Privacy Preserving Neural Network Modelling in Insurance using Horizontal Federated Learning

Release 1.0

IFoA Federated Learning Working Party:
Malgorzata Smietanka (Chair),
Dylan Liew (Deputy),
Scott Hand,
Harry Lohhy

May 15, 2024

CONTENTS

CHAPTER

ONE

CODE

1.1 architecture module

class architecture.**MultipleRegression**(*num_features=39*, *num_units_1=25*, *num_units_2=2*, *activation=<class 'torch.nn.modules.activation.Tanh'>*, *dropout_rate=0*)

Bases: Module

A neural network model for multiple regression with two hidden layers.

Attributes:

layer_1
[nn.Linear] The first linear layer.

layer_2
[nn.Linear] The second linear layer.

layer_out
[nn.Linear] The output layer.

dropout
[nn.Dropout] Dropout layer for regularization.

act
[activation] The activation function.

Parameters:

num_features
[int] Number of input features. Default is 39.

num_units_1
[int] Number of units in the first hidden layer. Default is 25.

num_units_2
[int] Number of units in the second hidden layer. Default is 2.

activation
[callable] Activation function to use. Default is nn.Tanh.

dropout_rate
[float] Dropout rate. Default is 0.

Methods:

forward(inputs)
Performs a forward pass through the model.

predict(test_inputs)
Predicts outputs for the given inputs.

`forward`(*inputs*)

Performs a forward pass through the model.

Parameters:
inputs (Tensor): Input tensor.

Returns:
Tensor: The model's output tensor.

`predict`(*test_inputs*)

Predicts outputs for the given inputs without applying dropout.

Parameters:
test_inputs (Tensor): Input tensor for prediction.

Returns:
Tensor: The predicted output tensor.

`training: bool`

architecture.`get_parameters`(*model*) → List[ndarray]

Retrieves the parameters of the given model as a list of numpy arrays.

Parameters:
model (nn.Module): The model from which to retrieve parameters.

Returns:
List[np.ndarray]: A list containing the model's parameters as numpy arrays.

1.2 insur_FL_client module

class insur_FL_client.`ClaimsFrequencyFLClient`(**args: Any*, ***kwargs: Any*)

Bases: `NumPyClient`

A client class for federated learning using Flower framework, designed to handle the training and evaluation of a machine learning model on local data and interact with a federated learning server.

Attributes:

- model (torch.nn.Module): The local machine learning model.
- optimizer (torch.optim.Optimizer): The optimizer used for training the model.
- criterion (torch.nn.Module): The loss function used for model training.
- trainset (torch.utils.data.Dataset): The training dataset.
- valset (torch.utils.data.Dataset): The validation dataset.
- testset (torch.utils.data.Dataset): The test dataset.
- num_examples (Dict): A dictionary containing the number of examples.
- exposure (float): A parameter used to weight the parameter updates.
- noise (List): A list of noise values added to model parameters for secure aggregation.

evaluate(*parameters: List[ndarray]*, *config: Dict[str, str]*) → Tuple[float, int, Dict]

Evaluates the model with the provided global parameters on local test data.

This method is intended to be used in a federated learning context where global model parameters are evaluated on a client's local test dataset. The method sets the model's parameters to the provided global parameters, evaluates these parameters on the local test dataset, and returns the evaluation loss, the number of test samples, and a dictionary containing evaluation metrics such as accuracy.

Parameters:

- **parameters**
  [List[np.ndarray]] A list of NumPy ndarrays representing the global model parameters to be evaluated.
- **config**
  [Dict[str, str]] A dictionary containing configuration options for the evaluation process. This could include model-specific settings or evaluation hyperparameters. Note: Currently, this parameter is not directly used in the method but is included for consistency and future extensions.

Returns:

Tuple[float, int, Dict]

A tuple containing three elements:

- The evaluation loss as a float.
- The number of samples in the test dataset as an int.
- A dictionary containing evaluation metrics, with at least an "accuracy" key providing the accuracy of the model on the test dataset calculated as the loss divided by the number of test loader batches.

Examples:

```
>>> client_evaluator = ModelEvaluator(model, criterion, testset)
>>> global_params = [...]  # Global parameters obtained from the server
>>> loss, num_samples, metrics = client_evaluator.evaluate(global_params, {}
↪)
>>> print(f"Test Loss: {loss}, Test Accuracy: {metrics['accuracy']}")
```

Notes:

- The method utilizes a DataLoader to iterate through the test dataset, and the batch size for the DataLoader is determined by the global BATCH_SIZE variable.
- The accuracy calculation in the returned dictionary is a simplified example. Depending on the model and the task, you might need a more sophisticated method to calculate accuracy or other relevant metrics.
- Ensure that the global *BATCH_SIZE* variable is appropriately set for the evaluation DataLoader to function correctly.

fit(*parameters: List[ndarray]*, *config: Dict[str, str]*) → Tuple[List[ndarray], int, Dict]

Trains the model locally using provided parameters and configuration settings.

This method sets the initial model parameters, trains the model on a local dataset, and returns the updated model parameters after training. It encapsulates the process of local training within a federated learning framework, including setting initial parameters, executing the training loop, and optionally evaluating the model on a validation set.

Parameters:

parameters
[List[np.ndarray]] A list of NumPy ndarrays representing the model parameters to be set before training begins. These parameters might come from a central server in a federated learning setup.

config
[Dict[str, str]] A dictionary containing configuration options for the training process. This may include hyperparameters such as learning rate, batch size, or any other model-specific settings.

Returns:

Tuple[List[np.ndarray], int, Dict]

A tuple containing three elements:

- A list of NumPy ndarrays representing the updated model parameters after training.
- An integer representing the number of training samples used in the training process. This could be used for weighted averaging in a federated learning setup.
- A dictionary containing additional information about the training process. For example, it could include metrics such as training loss or accuracy, or model-specific metrics like 'exposure' in this case.

Examples:

```
>>> model_trainer = ModelTrainer(model, optimizer, criterion, trainset,␣
↪valset, testset)
>>> updated_params, num_samples, metrics = model_trainer.fit(initial_params,
↪ config)
>>> print(metrics)
{'exposure': ...}
```

Notes:

- The training process uses the DataLoader from PyTorch to load the training and validation datasets with the specified batch size. It's important to ensure that the datasets are properly initialized and passed to the ModelTrainer before calling *fit*.
- The configuration dictionary must include all necessary settings required by the training and evaluation process. Missing configurations might result in default values being used or in runtime errors.
- The method internally calls *set_parameters* to set the model's initial parameters and *get_parameters* to retrieve the updated parameters after training. Ensure that these methods are implemented correctly for the *fit* method to work as expected.
- This method appends the training statistics to an internal list *self.stats* after each training session, allowing for tracking of performance over multiple rounds of federated learning.

`get_parameters`(*config*) → List[ndarray]

Retrieves local model parameters, with optional quantization and noise addition based on configuration flags.

This method extracts the model's parameters as numpy arrays. If the QUANTISATION flag is set in the provided configuration, the parameters are quantized. Similarly, if the SMPC_NOISE flag is set, noise is added to the parameters. This is part of preparing the model's parameters for secure multi-party computation (SMPC) or other privacy-preserving mechanisms.

Parameters:

config
[dict] A configuration dictionary that may contain flags like QUANTISATION and SMPC_NOISE to indicate whether quantization or noise addition should be applied to the model parameters.

Returns:

List[np.ndarray]
A list of numpy arrays representing the model's parameters after applying quantization and/or noise addition as specified in the configuration. Each numpy array in the list corresponds to parameters of a different layer or component of the model.

Examples:

```
>>> model = YourModelClass()
>>> config = {'QUANTISATION': True, 'SMPC_NOISE': False}
>>> parameters = model.get_parameters(config)
>>> type(parameters)
<class 'list'>
>>> type(parameters[0])
<class 'numpy.ndarray'>
```

Notes:

- QUANTISATION and SMPC_NOISE are flags handling quantization and noise addition.

set_parameters(*parameters: List[ndarray]*) → None

Updates the model's parameters with new values provided as a list of NumPy ndarrays.

This method takes a list of NumPy arrays containing new parameter values and updates the model's parameters accordingly. It's typically used to set model parameters after they have been modified or updated elsewhere, possibly after aggregation in a federated learning scenario or after receiving updates from an optimization process.

Parameters:

parameters
[List[np.ndarray]] A list of NumPy ndarrays where each array corresponds to the parameters for a different layer or component of the model. The order of the arrays in the list should match the order of parameters in the model's state_dict.

Returns:
None

Examples:

```
>>> model = YourModelClass()
>>> new_parameters = [np.array([[0.1, 0.2], [0.3, 0.4]]), np.array([0.5, 0.
↪6])]
>>> model.set_parameters(new_parameters)
>>> # Model parameters are now updated with `new_parameters`.
```

Notes:

- This method assumes that the provided list of parameters matches the structure and order of the model's parameters. If the order or structure of *parameters* does not match, this may lead to incorrect assignment of parameters or runtime errors.
- The method converts each NumPy ndarray to a PyTorch tensor before updating the model's state dict. Ensure that the data types and device (CPU/GPU) of the NumPy arrays are compatible with your model's requirements.

insur_FL_client.**initialize_model**()

Initializes and returns the machine learning model along with the optimizer and loss criterion.

This function initializes a multiple regression model with architecture specified by *archit.MultipleRegression*. The model configuration (number of features and units in the first and second layer) is read from *run_config*. The model is moved to a device (e.g., CPU or GPU) specified by the global *device* variable. The optimizer used is NAdam, and the loss criterion is Poisson negative log likelihood.

Note:

- The function relies on global variables *device*, *run_config*, *archit*, *optim*, and *nn* for its operation.
- *run_config* should have *NUM_FEATURES*, *NUM_UNITS_1*, and *NUM_UNITS_2* attributes defined.
- *device* should be defined globally and indicate where the model should be allocated (e.g., 'cuda' or 'cpu').

Returns:

tuple: A tuple containing three elements:

- model (archit.MultipleRegression): The initialized multiple regression model.
- optimizer (optim.NAdam): The optimizer for the model.
- criterion (nn.PoissonNLLLoss): The loss criterion.

Example:

```
>>> model, optimizer, criterion = initialize_model()
>>> print(type(model))
<class 'archit.MultipleRegression'>
>>> print(type(optimizer))
<class 'torch.optim.nadam.NAdam'>
>>> print(type(criterion))
<class 'torch.nn.modules.loss.PoissonNLLLoss'>
```

insur_FL_client.**main**()

Main function to execute the training process.

This function begins by parsing command-line arguments to configure the training session. It then initializes the random seeds for reproducibility and processes the specified client's dataset. Based on the provided arguments, it either proceeds with federated learning or exits if centralized training is indicated.

Federated learning involves initializing a model, optimizer, and loss criterion, followed by applying noise to the training process for privacy preservation. The model is then trained with data from the specified client, communicating with a federated learning server as configured.

Finally, the trained model's state is saved, and training loss statistics are written to a file.

Returns:
int: 0, indicating successful completion of the training process.

Note:
This function relies on external configurations from *run_config*, utility functions from *utils*, and the *Claims-FrequencyFLClient* for the FL training setup. It assumes the presence of a federated learning server listening on the specified address.

Example usage:
To run this script, ensure that the command-line arguments are correctly set for the desired training configuration. For example: *python3 insur_FL_client.py –agent_id 1 –if_FL 1* This would initiate federated learning for client with agent_id 1.

insur_FL_client.**parse_args**()

Parses command-line arguments for training configuration.

This function uses *argparse* to define and parse command-line arguments necessary for specifying the training mode (centralized or federated learning), the number of federated learning participants, and the partition of data to use for training.

Returns:

Namespace: An argparse.Namespace object containing the arguments and their values. - *agent_id* (int): Specifies the partition of data for training. A value of -1 indicates that all data should be used (training a global model). Valid values are in the range [-1, run_config.server_config["num_clients"]]. - *if_FL* (int): Determines the pipeline type. A value of 0 sets the training to centralized (currently disabled), while a value of 1 sets it to federated learning.

Example command line usage:

For federated learning with all data: *python script.py –agent_id -1 –if_FL 1*

For federated learning for agent 3: *python script.py –agent_id 3 –if_FL 1*

insur_FL_client.**test**(*model*, *criterion*, *val_loader*) → float

Evaluates the performance of the model on a validation dataset.

This function iterates over the provided validation DataLoader, computes the loss of the model predictions against the true labels using the provided loss criterion, and sums up the loss over all validation batches to get the total validation loss. The model is set to evaluation mode during this process to disable dropout or batch normalization layers that behave differently during training.

Parameters:

model

[torch.nn.Module] The neural network model to be evaluated. It should already be trained or loaded with pre-trained weights.

criterion

[torch.nn.Module] The loss function used to calculate the loss between the model predictions and the true labels.

val_loader

[torch.utils.data.DataLoader] A DataLoader providing batches of validation data including features and labels.

Returns:

float

The total loss computed over all batches of the validation dataset.

Example:

```
>>> model = MyCustomModel()
>>> criterion = torch.nn.MSELoss()
>>> val_dataset = CustomDataset(...)
>>> val_loader = DataLoader(val_dataset, batch_size=64)
>>> total_val_loss = test(model, criterion, val_loader)
>>> print(f'Total Validation Loss: {total_val_loss}')
```

Notes:

- The function automatically moves the input and target data to the same device as the model before making predictions.

Ensure that the model and criterion are already moved to the appropriate device (CPU or GPU) before calling this function. - The function uses *torch.no_grad()* context manager to disable gradient computation during evaluation, improving memory efficiency and speed. - It's important to call *model.eval()* before evaluating the model to set the model to evaluation mode. This is necessary for models that have layers like dropout or batch normalization that behave differently during training and evaluation.

insur_FL_client.**train**(*model*, *optimizer*, *criterion*, *train_loader: DataLoader*, *val_loader: DataLoader*, *epochs=10*)

Trains the given model using the specified training and validation data loaders, optimizer, and loss function across a defined number of epochs. Evaluates the model on the validation dataset after each training epoch and reports the training and validation losses.

Parameters:

model
[torch.nn.Module] The neural network model to be trained.

optimizer
[torch.optim.Optimizer] The optimizer used for adjusting the model parameters based on the computed gradients.

criterion
[torch.nn.Module] The loss function used to evaluate the goodness of the model's predictions.

train_loader
[torch.utils.data.DataLoader] DataLoader for the training data, providing batches of data.

val_loader
[torch.utils.data.DataLoader] DataLoader for the validation data, used to assess the model's performance.

epochs
[int, optional] The number of complete passes through the training dataset. Defaults to the global variable *EPOCHS*.

Returns:

Tuple[torch.nn.Module, Dict[str, List[float]]]
A tuple containing: - The trained model. - A dictionary with keys 'train' and 'val', each mapping to a list of loss values recorded at the end of each epoch.

Example:

```
>>> model = MyCustomModel()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
>>> criterion = torch.nn.CrossEntropyLoss()
>>> train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
>>> val_loader = DataLoader(val_dataset, batch_size=64)
>>> trained_model, loss_stats = train(model, optimizer, criterion, train_loader,
↪ val_loader, epochs=10)
>>> print(loss_stats['train'])
>>> print(loss_stats['val'])
```

Notes:

- This function assumes that the *model*, *optimizer*, *criterion*, and data loaders have been initialized

before being passed as arguments. - The function sets the model in training mode (*model.train()*) at the beginning of each epoch and uses the optimizer to update model parameters based on the computed gradients. - Losses for both training and validation phases are accumulated over each epoch and reported at the

end. - It's important to ensure that the device (*cpu* or *cuda*) is correctly configured for the model, data, and criterion before calling this function.

1.3 insur_FL_server module

insur_FL_server.**init_parameters**() → flwr.common.Parameters

Initialize model parameters using Xavier initialization and return them in a format suitable for FL.

Returns:

fl.common.ParametersRes: The model's parameters formatted for the Flower FL framework.

insur_FL_server.**start_FL_server**()

Starts a Federated Learning (FL) server with a custom aggregation strategy.

This function initializes the random seed for reproducibility, sets up a custom strategy for aggregating updates from clients participating in the federated learning process, and starts the FL server. The server listens for client connections and orchestrates the federated learning process based on the specified strategy and server configuration.

The aggregation strategy, *LocalUpdatesStrategy*, is configured to require participation from all clients for both training (fit) and evaluation phases. It also initializes the model parameters for the federated learning process.

The server configuration specifies the number of federated learning rounds to be conducted.

External Dependencies:

- *utils.seed_torch*: Ensures reproducible results by setting the random seed.
- *insur_secAgg.LocalUpdatesStrategy*: A custom class for the aggregation strategy.
- *fl.server.start_server*: Function to initiate the FL server.
- *run_config.server_config*: Contains server-side configurations such as *num_clients* and *num_rounds*.
- *init_parameters()*: Function to initialize model parameters for the federated learning process.

Note:

The server address is hardcoded to "[::]:8080", indicating that the server will listen on all available interfaces and the port 8080.

Example:

To start the FL server, simply call this function from your script:

```
`python3 start_FL_server()`
```

1.4 insur_maskedAgg module

class insur_maskedAgg.**LocalUpdatesStrategy**(**args: Any*, ***kwargs: Any*)

Bases: FedAvg

A strategy for federated averaging that considers local updates with optional quantization.

Extends the FedAvg strategy from Flower to aggregate fit results using custom logic, including support for quantized model parameter aggregation.

aggregate_fit(*server_round: int*, *results: List[Tuple[flwr.server.client_proxy.ClientProxy, flwr.common.FitRes]]*, *failures: List[Tuple[flwr.server.client_proxy.ClientProxy, flwr.common.FitRes] | BaseException]*) → Tuple[flwr.common.Parameters | None, Dict[str, flwr.common.Scalar]]

Aggregate fit results using weighted average with support for quantized parameters.

Parameters:

server_round (int): The current round of the server. results (List[Tuple[ClientProxy, FitRes]]): The results from clients' local training. failures (List[Union[Tuple[ClientProxy, FitRes], BaseException]]): A list of clients that failed to return results.

Returns:

Tuple[Optional[Parameters], Dict[str, float]]: Aggregated parameters and aggregated metrics.

insur_maskedAgg.**aggregate_qt**(*results: List[Tuple[flwr.common.NDArrays, int64]]*) → flwr.common.NDArrays

Compute the weighted average of quantized model parameters and dequantize the result.

Parameters:

results (List[Tuple[NDArrays, int]]): A list of tuples containing quantized model parameters (NDArrays) and the number of examples (int) used for training.

Returns:

NDArrays: The weighted and dequantized average of model parameters.

insur_maskedAgg.**aggregate_qt2**(*results: List[Tuple[flwr.common.NDArrays, int]]*) → flwr.common.NDArrays

insur_maskedAgg.**aggregate_weighted_average**(*results: List[Tuple[flwr.common.NDArrays, int]]*) → flwr.common.NDArrays

Compute the weighted average of model parameters.

Parameters:

results (List[Tuple[NDArrays, int]]): A list of tuples where each tuple contains model parameters (NDArrays) and the number of examples (int) used for training.

Returns:

NDArrays: The weighted average of model parameters.

1.5 prepare_dataset module

prepare_dataset.**main**()

Main execution function that orchestrates the dataset preparation and validation for federated learning.

prepare_dataset.**parse_arguments**() → Namespace

Parse command line arguments for federated learning dataset preparation and validation.

Returns:

argparse.Namespace An object containing the parsed command line arguments with the following attributes

num_agents

[int] Specifies the number of agents among which the dataset will be partitioned.

prepare_dataset.**prepare_and_validate_dataset**(*num_agents: int*)

Prepares the dataset for federated learning by partitioning it uniformly (by number of records) among the specified number of agents. Performs validation checks to ensure the integrity of the dataset partitioning.

Parameters:

num_agents
[int] The number of agents among which the dataset will be partitioned.

1.6 skorch_tuning module

skorch_tuning.**main**()

1.7 utils module

utils.**create_df_sum**(*df_test_pred*, *factor*, *NUM_AGENTS*)

Create a summary DataFrame aggregating predictions by binned factors.

Parameters:

df_test_pred
[pandas.DataFrame] The DataFrame with test predictions.

factor
[str] The factor for binning.

NUM_AGENTS
[int] The number of agents.

Returns:

df_sum
[pandas.DataFrame] The summary DataFrame aggregated by binned factors.

utils.**create_df_test_pred**(*df_test*, *X_test*, *NUM_AGENTS*, *global_model*, *fl_model*, *agent_model_dictionary*)

Generate predictions for the test dataset using various models.

Parameters:

df_test
[pandas.DataFrame] The test dataset.

X_test
[numpy.ndarray] The features of the test dataset.

NUM_AGENTS
[int] The number of agents.

global_model
The global model for prediction.

fl_model
The federated learning model for prediction.

agent_model_dictionary
[dict] A dictionary containing agent models.

Returns:

df_test
[pandas.DataFrame] The test dataset with predictions appended.

utils.**create_test_data**()

Create test data for evaluation.

This function loads test data, undummifies categorical variables, applies scaling to certain features, bins numerical factors, and returns processed test datasets for evaluation.

Returns:

X_test
[pandas.DataFrame] Test features dataset.

y_test
[pandas.DataFrame] Test labels dataset.

df_test
[pandas.DataFrame] Processed test dataset.

utils.**double_lift_rebase**(*df_test_pred*, *model1*, *model2*)

Generate a double lift chart comparing the performance of two models.

Parameters:

df_test_pred
[pandas.DataFrame] The DataFrame with test predictions.

model1
[str] The name of the first model.

model2
[str] The name of the second model.

Returns: - None

utils.**encode_and_scale_dataframe**(*df*)

Encodes categorical variables and scales numerical features within the DataFrame.

Parameters:

df
[DataFrame] The DataFrame to encode and scale.

Returns:

DataFrame
The encoded and scaled DataFrame.

MinMaxScaler
The scaler used for numerical feature scaling.

Usage: ` `df_encoded, scaler = encode_and_scale_dataframe(df_preprocessed)` `

utils.**frequency_conversion**(*FACTOR*, *df*, *freq_dictionary*)

Perform frequency conversion on a DataFrame.

Parameters:

FACTOR
[str] The factor to be converted.

df
[pandas.DataFrame] The DataFrame containing the data.

freq_dictionary
[dict] A dictionary mapping factor keys to frequency keys.

Returns:

df
[pandas.DataFrame] The DataFrame with frequency conversion applied.

utils.**hyperparameter_counts**(*dataframe*, *hyperparameter*, *x_label*, *title*, *name*)

Plots and saves a bar chart of the value counts for a specified hyperparameter in a given DataFrame.

This function visualizes the distribution of values for a selected hyperparameter within a dataset, highlighting the frequency of each unique value. The resulting bar chart is saved to a file.

Parameters:

dataframe
[pandas.DataFrame] The DataFrame containing the data from which to count the hyperparameter values.

hyperparameter
[str] The name of the column in *dataframe* representing the hyperparameter whose distribution is to be plotted.

x_label
[str] The label for the x-axis of the plot, typically the name of the hyperparameter.

title
[str] The title of the plot, describing what the plot shows.

name
[str] The filename under which the plot will be saved. The plot is saved in the '../results/' directory. The '.png' extension is recommended to be included in the *name* for clarity.

Examples:

```
>>> df = pd.DataFrame({'model_depth': [2, 3, 4, 2, 3, 3, 2]})
>>> hyperparameter_counts(df, 'model_depth', 'Model Depth', 'Distribution of␣
↪Model Depths', 'model_depth_distribution.png')
# This will create and save a bar chart visualizing the frequency of different␣
↪model depths in the dataset.
```

Notes:

- The plot is saved with a white background to ensure readability when viewed on various devices.
- Ensure the '../results/' directory exists before calling this function, or adjust the save path accordingly.
- The function does not return any value. It directly saves the generated plot to a file.

utils.**load_individual_data**(*agent_id*, *include_val_in_train=False*)

Loads individual or global datasets as PyTorch TensorDatasets, with an option to include validation data in the training set.

This function dynamically loads training, validation, and test data from CSV files located in a specified directory. It can load data for a specific agent by ID or global data if the agent ID is set to -1. There is an option to merge training and validation datasets for scenarios where validation data should be included in training, e.g., for certain types of model tuning.

Parameters:

agent_id
[int] The identifier for the agent's dataset to load. If set to -1, global datasets are loaded.

include_val_in_train
[bool, optional] Determines whether validation data is included in the training dataset. Default is False.

Returns:

tuple
A tuple containing the training dataset, validation dataset, test dataset, column names of the training features, a tensor of test features, and the total exposure calculated from the training (and optionally validation) dataset.

Examples:

```
>>> train_dataset, val_dataset, test_dataset, column_names, test_features,␣
↪exposure = load_individual_data(-1, True)
>>> print(f"Training dataset size: {len(train_dataset)}")
```

utils.**load_individual_skorch_data**(*agent_id*)

Loads training, validation, and test datasets for a specified agent or for global model training.

This function reads the datasets from CSV files. If *agent_id* is -1, it loads the global datasets. Otherwise, it loads the agent-specific datasets based on the provided *agent_id*.

Parameters:

agent_id
[int] The identifier for the specific agent's dataset to load. If set to -1, the function loads the global training, validation, and test datasets.

Returns:

tuple
A tuple containing the training features (X_train_sc), training labels (y_tr), validation features (X_val_sc), validation labels (y_vl), test features (X_test_sc), test labels (y_te), column names of the training features (X_column_names), and the total exposure from the training set (exposure).

Examples:

```
>>> X_train, y_train, X_val, y_val, X_test, y_test, column_names, exposure =␣
↪load_individual_skorch_data(-1)
>>> print(f"Training data shape: {X_train.shape}")
```

utils.**load_model**(*agent=-1*, *num_features=39*)

Load a pre-trained neural network model for a specific agent.

Parameters:

agent
[int] The ID of the agent whose model to load. Default is -1.

num_features
[int] The number of input features for the model. Default is NUM_FEATURES.

Returns:

loaded_agent_model
[NeuralNetRegressor] The loaded neural network model for the specified agent.

utils.**lorenz_curve**(*y_true*, *y_pred*, *exposure*)

Calculates the Lorenz curve for given true values, predicted values, and exposures.

The Lorenz curve is a graphical representation of the distribution of income or wealth. In this context, it is used to show the distribution of claims or losses in insurance, ordered by predicted risk. This function calculates the cumulative percentage of claims and exposures, sorted by the predicted risk.

Parameters:

y_true
[array_like] The true values of the claims or losses.

y_pred
[array_like] The predicted risk scores associated with each claim or loss.

exposure
[array_like] The exposure values associated with each observation.

Returns:

tuple of numpy.ndarray
A tuple containing two arrays: the cumulative percentage of exposure and the cumulative percentage of claims, both sorted by the predicted risk.

Examples:

```
>>> y_true = np.array([100, 50, 20])
>>> y_pred = np.array([0.2, 0.5, 0.1])
>>> exposure = np.array([1, 2, 1])
>>> cumulated_exposure, cumulated_claims = lorenz_curve(y_true, y_pred,␣
↪exposure)
>>> print(cumulated_exposure)
>>> print(cumulated_claims)
```

utils.**one_way_graph_comparison**(*factor*, *df_test_pred*, *agents_to_graph_list*, *NUM_AGENTS*)

Generate a one-way graph comparison of actual vs. predicted frequencies by agents.

Parameters:

factor
[str] The factor for binning.

df_test_pred
[pandas.DataFrame] The DataFrame with test predictions.

agents_to_graph_list
[list] List of agent indices to include in the graph.

NUM_AGENTS
[int] The total number of agents.

Returns: None

utils.**preprocess_dataframe**(*df*)

Applies preprocessing steps to the dataframe, including shuffling, data type transformations, and value capping based on specified criteria.

Parameters:

df
[DataFrame] The pandas DataFrame to preprocess.

Returns:

DataFrame
The preprocessed DataFrame.

Usage: ` `df_preprocessed = preprocess_dataframe(df)` `

utils.**row_check**(*agents: int = 10*)

Validates the integrity of the dataset across multiple agents by checking the total number of rows, the sum of exposure values, and the sum of claims across training, validation, and test datasets.

This function ensures that the combined dataset from multiple agents, along with the test dataset, matches expected values for total row count, total exposure, and total claims. These checks are critical for verifying data integrity and consistency before proceeding with further data analysis or model training.

Parameters:

agents
[int, optional] The number of agents (or partitions) for which training and validation datasets are available. Defaults to 10.

Raises:

AssertionError
If the total number of rows, total exposure, or total claims do not match expected values, an AssertionError is raised indicating which specific integrity check has failed.

Notes:

- Assumes existence of CSV files in '../data/' following specific naming conventions.
- Useful for data preprocessing in machine learning workflows involving multiple sources or agents.
- 'Exposure' and '0' are assumed to be column names in the respective CSV files for exposure and claims.

Example:

```
>>> row_check(agents=5)
# Checks datasets for 5 agents, along with the test dataset, and prints the␣
↪status of each check.
```

utils.**seed_torch**(*seed=300*)

Seeds the random number generators of PyTorch, NumPy, and Python's *random* module to ensure reproducibility of results across runs when using PyTorch for deep learning experiments.

This function sets the seed for PyTorch (both CPU and CUDA), NumPy, and the Python *random* module, enabling CuDNN benchmarking and deterministic algorithms. It is crucial for experiments requiring reproducibility, like model performance comparisons. Note that enabling CuDNN benchmarking and deterministic operations may impact performance and limit certain optimizations.

Parameters:

seed
[int, optional] The seed value to use for all random number generators. The default value is *SEED*, which should be defined beforehand.

Returns:

None
This function does not return a value but sets the random seed for various libraries.

Notes:

- When using multiple GPUs, *th.cuda.manual_seed_all(seed)* ensures all GPUs are seeded,

crucial for reproducibility in multi-GPU setups.

Example:

```
>>> SEED = 42
>>> seed_torch(SEED)
```

utils.**split_data**(*df_encoded*)

Splits the encoded DataFrame into training, validation, and test sets.

Parameters:

df_encoded
[DataFrame] The encoded DataFrame from which to split the data.

Returns:

tuple
Contains training, validation, and test sets (X_train, X_val, X_test, y_train, y_val, y_test).

Usage: ` `X_train, X_val, X_test, y_train, y_val, y_test = split_data(df_encoded)` `

utils.**training_loss_curve**(*estimator*, *agent_id*)

Plots the training and validation loss curves along with the percentage of Poisson Deviance Explained (PDE).

Parameters:

estimator
[object] The trained model or estimator that contains the training history. It is expected to have a 'history' attribute that is a NumPy array or a similar structure with 'train_loss', 'valid_loss', and 'weighted_PDE_best' columns.

agent_id
[int or str] Identifier for the agent. Used for titling the plot and naming the saved figure file.

Notes:

- This function saves the generated plot as a PNG file in a directory named after the agent.
- Ensure the directory '../ag_{agent_id}/' exists or adjust the save path accordingly.
- The function uses matplotlib for plotting and requires this library to be installed.

utils.**undummify**(*df*, *prefix_sep='_'*)

Reverse one-hot encoding (dummy variables) in a DataFrame.

Parameters:

- df (pandas.DataFrame): The DataFrame containing dummy variables.
- prefix_sep (str, optional): Separator used in column prefixes. Default is "_".

Returns:
undummified_df (pandas.DataFrame): The DataFrame with one-hot encoding reversed.

utils.**uniform_partitions**(*agents: int = 10*, *num_features: int | None = None*)

Splits and saves the dataset into uniform partitions for a specified number of agents.

This function loads a dataset via a previously defined *upload_dataset* function, then partitions the training and validation datasets uniformly across the specified number of agents. Each partition is saved to CSV files, containing both features and labels for each agent's training and validation datasets.

Parameters:

agents
[int, optional] The number of agents to split the dataset into. Defaults to 10.

num_features
[int, optional] The number of features in the dataset. Automatically inferred if not specified.

Notes:

- Requires *upload_dataset* and *seed_torch* to be defined and accessible within the scope.
- Saves partitioned data files in the '../data/' directory.

Example:

```
>>> uniform_partitions(agents=5)
Creates and saves 5 sets of training and validation data for 5 agents, storing␣
↪them in '../data/'.
```

Raises:

FileNotFoundError
If the '../data/' directory does not exist or cannot be accessed.

Returns:

None
The function does not return a value but saves partitioned datasets to disk.

utils.**upload_dataset**()

Uploads, preprocesses, encodes, scales, and splits the dataset into training, validation, and test sets.

Assumes the existence of a global *DATA_PATH* variable pointing to the dataset's location and a *SEED* for reproducibility.

Returns:

tuple
Contains the training, validation, and test sets, feature names, and the scaler.

Usage: ` X_train, X_val, X_test, y_train, y_val, y_test, feature_names, scaler = upload_dataset() `

:exclude-members: load_model, frequency_conversion, undummify, create_test_data, create_df_test_pred, create_df_sum, one_way_graph_comparison, double_lift_rebase

1.8 utils_quantisation module

utils_quantisation.**add_mod**(*x: ndarray[Any, dtype[int64]]*, *y: ndarray[Any, dtype[int64]]*) → ndarray[Any, dtype[int64]]

Performs modular addition on two integer numpy arrays with a predefined modulus.

Parameters:

x
[NDArrayInt] The first integer numpy array.

y
[NDArrayInt] The second integer numpy array.

Returns:

NDArrayInt
The result of modular addition of *x* and *y*.

utils_quantisation.**dequantize**(*quantized_parameters: List[ndarray[Any, dtype[int64]]]*, *clipping_range: float*, *target_range: int64*, *ag_no: int*) → List[ndarray[Any, dtype[float64]]]

Dequantizes a list of integer numpy arrays back into float numpy arrays, adjusting for a specified clipping range.

Parameters:

quantized_parameters
[List[NDArrayInt]] The list of quantized integer numpy arrays to be dequantized.

clipping_range
[float] The clipping range to adjust the dequantized values within.

target_range
[np.int64] The original target range used for quantization.

ag_no
[int] A factor to adjust the dequantization process, potentially representing the number of aggregating agents.

Returns:

List[NDArrayFloat]
The list of dequantized float numpy arrays.

utils_quantisation.**dequantize_mean**(*quantized_parameters: List[ndarray[Any, dtype[int64]]]*, *clipping_range: float*, *target_range: int64*, *ag_no: int*) → List[ndarray[Any, dtype[float64]]]

Dequantizes integer Numpy arrays back to float Numpy arrays with an adjusted range based on clipping range, target range, and the number of agents.

This function reverses the quantization process applied to the original float arrays, taking into account the clipping range, target range, and an adjustment factor derived from the number of agents. It's used to restore the approximate original float values from their quantized integer representations, applying an average based on the number of agents to adjust the quantization scale.

Parameters:

quantized_parameters
[List[NDArrayInt]] A list of quantized integer Numpy arrays to be dequantized.

clipping_range
[float] The maximum absolute value allowed in the original float arrays.

target_range
[np.int64] The target range used during the quantization process.

ag_no
[int] The number of agents involved, which affects the dequantization scale.

Returns:

List[NDArrayFloat]
A list of dequantized float Numpy arrays, with values approximately restored to their original scale and centered around the original clipping range.

Example:

```
>>> quantized_parameters = [np.array([1000, -1000, 2000], dtype=np.int64)]
>>> clipping_range = 1.0
>>> target_range = 10000
>>> ag_no = 5
```

(continues on next page)

(continued from previous page)

```
>>> dequantized = dequantize_mean(quantized_parameters, clipping_range, target_
↪range, ag_no)
>>> print(dequantized)
[array([-0.2, 0.2, -0.4], dtype=float64)]
```

utils_quantisation.**quantize**(*parameters: List[ndarray[Any, dtype[float64]]]*, *clipping_range: float*, *target_range: int64*) → List[ndarray[Any, dtype[int64]]]

Quantizes a list of float numpy arrays into integer numpy arrays within a specified target range.

Parameters:

parameters
[List[NDArrayFloat]] The list of float numpy arrays to be quantized.

clipping_range
[float] The range within which values are clipped before quantization.

target_range
[np.int64] The integer range to quantize the values into.

Returns:

List[NDArrayInt]
The list of quantized integer numpy arrays.

1.9 utils_smpc module

utils_smpc.**calc_noise**(*file_path: str*, *agent_number: int*) → dict

Calculate noise based on seeds stored in a CSV file for a specific agent.

This function reads a CSV file containing seeds for each collaborator by round, generates noise vectors for each round, and applies quantization if enabled in the run configuration. It differentiates the noise for the specific agent from others.

Parameters:

file_path
[str] The path to the CSV file containing the seeds.

agent_number
[int] The index of the current agent for whom the noise is being calculated.

Returns:

dict
A dictionary with round numbers as keys and noise vectors as values.

Example:

```
>>> noises = calc_noise('path/to/seeds.csv', 2)
>>> print(noises[1])  # Noise vector for round 1
```

utils_smpc.**calc_noise_zero**(*file_path: str*, *agent_number: int*) → dict

Generate a dictionary of zero noise vectors for each round based on the CSV file.

Parameters:

file_path
[str] The path to the CSV file containing the seeds.

agent_number
[int] The agent number, unused in this function but maintained for API consistency.

Returns:

dict
A dictionary with round numbers as keys and zero noise vectors as values.

Example:

```
>>> zero_noises = calc_noise_zero('path/to/seeds.csv', 2)
>>> print(zero_noises[1])  # Zero noise vector for round 1
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX