出这些日期对应的数据，计算在**不充电时间片**条件下，每个 time slot 的平均放电量（`groupby("time")["discharge"].mean()`），并加入 `avg_discharge_by_time_list`。

同时，还会基于所有日期（排除当前日期）计算一个**全局平均耗电曲线** `alldays_avg_discharge_by_time`，作为回退使用。最终，对每个 `day`，将 `(avg_discharge_by_time_list, alldays_avg_discharge_by_time)` 存入 `average_per_granularity_discharge_dict[day]` 中。

4.2.4 预测接口与 α/β 平滑

预测接口与基础版保持一致，核心由内部的 `predict()` 完成，外部通过：

```
predict_battery_life(start_time, last_hist_discharge, capacity, ratio)
predict_discharge(start_time, last_hist_discharge, target_remaining_mins)
```

进行调用。

- 根据 `start_time` 所在的时间段选择对应的平均耗电曲线：
  - 0–9 点：使用该日的全局平均曲线；
  - 9–15 点：使用该日 `avg_discharge_by_time_list[0]` 对应的曲线；
  - 15–24 点：使用该日 `avg_discharge_by_time_list[1]` 对应的曲线； 若当前时刻在该曲线中缺失，则回退到 `alldays_avg_discharge_by_time`，再不行则回退到一个“熄屏基线”的固定耗电值。
- 在每个时间片上，迭代更新预测值：

  ```
  predicted = alpha * last_hist_discharge + beta * hist_avg
  ```

  其中 `last_hist_discharge` 为上一时间片的预测值（第一步则来自真实历史），`hist_avg` 为当前 time slot 的历史平均耗电。
- 当 `opcode` 为 1 时，表示预测“还能用多久”：根据剩余容量和每片耗电量，累积得到电池寿命（分钟）；当 `opcode` 为 2 时，表示预测“未来若干分钟的耗电量”，则按时间片累加预测放电量。

5. 耗电预测的评估流程

本项目通过 `discharge_prediction_.py` 脚本，对不同预测方法在真实用户数据上的耗电预测性能进行系统评估。整个流程从行为向量生成、样本构建，到多方法预测与误差统计，形成一条完整的 evaluation pipeline，默认评估结果会保存在项目根目录的 `./results/` 目录下。

5.1 评估场景设计

脚本预设了一组统一的评估场景，用于覆盖一天中不同时间段、不同预测窗口长度：

- **起始时间（start_time）**：
  `start_time_list = range(0, 24)`，即从每天 0 点到 23 点，每个整点均作为一个潜在的预测起点。
- **预测时长（interval）**：
  `interval_list = [12, 9, 6, 3, 1]`，分别表示预测未来 12/9/6/3/1 小时的总放电量。
- **时间粒度自适应（interval_granularity_map）**：
  不同预测时长使用不同的时间片粒度，以平衡时间分辨率与样本数量：
  - 12 / 9 / 6 小时 → 使用 **90 分钟粒度** 的行为向量；
  - 3 / 1 小时 → 使用 **60 分钟粒度** 的行为向量。

```
interval_granularity_map = {
    12: 90,
    9: 90,
    6: 90,
    3: 60,
    1: 60,
}
```

- **历史窗口长度（seq_len）**： `seq_len = 5`，即对每个预测起点，必须能够向前找到连续 5 个时间片（5 × granularity）的历史行为向量，作为模型的输入上下文。
- **样本约束条件**： 对于任意一条评估样本，必须同时满足：
  - 预测起点之前存在 **完整的 5 个历史时间片**（无缺失向量）；
  - 在整个预测时间窗口内 **没有发生充电事件**（`is_charging == False`），以避免充电行为干扰纯放电过程的建模。

只有满足以上条件的 `(user, day, start_time, interval)` 组合才会被纳入评估样本集。

5.2 行为向量与数据集构建

5.2.1 多粒度行为向量生成

对每个用户 `user`，评估函数 `predictUserDischarge(user, method_name)` 首先通过 `BehaveVectorGenerator` 为所有需要的时间粒度一次性生成行为向量序列：

```
behaveVectorGenerator = BehaveVectorGenerator(DATA_DIR)
needed_granularities = sorted(set(interval_granularity_map[i] for i in
interval_list))

user_vector_seq_by_g = {}
for g in needed_granularities:
    user_vector_seq_by_g[g] = behaveVectorGenerator.generate(
        user, g, use_appname=False, reflush=regenerate_cached_data
    )
```

- `generate()` 返回一个以时间戳为键、以 `(behavior_vector, discharge, is_charging)` 为值的时间序列字典。
- 若 `regenerate_cached_data=False`，优先从用户目录下缓存文件中读取，以减少重复计算。

5.2.2 DatasetGenerator 构建样本集

`DatasetGenerator.generate_dataset()` 负责从行为向量序列中构建评估样本集。其输出结构为：

```
val_dict: Dict[(start_time, interval), List[(day, vector_list, discharge)]]
```

其中：

- `start_time`：预测起点小时（0–23）；
- `interval`：预测时长（1/3/6/9/12 小时）；
- `day`：样本对应的日期；
- `vector_list`：长度为 `seq_len` 的历史行为向量序列（按时间排序）；
- `discharge`：从 `start_time` 起，在 `interval` 小时窗口内实际累计的总放电量。

在构建过程中：

1. 对每个 `(start_time, interval)` 和每个 `day`：
    - 从 `start_time` 向前，以当前粒度 `vector_granularity` 回溯 `seq_len` 个时间片，若任一时间片缺失，则该样本丢弃。
    - 从 `start_time` 起，向前累计 `interval * 60` 分钟，每 `vector_granularity` 为一个步长：
        - 若有时间片缺失，或该时间片标记为充电（`is_charging=True`），则丢弃该样本；
        - 否则累加对应的 `discharge`，作为该样本的真实总放电量。
2. 将所有满足约束的样本追加到 `val_dict[(start_time, interval)]` 中。
3. 为提升后续调用效率，Dataset 会以粒度和 interval 组合为 key 缓存在用户目录下，例如：

```
{user}/dataset_{vector_granularity}_{seq_len}_{interval_tag}
# interval_tag 形如 "12_9_6_3_1h" 或单 interval "12h"
```

同一 `user + 粒度 + seq_len + interval组合` 的数据集在后续运行中可直接加载，无需重新扫描时间序列。

5.3 预测方法与模型拟合

脚本通过一个生成对象函数 `make_predictor(method_name, user)` 来构造不同的预测器实例，支持的 `method_name` 包括：

- `kmeans`：基础版 `BatteryKmeansPredictor`
- `kmeans_plus`：增强特征版 `BatteryKmeansPredictorPlus`
- `kmeans_plus_huber`：引入 Huber 损失鲁棒回归的变体

- `baseline`：统计基线预测器 `BatteryStatisticalPredictor`
- `aosp`：Android 原生策略仿真 `BatteryAospPredictor`
- `ae`：基于自动编码器表征的 `BatteryAEPredictor`
- `dt`：基于决策树/时间序列模型的 `BatteryDTPredictor`

在同一用户内部，不同 **时间粒度** 下的数据会分别拟合各自的一套模型，以便后续在不同预测窗口中复用：

```
fitted_by_g = {}
for g in needed_granularities:
    predictor = make_predictor(method_name, user=user)

    if method_name in ["kmeans", "kmeans_plus", "baseline", "ae", "dt",
"kmeans_plus_huber"]:
        uvs = user_vector_seq_by_g[g]
        predictor, err = safe_fit(predictor, uvs, g)
    elif method_name == "aosp":
        predictor, err = safe_fit(predictor, f"
{DATA_DIR}/{user}/battery_data.csv", g, battery_capacity)
```

- `safe_fit()` 用于捕获拟合异常，并将异常用户加入 `detacted_abnormal_user_in_runtime` 集合，避免单个用户数据问题影响整体评估。
- 每个粒度 `g` 只在首次使用时调用一次 `fit()`，结果缓存在 `fitted_by_g[g]` 中；后续同粒度的不同 interval 直接共享该模型。

5.4 误差计算与评价指标

评估主循环对每个 `(start_time, interval)` 组合进行预测与误差统计：

1. **按 interval 选择粒度与模型**
    - 对给定 `interval`，通过 `interval_granularity_map[interval]` 取出对应的 `granularity`；
    - 从 `fitted_by_g` 中取出该粒度下的 `batteryPredictor`；
    - 若模型缺失或该粒度下无法构建数据集（例如行为向量为空），则将对应单元的误差记为 `NaN`，权重记为 0。
2. **样本级预测与误差**

    对 `val_dict[(start_time, interval)]` 中的每个样本 `(day, vector_list, discharge)`：
    - `start_predict_time`：由 `day` 与 `start_time` 组合得到预测起始时间；
    - `last_hist_discharge`：取历史窗口里最后一个时间片的真实放电量 `vector_list[-1][1]`，作为预测起点的最近观测耗电基准；
    - 若历史最后一个时间片处于充电状态（`vector_list[-1][2] == True`），则为稳妥起见再次跳过该样本；
    - 调用预测器：

```
predicted_discharge = batteryPredictor.predict_discharge(
    start_predict_time, last_hist_discharge, interval * 60
)
```

若返回 `-1` 表示模型拒绝该样本（如内部判断异常），同样跳过。

- 将预测值与真实值累积，同时记录该样本的 **归一化绝对误差**：

```
error_abs_list[idx].append(
    (predicted_discharge - real_discharge) / battery_capacity
)
```

3. **起点级别相对误差矩阵**

对每个 `(interval_idx, start_time)`：

- 若 `valid_count > 0` 且 `real_sum > 0`，则计算该起点下的**相对误差**：

```
error_mat[idx][start_time] = (predicted_sum - real_sum) / real_sum
weight_mat[idx][start_time] = valid_count
```

- 否则记为 `NaN`，权重为 0。

从而对每个用户得到一个形如 `(len(interval_list), 24)` 的误差矩阵 `error_mat` 与权重矩阵 `weight_mat`，行对应预测时长、列对应起始小时。

4. **用户内平均误差（跨起点加权）**

调用工具函数 `weighted_average_2d` 沿起点维度（axis=1）进行加权平均，得到每个 interval 在该用户上的平均误差：

```
average_err, weights = weighted_average_2d(np.abs(error_mat), weight_mat,
axis=1)
```

- `average_err[i]`：该用户在第 `i` 个预测时长上的加权平均绝对相对误差；
- `weights[i]`：参与该时长统计的有效样本总数量。

函数最终返回：

```
user, error_mat, average_err, weights, error_abs_list
```

5.5 跨用户汇总与结果输出

在最外层，脚本对配置的 `eval_methods` 中的每个方法逐一评估，例如：

```
eval_methods = ["kmeans_plus"]
```

通过 `ProcessPoolExecutor(max_workers)` 对 `user_list` 中的所有用户并行执行 `predictUserDischarge()`，收集得到：

- `user_error_mat[user]`：每个用户的误差矩阵；
- `user_average_err[user]`：每个用户在各个预测时长上的平均误差；
- `user_weights[user]`：对应的样本数权重；
- `user_error_abs_list[user]`：样本级绝对误差（按 interval 组织的列表）。

随后，脚本完成以下输出（以 `method` 表示当前评估方法名）：

1. **用户级平均误差 CSV**

   ```
   df_user_average_err = pd.DataFrame(user_average_err)
   df_user_average_err.to_csv(f"{result_dir + method}.csv", index=False)
   ```

   - 行：不同预测时长（顺序与 `interval_list` 一致）；
   - 列：不同用户 ID；
   - 值：该用户在对应预测时长上的平均绝对相对误差。

2. **总体加权平均误差（跨用户）**

   将所有用户的 `average_err` 与 `weights` 堆叠为二维数组，调用 `weighted_average_2d` 沿用户维度（axis=0）进行加权平均，得到每个 interval 的全局平均误差：

   ```
   avg_array, weights = weighted_average_2d(stacked_error, stacked_weights,
   axis=0)
   ```

   结果写入文本文件：

   ```
   ./results/{method}.txt

   interval 12h's average error is X1, valid count is N1
   interval 9h's  average error is X2, valid count is N2
   ...
   ```

3. **MAE / battery_capacity 指标**

   基于 `error_abs_list`（即样本级 `(pred - real) / battery_capacity`），脚本进一步计算：

   - **每用户整体 MAE/capacity**（所有预测区间与起点合并后的平均绝对误差）：

```
per_user_overall_mae_cap[user] = mean(abs(errs_user_all_intervals))
```

输出至：

```
./results/{method}_mae_cap_per_user.csv
```

- **全局整体 MAE/capacity**：将所有用户、所有 interval 的样本误差合并后取平均，得到单一标量 `overall_mae_cap`。
- **按 interval 的 MAE/capacity**：对每个预测时长，单独收集所有用户、所有起点的样本误差并求平均，得到列表 `interval_mae_cap_sample_weighted`。

以上指标写入：

```
./results/{method}_mae_cap.txt
```

内容形如：

```
overall MAE/capacity: 0.12
interval 12h MAE/capacity: 0.15
interval 9h  MAE/capacity: 0.14
...
```

4. **可选图表**

如需可视化分析，可以启用下面两类图表生成函数（已在脚本末尾给出调用样例并默认注释）：

- `draw_error_abs_cdf(user_error_abs_list, interval_list, method, DATA_DIR)`：绘制不同 interval 下误差绝对值的 CDF 曲线；
- `draw_error_matrix_figure(user_error_mat, method)`：绘制每个用户在不同起点/interval 下的误差矩阵热力图。