ECE5831 Final Project

Version

Table of Contents

Contents:

Project Documentation

Preprocessing

class Preprocessing. Preprocessing (*districts : Dict | None = None*)

Bases: `object`

assign_district (*row : Series*) → str

Assigns a district to a given point.

Parameters: - row (pd.Series): A row containing ‘Long’ and ‘Lat’ for the point.

Returns: - str: The name of the closest district containing the point or “no district” if none found.

assign_district_vectorized (*taxi_df : DataFrame*) → DataFrame

Assigns district names to taxi data based on their start coordinates using vectorized operations.

This method iterates over the districts dataframe and assigns the corresponding district name to each taxi trip based on the start longitude and latitude. If the districts data is not loaded, it assigns “no district” to all taxi trips.

Parameters:

taxi_df pd.DataFrame

DataFrame containing taxi trip data with columns ‘START_LONG’, ‘START_LAT’, and ‘DISTRICT_NAME’.

Returns:

pd.DataFrame

Updated DataFrame with the ‘DISTRICT_NAME’ column assigned based on the start coordinates.

Raises:

None

Notes:

- The method assumes that the ‘districts_df’ attribute is a DataFrame with columns ‘left_long’,

‘right_long’, ‘lower_lat’, ‘upper_lat’, and ‘DISTRICT_NAME’. - The ‘DISTRICT_NAME’ column in the taxi_df should initially be set to “no district” for proper assignment. - Logging is used to provide information and debugging details about the assignment process.

assign_districts (*taxi_df : DataFrame* , *method : str = 'vectorized'* , *sample_size : int = 1000* , *use_sample : bool = False*) → DataFrame

Assign district names to taxi data points using the specified method.

Parameters:

taxi_df pd.DataFrame

DataFrame containing taxi trajectory data.

method str, optional

Method to use (‘vectorized’ or ‘row-wise’). Defaults to ‘vectorized’.

sample_size int, optional

Number of samples to process for testing. Defaults to 1000.

use_sample bool, optional

Whether to process a sample or the entire dataset. Defaults to False.

Returns:

pd.DataFrame

Taxi DataFrame with assigned district names.

calculate_avg_speed (*df : DataFrame* , *distance_column : str = 'TRIP_DISTANCE'* , *travel_time_column : str = 'TRAVEL_TIME'* , *speed_column : str = 'AVG_SPEED'* , *unit : str = 'km/h'*) → DataFrame

Calculate the average speed for each trip based on distance and travel time.

Parameters:

df pd.DataFrame

The input DataFrame.

distance_column str, optional

The name of the column that contains trip distance (default is “TRIP_DISTANCE”).

travel_time_column str, optional

The name of the column that contains travel time data (default is “TRAVEL_TIME”).

speed_column str, optional

The name of the column to store the calculated average speed (default is “AVG_SPEED”).

unit str, optional

Unit for average speed calculation (“km/h” or “miles/h”). Defaults to “km/h”.

Returns:

pd.DataFrame

DataFrame with a new column containing the average speed of each trip.

Raises:

ValueError

If the specified distance or travel time columns do not exist or contain invalid data.

calculate_end_time (*df : DataFrame* , *start_time_column : str = 'TIMESTAMP'* , *travel_time_column : str = 'TRAVEL_TIME'*) → DataFrame

Calculate end time by adding travel time to the start time.

Parameters:

df pd.DataFrame

The input DataFrame.

start_time_column str, optional

The name of the column that contains start time data (default is “TIMESTAMP”).

travel_time_column str, optional

The name of the column that contains travel time data (default is “TRAVEL_TIME”).

Returns:

pd.DataFrame
The DataFrame with a new column ‘END_TIME’.

Raises:

ValueError
If the specified start time or travel time columns do not exist or contain invalid data.

calculate_travel_time (*df : DataFrame* , *polyline_column : str = 'POLYLINE'*) → DataFrame
Calculate the travel time based on the polyline data where each point represents 15 seconds.

Parameters:

df pd.DataFrame
The input DataFrame with a ‘POLYLINE’ column.

polyline_column str, optional
The name of the column containing the polyline data.

Returns:

pd.DataFrame
DataFrame with an added ‘TRAVEL_TIME’ column containing the travel time in seconds.

calculate_trip_distance (*df : DataFrame* , *polyline_column : str = 'POLYLINE'* , *distance_column : str = 'TRIP_DISTANCE'* , *unit : str = 'km'*) → DataFrame
Calculate the total distance of each trip based on the polyline data.

Parameters:

df pd.DataFrame
The input DataFrame with a 'POLYLINE' column.

polyline_column str, optional
The name of the column containing the polyline data (default is "POLYLINE").

distance_column str, optional
The name of the column to store the calculated trip distance (default is "TRIP_DISTANCE").

unit str, optional
Unit for distance calculation ("km" or "miles"). Defaults to "km".

Returns:

pd.DataFrame
DataFrame with a new column containing the total distance of each trip.

Raises:

ValueError
If the specified polyline column does not exist or contain invalid data.

convert_coordinates (*string*)
Loads list of coordinates from given string and swap out longitudes & latitudes. We do the swapping because the standard is to have latitude values first, but the original datasets provided in the competition have it backwards.

convert_timestamp (*df : DataFrame* , *timestamp_column : str = 'TIMESTAMP'*) → DataFrame
Converts a UNIX timestamp into a windows timestamp in the year-month-day hour minute-second format

Parameters:

df pd.DataFrame
The input DataFrame.

data_column str, optional

The name of the column that indicates the timestamp data (default is “TIMESTAMP”).

Returns:

pd.DataFrame

The DataFrame with the converted timestamp in a new column named ‘CONVERTED_TIMESTAMP’.

Raises:

ValueError

If the specified timestamp column does not exist in the DataFrame.

drop_columns (*df : DataFrame* , *columns_to_drop : list | None = None*) → DataFrame

Drops the specified columns from the DataFrame.

Parameters:

df pd.DataFrame

The input DataFrame from which columns will be dropped.

columns_to_drop list, optional

A list of column names to drop. If not provided, defaults to [“ORIGIN_CALL”, “ORIGIN_STAND”].

Returns:

pd.DataFrame

A new DataFrame with the specified columns removed.

Raises:

ValueError

If the input DataFrame is empty.

drop_nan (*df : DataFrame*) → DataFrame

Cleans the input DataFrame by handling missing data.

Args:

df (pd.DataFrame): The DataFrame to clean.

Returns:

pd.DataFrame: The cleaned DataFrame.

Raises:

TypeError: If the input is not a pandas DataFrame.

extract_coordinates (*df : DataFrame* , *polyline_column : str = 'POLYLINE'*) → DataFrame

Extract some features from the original columns in the given dataset.

static haversine (*lon1 : float* , *lat1 : float* , *lon2 : float | ndarray* , *lat2 : float | ndarray* , *unit : str = 'km'*) → float | ndarray

Calculate the great-circle distance between one point and multiple points on the Earth.

Parameters: - lon1 (float): Longitude of the first point in decimal degrees. - lat1 (float): Latitude of the first point in decimal degrees. - lon2 (float or np.ndarray): Longitude(s) of the second point(s) in decimal degrees. - lat2 (float or np.ndarray): Latitude(s) of the second point(s) in decimal degrees. - unit (str, optional): Unit of distance ('km', 'miles', 'nmi'). Defaults to 'km'.

Returns: - float or np.ndarray: Distance(s) between the first point and second point(s) in the specified unit.

load_districts (*districts : Dict*) → DataFrame

Convert a districts dictionary to a pandas DataFrame with calculated center coordinates.

Parameters: - districts (dict): Dictionary containing district boundary information.

Returns: - pd.DataFrame: Processed DataFrame with district boundaries and center coordinates.

parse_and_correct_polyline_coordinates (*df : DataFrame* , *polyline_column : str = 'POLYLINE'*) → DataFrame | None

Parses and corrects coordinates in the specified column of the DataFrame. Converts string representations of lists into actual lists and swaps latitude and longitude values as per standard convention.

Args:

df (pd.DataFrame): The input DataFrame. polyline_column (str): The column containing string representations of lists.

Returns:

Optional[pd.DataFrame]: The DataFrame with the specified column converted to lists and corrected coordinates.

Raises:

ValueError: If the specified column does not exist. IndexError: If the DataFrame is empty. ValueError: If a cell in the specified column cannot be parsed.

remove_missing_gps (*df : DataFrame* , *missing_data_column : str = 'MISSING_DATA'* , *missing_flag : bool = True*) → DataFrame

Remove rows from the DataFrame where the specified missing data column has the missing flag.

Parameters:

df pd.DataFrame

The input DataFrame.

missing_data_column str, optional

The name of the column that indicates missing data (default is “MISSING_DATA”).

missing_flag bool, optional

The flag value that indicates missing data (default is True).

Returns:

pd.DataFrame

The DataFrame with rows containing missing data removed.

Raises:

ValueError

If the specified missing data column does not exist in the DataFrame.

safe_convert_string_to_list (*df : DataFrame* , *polyline_column : str = 'POLYLINE'*) → list | None

Converts string representations of lists in the specified column to actual lists.

Args:

df (pd.DataFrame): The input DataFrame. polyline_column (str): The column containing string representations of lists.

Returns:

Optional[pd.DataFrame]: The DataFrame with the specified column converted to lists.

Raises:

ValueError: If the specified column does not exist. IndexError: If the DataFrame is empty. ValueError: If a cell in the specified column cannot be parsed.

separate_timestamp (*df : DataFrame* , *timestamp_column : str = 'TIMESTAMP'*) → DataFrame

Calculate the weekday from the timestamp column and add it as a new column.

Parameters:

df pd.DataFrame

The input DataFrame with a datetime timestamp column.

timestamp_column str, optional

The name of the column that contains datetime timestamp data (default is “TIMESTAMP”).

Returns:

pd.DataFrame

The DataFrame with a new column named ‘WEEKDAY’.

Raises:

ValueError

If the specified timestamp column does not exist in the DataFrame.

Feature Engineering

class FeatureEngineering. EnhancedFeatureEngineeringPipeline

Bases: `object`

A pipeline for feature engineering that includes encoding categorical variables, scaling numerical features, and converting polyline data to image representations.

Attributes:

scaler (MinMaxScaler): Scaler for numerical features. numerical_imputer (SimpleImputer): Imputer for numerical features. categorical_imputer (SimpleImputer): Imputer for categorical features. numerical_features (list): List of numerical feature names. categorical_features (list): List of categorical feature names. encoders (dict): Dictionary of OneHotEncoders for categorical features.

convert_route_to_image (*polyline : list*) → ndarray

Converts a polyline to a grayscale image representation.

Args:

polyline (list): A list of [latitude, longitude] points.

Returns:

np.ndarray: A 32x32 grayscale image representing the route.

fit (*dataframe*)

Fits the encoders and scaler on the provided training data.

Args:

dataframe (pd.DataFrame): The training data containing features to fit.

fit_transform (*dataframe : DataFrame*)

Fits the pipeline on the data and then transforms it.

Args:

dataframe (pd.DataFrame): The data to fit and transform.

Returns:

tuple: A tuple containing:

- route_images_tensor (torch.Tensor): Tensor of route images.
- additional_features_tensor (torch.Tensor): Tensor of additional engineered features.

transform (*dataframe*)

Transforms the data by encoding categorical features, scaling numerical features, and converting polyline data to image representations.

Args:

dataframe (pd.DataFrame): The data to transform.

Returns:

tuple: A tuple containing:

- route_images_tensor (torch.Tensor): Tensor of route images.
- additional_features_tensor (torch.Tensor): Tensor of additional engineered features.

class FeatureEngineering. FeatureEngineeringPipeline

Bases: `object`

A pipeline for feature engineering that includes encoding categorical variables, scaling numerical features, and converting polyline data to image representations.

Attributes:

scaler (MinMaxScaler): Scaler for numerical features. numerical_imputer (SimpleImputer): Imputer for numerical features. categorical_imputer (SimpleImputer): Imputer for categorical features. numerical_features (list): List of numerical feature names. categorical_features (list): List of categorical feature names. weekday_encoder (OneHotEncoder): Encoder for the 'WEEKDAY' feature. month_encoder (OneHotEncoder): Encoder for the 'MONTH' feature. cluster_encoder (OneHotEncoder): Encoder for 'DISTRICT_CLUSTER' and 'TIME_CLUSTER' features.

convert_route_to_image (*polyline : list*) → ndarray

Converts a polyline to a grayscale image representation.

Args:

polyline (list): A list of [latitude, longitude] points.

Returns:

np.ndarray: A 32x32 grayscale image representing the route.

convert_route_to_image_optimized (*polyline : list*) → ndarray

Optimized version of the function to convert a polyline to a grayscale image representation.

Args:

polyline (list): A list of [latitude, longitude] points.

Returns:

np.ndarray: A 32x32 grayscale image representing the route.

fit (*dataframe*)

Fits the encoders and scaler on the provided training data.

Args:

dataframe (pd.DataFrame): The training data containing features to fit.

fit_transform (*dataframe : DataFrame*)

Fits the pipeline on the data and then transforms it.

Args:

dataframe (pd.DataFrame): The data to fit and transform.

Returns:

tuple: A tuple containing:

- route_images_tensor (torch.Tensor): Tensor of route images.
- additional_features_tensor (torch.Tensor): Tensor of additional engineered features.

transform (*dataframe*)

Transforms the data by encoding categorical features, scaling numerical features, and converting polyline data to image representations.

Args:

dataframe (pd.DataFrame): The data to transform.

Returns:

tuple: A tuple containing:

- route_images_tensor (torch.Tensor): Tensor of route images.
- additional_features_tensor (torch.Tensor): Tensor of additional engineered features.

class FeatureEngineering. FeatureEngineeringPipelineWithEmbeddings

Bases: `object`

A pipeline for feature engineering that includes encoding categorical variables using embeddings, scaling numerical features, and converting polyline data to image representations.

Attributes:

scaler (MinMaxScaler): Scaler for numerical features. numerical_imputer (SimpleImputer): Imputer for numerical features. categorical_imputer (SimpleImputer): Imputer for categorical features. numerical_features (list): List of numerical feature names. categorical_features (list): List of categorical feature names. embeddings (dict): Dictionary of Embedding layers for categorical features.

convert_route_to_image (*polyline : list*) → ndarray

Converts a polyline to a grayscale image representation.

Args:

polyline (list): A list of [latitude, longitude] points.

Returns:

np.ndarray: A 32x32 grayscale image representing the route.

fit (*dataframe*)

Fits the scaler on the provided training data.

Args:

dataframe (pd.DataFrame): The training data containing features to fit.

fit_transform (*dataframe : DataFrame*)

Fits the pipeline on the data and then transforms it.

Args:

dataframe (pd.DataFrame): The data to fit and transform.

Returns:

tuple: A tuple containing:

- route_images_tensor (torch.Tensor): Tensor of route images.
- additional_features_tensor (torch.Tensor): Tensor of additional engineered features.

transform (*dataframe*)

Transforms the data by encoding categorical features using embeddings, scaling numerical features, and converting polyline data to image representations.

Args:

dataframe (pd.DataFrame): The data to transform.

Returns:

tuple: A tuple containing:

- route_images_tensor (torch.Tensor): Tensor of route images.
- additional_features_tensor (torch.Tensor): Tensor of additional engineered features.

Districts

exception **districts. DistrictLoadError**

Bases: `Exception`

districts. load_districts (*districts_path : Path*) → Dict

Loads district data from a JSON file.

Args:

districts_path (str): Path to the districts JSON file.

Returns:

dict: Districts data.

Raises:

SystemExit: If any error occurs during loading.

Cluster Districts

cluster_districts. HDBSCAN_Clustering_Aggregated_optimized (*df : DataFrame*) → DataFrame

Apply HDBSCAN clustering to polylines grouped by WEEKDAY and TIME, with aggregated membership probabilities.

Parameters:

df (pd.DataFrame):

- **Must contain the following columns:**
  - 'WEEKDAY' (int or str): Represents the day of the week.
  - 'TIME' (str or appropriate time format): Represents the time slot.
  - 'POLYLINE' (str): String representation of a list of coordinate pairs, e.g., "[[x1, y1], [x2, y2], ...]".

Returns:

pd.DataFrame: Original DataFrame augmented with the following columns:

- 'CLUSTER' (int): Assigned cluster label by HDBSCAN.
- 'DOM' (float): Dominant membership probability.
- 'PROBABILITY' (float): Aggregated membership probability.
- 'OUTLIER_SCORE' (float): Outlier score assigned by HDBSCAN.
- 'MEMBERSHIP_QUALITY' (float): Computed as DOM * (1 - OUTLIER_SCORE).

Example:

```
>>> df = pd.DataFrame({
...     'WEEKDAY': ['Monday', 'Monday'],
...     'TIME': ['Morning', 'Morning'],
...     'POLYLINE': ['[[0, 0], [1, 1], [2, 2]]',
'[[10, 10], [11, 11], [12, 12]]']
... })
>>> clustered_df =
HDBSCAN_Clustering_Aggregated_optimized(df)
```

cluster_districts. add_temporal_context (*df : DataFrame*) → DataFrame

Add temporal context to the combined cluster feature.

Args:

df (pd.DataFrame): A dataframe containing ‘COMBINED_CLUSTER’ and ‘TIME_PERIODS’ columns.

Returns:

pd.DataFrame: The input dataframe with an additional ‘REGIONAL_TEMPORAL_CONTEXT’ column.

cluster_districts. aggregate_district_clusters (*df : DataFrame*) → DataFrame

Aggregate district and regional clusters into a composite feature.

Args:

df (pd.DataFrame): A dataframe containing ‘DISTRICT_CLUSTER’ and ‘REGIONAL_CLUSTER’ columns.

Returns:

pd.DataFrame: The input dataframe with an additional ‘COMBINED_CLUSTER’ column.

cluster_districts. aggregate_historical_data (*df : DataFrame*) → DataFrame

Aggregate historical data by grouping by regional temporal context and calculating averages.

Args:

df (pd.DataFrame): A dataframe containing ‘REGIONAL_TEMPORAL_CONTEXT’ column.

Returns:

pd.DataFrame: The input dataframe with merged aggregated historical features.

cluster_districts. cluster_trip_district (*df: ~pandas.core.frame.DataFrame , districts: <module 'json' from '/usr/lib/python3.10/json/__init__.py'>*) → DataFrame

Cluster districts into predefined groups and map each district in the dataframe to its corresponding cluster.

Args:

df (pd.DataFrame): A dataframe containing a ‘DISTRICT_NAME’ column with district names.

Returns:

pd.DataFrame: The input dataframe with an additional ‘CLUSTER’ column indicating the assigned cluster for each district.

cluster_districts. cluster_trip_time (*df : DataFrame*) → DataFrame

Cluster the time of trips into three distinct periods (morning, afternoon, night) based on the ‘TIME’ column.

Args:

df (pd.DataFrame): A dataframe containing a ‘TIME’ column with time values.

Returns:

pd.DataFrame: The input dataframe with additional 'TIME_CLUSTER' and 'TIME_PERIODS' columns indicating the time cluster and corresponding time period.

cluster_districts. determine_optimal_clusters (*data* , *max_k = 10*)

cluster_districts. determine_traffic_status (*df : DataFrame* , *dom_column : str = 'DOM'* , *light_threshold : float = 0.4* , *medium_threshold : float = 0.7*) → DataFrame

Determines the traffic status based on the DOM value and manual thresholds for light, medium, and high traffic.

Args:

df (pd.DataFrame): A dataframe containing the DOM values. dom_column (str): The column name for the DOM values. light_threshold (float, optional): The threshold below which traffic is considered 'Light'. medium_threshold (float, optional): The threshold above which traffic is considered 'High'. Values in between are 'Medium'.

Returns:

pd.DataFrame: The input dataframe with a new 'TRAFFIC_STATUS' column indicating 'Light', 'Medium', or 'Heavy' traffic.

cluster_districts. determine_traffic_status_by_quality (*df : DataFrame* , *quality_column : str = 'MEMBERSHIP_QUALITY'* , *cluster_column : str = 'CLUSTER'* , *light_threshold : float = 0.4* , *medium_threshold : float = 0.7* , *categories : List [str] = ['Light', 'Medium', 'Heavy']* , *default_category : str = 'Unknown'* , *inplace : bool = False*) → DataFrame

Determines the traffic status based on the MEMBERSHIP_QUALITY value and manual thresholds for Light, Medium, and Heavy traffic.

Args:

df (pd.DataFrame): A dataframe containing the MEMBERSHIP_QUALITY and CLUSTER values. quality_column (str): The column name for the MEMBERSHIP_QUALITY values. cluster_column (str): The column name for the CLUSTER values. light_threshold (float, optional): The threshold below which traffic is considered 'Light'. medium_threshold (float, optional): The threshold above which traffic is considered 'Heavy'. Values in between are 'Medium'. categories (List[str], optional): List of category labels corresponding to the conditions. Default is ["Light", "Medium", "Heavy"]. default_category (str, optional): Label for values that do not meet any condition. Default is "Unknown". inplace (bool, optional): Whether to modify the original dataframe. If False, returns a new dataframe. Default is False.

Returns:

pd.DataFrame: The input dataframe with a new 'TRAFFIC_STATUS' column indicating 'Light', 'Medium', 'Heavy', or 'Unknown' traffic.

Example:

```
>>> import pandas as pd
>>> data = {
...     'WEEKDAY': ['Sunday', 'Thursday', 'Tuesday',
'Wednesday', 'Saturday'],
...     'TIME': [14, 15, 22, 14, 10],
...     'CLUSTER': [0.0, 8.0, -1.0, 0.0, -1.0],
...     'DOM': [0.842566, 0.971883, 0.0, 0.908505, 0.0],
...     'OUTLIER_SCORE': [0.157434, 0.028117, 0.047706,
0.091495, 0.011821],
...     'MEMBERSHIP_QUALITY': [0.709918, 0.944557, 0.0,
0.825381, 0.0]
... }
>>> df = pd.DataFrame(data)
>>> result_df = determine_traffic_status_by_quality(df)
>>> print(result_df)
    WEEKDAY  TIME  CLUSTER       DOM  OUTLIER_SCORE
MEMBERSHIP_QUALITY TRAFFIC_STATUS
0     Sunday    14      0.0  0.842566
0.157434            0.709918          Heavy
1  Thursday    15      8.0  0.971883
0.028117            0.944557          Heavy
2    Tuesday    22     -1.0  0.000000
0.047706            0.000000        Unknown
3  Wednesday    14      0.0  0.908505
0.091495            0.825381          Heavy
4   Saturday    10     -1.0  0.000000
0.011821            0.000000        Unknown
```

cluster_districts. encode_geographical_context (*df : DataFrame*) → DataFrame

Encode geographical context by clustering start and end coordinates.

Args:

df (pd.DataFrame): A dataframe containing ‘START_LAT’, ‘START_LONG’, ‘END_LAT’, ‘END_LONG’ columns.

Returns:

pd.DataFrame: The input dataframe with an additional ‘REGIONAL_CLUSTER’ column indicating the geographical cluster.

cluster_districts. extract_features (*polyline*)

Extract meaningful features from a polyline for clustering.

cluster_districts. parse_polyline (*polyline_str*)

cluster_districts. traffic_congestion_indicator (*df : DataFrame*) → DataFrame

Calculate traffic congestion indicator based on travel time and trip distance.

Args:

df (pd.DataFrame): A dataframe containing 'TRAVEL_TIME' and 'TRIP_DISTANCE' columns.

Returns:

pd.DataFrame: The input dataframe with an additional 'CONGESTION' column.

Helper Functions

helper. categorize_time (*df : DataFrame*) → str

Take an interval and maps it to a time category

helper. download_dataset ( )

helper. file_load (*file_path : str | Path*) → DataFrame

Loads a file into a pandas DataFrame.

Parameters:

file_path (Union[str, Path]): The path to the file to be loaded.

Returns:

pd.DataFrame: The loaded DataFrame.

Raises:

FileNotFoundError: If the file does not exist. ValueError: If the file format is unsupported. pd.errors.ParserError: If there's an error parsing the file.

helper. file_load_large_csv (*file_path : str | Path* , *chunksize : int = 100000* , *** read_csv_kwargs : Dict | None*) → DataFrame

Loads a large CSV file into a pandas DataFrame by reading it in chunks.

Parameters:

file_path (Union[str, Path]): The path to the CSV file to be loaded. chunksize (int, optional): Number of rows per chunk. Default is 100,000. ** read_csv_kwargs: Additional keyword arguments to pass to pandas.read_csv.

Returns:

pd.DataFrame: The concatenated DataFrame containing all data from the CSV file.

Raises:

FileNotFoundError: If the specified file does not exist. ValueError: If the file is not a CSV or if no data is read from the file. pd.errors.ParserError: If there's an error parsing the CSV file.

helper. load_config (*config_path : str = 'config.yaml'*) → dict

Load configuration from a YAML file.

Args:

config_path (str): Path to the YAML configuration file.

Returns:

dict: Parsed configuration as a dictionary.

Raises:

FileNotFoundError: If the configuration file does not exist. yaml.YAMLError: If there's an error parsing the YAML file.

helper. parse_arguments ( )

Parses command-line arguments.

Returns:

args argparse.Namespace

Parsed command-line arguments.

helper. plot_metrics (*metrics* , *output_dir = 'plots'*)

helper. plot_route_images (*route_images : Tensor* , *num_images : int = 5*)

Plots a specified number of route images to verify correctness.

Args:

route_images (torch.Tensor): Tensor of images representing routes. num_images (int): Number of images to display.

helper. read_csv_with_progress (*file_path* , *chunksize = 100000*)

Reads a CSV file with a progress bar.

Parameters: - file_path: Path to the CSV file. - chunksize: Number of rows per chunk.

Returns: - DataFrame containing the concatenated chunks.

helper. save_dataframe_if_not_exists (*df : DataFrame* , *file_path : str* , *file_format : str = 'csv'* , *** kwargs*) → bool

Save a DataFrame to a file only if the file does not already exist.

Parameters:

df pd.DataFrame

The DataFrame to save.

file_path str

The path to the file where the DataFrame should be saved.

file_format str, optional

The format to save the DataFrame in (e.g., 'csv', 'excel'). Default is 'csv'.

kwargs :
Additional keyword arguments to pass to the pandas saving method.

Returns:

bool
True if the file was saved, False if it already exists.

helper. save_dataframe_overwrite (*df : DataFrame* , *file_path : str* , *file_format : str = 'csv'* , ** *kwargs*) → None
Save a DataFrame to a file, overwriting it if it already exists.

Parameters:

df pd.DataFrame
The DataFrame to save.

file_path str
The path to the file where the DataFrame should be saved.

kwargs :
Additional keyword arguments to pass to the pandas saving method.

Returns:

None

Logger

logger. get_logger (*name : str = 'logger'*) → Logger

Get a logger with the specified name.

Args:

name (str): Name of the logger.

Returns:

logging.Logger: Configured logger instance.

logger. load_config (*config_path : str = 'config.yaml'*) → Dict

Load configuration from a YAML file.

Args:

config_path (str): Path to the YAML configuration file.

Returns:

dict: Parsed configuration as a dictionary.

Raises:

FileNotFoundError: If the configuration file does not exist. yaml.YAMLError: If there's an error parsing the YAML file.

logger. setup_logging (*config : Dict*) → None

Set up logging based on the provided configuration dictionary.

Args:

config (dict): Configuration dictionary loaded from YAML.

train Module

load_tensor (*file_path : str*) → torch.Tensor

Load a tensor from a given file path.

Args:

file_path (str): Path to the tensor file.

Returns:

torch.Tensor: Loaded tensor.

evaluate (*model* , *dataloader* , *criterion* , *device*)

Evaluate the model on a given dataset.

Args:

model (nn.Module): The model to evaluate. dataloader (DataLoader): DataLoader for the dataset. criterion (nn.Module): Loss function. device (torch.device): Device to perform computations on.

Returns:

Tuple[float, float]: Average loss and accuracy.

get_class_names (*label_encoder_path : str* , *unique_labels : np.ndarray*) → list

Retrieve class names from a LabelEncoder if available, else use label indices.

Args:

label_encoder_path (str): Path to the saved LabelEncoder. unique_labels (np.ndarray): Array of unique label indices.

Returns:

list: List of class names.