MEG Algorithm Documentation Summary

1. Introduction

The Masked Ensemble Generator (MEG) is an advanced machine learning algorithm tailored to generate high-quality synthetic tabular data. It innovatively extends Generative Adversarial Networks (GANs) to handle the complexities of tabular datasets containing numerical, categorical, and binary features.

Key Features

Overcomes traditional GAN limitations in handling diverse tabular data.

Introduces masking and multiple generators for improved performance.

Generates many models through a combination of predictions from diverse learners and leveraging the strengths of different models

Masks can dynamically adapt during training enabling robust performance across different datasets and domains and achieving higher accuracy

Designed to work efficiently on large datasets, simultaneously generates and trains multiple models. Scalability allows it to process complex and high dimensional data effectively

Through masking can deal with missing or incomplete data without any serious performance degradation. Data irregularities are common in real world applications

Masks can be customised making MEG flexible and tailored to specific tasks

Use Cases:

Not limited to:

Healthcare analytics

Financial Forecasting

Recommender Systems

Fraud Detection

Cyber Security

Environmental Monitoring

2. Model Overview

2.1 Architecture

Masked Generators: Specialized generators produce specific data subsets based on masks.

Discriminator: Evaluates both real and synthetic data for adversarial training.

MEG typically involves a set of stages beginning with

Input Data Preprocessing:

Raw Data is preprocessed

Mask Generation:

Random features or data points are selected and masks are generated based on data relevance

Model Training with Masks:

A separate model is trained for each mask with each model specialising in predictions based on the masked data. Models types are based on use cases but can be decision trees, neural networks or other algorithms

Combination of Predictions:

Individual model outputs are combined with the aggregated ensemble output serving as the final prediction

2.2 Why Masking?

Masking divides the dataset into smaller subsets for each generator, simplifying the generation process and improving feature-specific accuracy.

2.3 Advantages

Enhanced data diversity.

Scalability for large datasets.

Superior feature-specific generation accuracy.

3. Class Details

3.1 MEG_Adapter.py

Main interface for:

Setting up and managing generators/discriminator.

Training and generating synthetic data.

Key Methods:

__init__(): Initializes model components.

load_model(): Loads or initializes generator models.

load_data(data): Prepares input data for training.

train(num_epochs, batch_size): Implements adversarial training.

generate(num_samples): Generates synthetic data.

3.2 MaskedGenerator

Generates data for specific features using a mask and a neural network.

Key Method:

forward(x): Processes input noise and applies the feature mask.

3.3 Discriminator

Binary classifier that differentiates between real and synthetic data.

Key Method:

forward(x): Predicts whether input data is real or synthetic.

4. Training Process

4.1 Adversarial Learning

Generators create synthetic data subsets.

Discriminator evaluates real vs. synthetic data.

Iterative optimization using loss functions to enhance both components.

4.2 Hyperparameters

Learning Rate: Typically 0.0002.

Batch Size: Commonly 64.

Epochs: Around 5000 for convergence.

5. Benchmarking Criteria

5.1 Effectiveness Metrics

Wasserstein Distance: Measures data distribution closeness.

KS Statistic: Assesses cumulative distribution similarity.

F1 Score: Evaluates label accuracy for synthetic data.

5.2 Efficiency Metrics

Training Time: Time required for convergence.

Memory Usage: Tracks computational resource use.

5.3 Data Quality Metrics

Chi-Square Test: Evaluates categorical feature similarity.

PCA Visualization: Visualizes distribution overlap.

6. Benchmarking Results

Wasserstein Distance: 0.12 (close match to real data).

KS Statistic: 0.05 (minimal difference).

Training Time: 5 minutes per run.

PCA Visualization: High overlap between real and synthetic data.

7. Using the MEG Model

7.1 Setup

Install required libraries: numpy, pandas, torch, etc.

Preprocess data (categorical variables as integers).

7.2 Training

Use train(num_epochs, batch_size) to train the model:

python

Copy code

```
meg_adapter.train(num_epochs=5000, batch_size=64)
```

7.3 Generating Data

Generate synthetic data with:

python

Copy code

```
synthetic_data = meg_adapter.generate(num_samples=1000)
```

8. Future Improvements

Dynamic Masking: Adjust masks during training.

Hyperparameter Tuning: Optimize training parameters.

Regularization: Add dropout or L2 regularization to prevent overfitting.

Configuration

There is a file named katabatic_config.json that defines the various models and configurations by specifying the module names and class names katabatic can use. This file links model names to their corresponding implementation details, enabling the Katabatic system to dynamically load and operate various models. Each model is defined as a key-value pair, where the key represents the model name, and the value is a nested JSON object containing information about the module and class associated with that model.

Each model consists of tdgm_module_name: which indicates the Python module that contains the implementation of the model and tdgm_class_name which defines the class name within the module that provides the model's implementation.

“tdgm_module_name”: "katabatic.models.model_template.mock_adapter"
Indicates the location of the mock_adapter module within the katabatic.models.model_template package.

“tdgm_class_name”: "MockAdapter"
Specifies the MockAdapter class within the above module.

So for MEG the configuration should be

“tdgm_module_name”: “katabatic.models.meg.meg_adapter,

“tdgm_class_name”: “MegAdapter”

If katabatic needs to use the MEG model the system will read this JSON file and find tdgm_module_name and tdgm_class_name. The module is imported and the MegAdapter class is initialised. The model is then used within the application

Implementation

In MEG_implementation initially a dataset is loaded into a pandas DataFrame and any rows with missing values are removed from the dataset. Categorical columns are then converted to numerical values. Features X and labels Y are separated. The dataset is then split into training and testing sets using an 80/20 split seen in test_size=0.2. X_train and y_train is the training data used to train the model, X_test and y_test is the test data used to evaluate the models performance. Data is then standardised using StandardScaler. Data is converted int PyTorch Tensors. Dtype=torch.float32 ensures that the data type is compatible with PyTorch neural networks, unsqueeze(1) adds an additional dimension to make them column vectors which is necessary for PyTorch models. MaskedGenerator and Discriminator classes are defined just like in the adapter(see adapter section for details). The input_dim is the size of the random noise vector that serves as the input to the generators, while output_dim corresponds to the number of features in the real data. To divide the feature space among the generators, masks are defined for each generator. These masks specify which features a generator is responsible for producing. For example, a mask like [1, 0, 0] indicates that the generator will only produce the first feature, while the other features will be zeroed out. To ensure full feature coverage, an assert statement validates that the sum of the masks equals the total number of output dimensions. Multiple MaskedGenerator instances are initialized using these masks, and a single discriminator is created to evaluate the authenticity of the generated data. The loss function used is Binary Cross-Entropy Loss (BCELoss), this is a binary classification problem where the discriminator distinguishes between real and fake data. Separate optimizers are set up for the discriminator and each generator, allowing independent updates during training. The training

loop runs for a specified number of epochs (5000 in this case), with mini-batches of size 64. The model goes through a training loop with the discriminator set to training mode with no gradients. A batch of real data is sampled from the training set with labels set to 1 indicating real data. Next, random noise is fed into an ensemble of generators whose outputs are summed to produce fake data. The labels for this fake data are set to 0 indicating fake data. The discriminator evaluates both the real and fake data generating outputs used to compute two losses, one for how well it identifies real data (d_loss_real) and another for how well it detects fake data (d_loss_fake). These losses are summed to calculate the total discriminator loss (d_loss), This loss is then used to perform backpropagation, updating the discriminator's weights. The fake data is detached from the computation graph to ensure that gradients do not flow back into the generators during this process. Each generator is trained going through a loop. A batch of random noise is generated and passed through the generator to produce fake data. This fake data is then evaluated by the discriminator to determine how realistic it appears. The generator's loss is computed using Binary Cross-Entropy Loss, comparing the discriminator's output against a target of 1. The loss is backpropagated to update the generator's weights, improving its ability to produce realistic data. This process is repeated for all generators in the ensemble. Every 500 epochs, the discriminator loss (D Loss) and the last computed generator loss (G Loss) are logged to monitor training progress. The generate_synthetic_data function creates random noise vectors for the specified number of samples (num_samples) and passes them through each generator in the ensemble. Each generator produces synthetic outputs for its assigned subset of features, and these outputs are detached from the computation graph and converted into NumPy arrays. The synthetic outputs from all generators are then summed along the feature axis to produce the final combined synthetic dataset. This dataset is converted into a Pandas DataFrame with columns named according to the original feature names (X.columns) and saved as a CSV file named masked_ensemble_synthetic_data.csv

Benchmarking

The Benchmarking 1.py file provides a framework for generating synthetic data using an ensemble of GANs with masked generators and benchmarking the generated synthetic data against real data.

Data is prepared at the start where data is loaded from a csv file with missing values removed. Categorical data such as male and female and purchase decisions like yes and no and converted into numerical representations of 1 or 0. The dataset is then split into features and labels of x and y and followed by splitting the data into training and testing sets. The data is then standardised using the StandardScaler to eliminate mean and unit variance. This processed data is then converted into PyTorch tensors.

The framework consists of two neural network models: a Masked Generator and a Discriminator. The Masked Generator is designed to create synthetic data by concentrating on specific subsets of features determined by a mask. This approach ensures that each generator focuses on distinct feature sets, promoting diversity and structure in the generated data. Meanwhile, the Discriminator's role is to differentiate between real and synthetic data, providing a probability score through a sigmoid activation function. These models operate in an adversarial setup, with the Discriminator serving as a quality assurance mechanism to refine the synthetic data produced by the generator.

The GAN ensemble architecture comprises several masked generators and a single discriminator. Each generator takes a noise vector as input with its size defining the input dimensions and produces outputs corresponding to the number of features in the dataset. Unique masks are assigned to each generator to specialize in specific features, ensuring comprehensive representation across all features. The framework employs binary cross-entropy loss (BCELoss) to calculate the losses for both

the discriminator and the generators. Additionally, Adam optimizers are used for the discriminator and each generator to perform gradient descent and enhance model performance.

Training is conducted over multiple epochs, with each iteration involving distinct updates for the discriminator and the generators. The discriminator is trained using real data labeled as 1 and synthetic data labeled as 0, with its loss calculated to improve its ability to differentiate real data from synthetic. The generators, in turn, are trained based on feedback from the discriminator, striving to generate synthetic data that the discriminator perceives as real. Throughout training, the loss values for both the discriminator and the generators are recorded and visualized to track progress and performance.

Benchmarking assesses the quality of synthetic data by comparing it with real data. This process utilizes the discriminator to distinguish between real and synthetic data, measuring performance through standard classification metrics like accuracy, precision, recall, and F1 score.

Adapter

The code in the file MEG_Adapter.py defines the MEG_Adapter class, which inherits from base class MEGModelSPI. This class combines multiple generator models to create synthetic data and incorporates a discriminator in a setup similar to a GAN. The main goal is to train the generators to produce masked synthetic data and assess its quality using the discriminator.

The MEG_Adapter class is designed to facilitate the creation of synthetic data using a multi-model generative adversarial approach. The class uses generator model MaskedGenerator and a single discriminator to mimic real-world data distributions. It provides a framework for loading data, training adversarial models, and generating synthetic data, enabling users to utilise datasets. The class initialises components setting up input and output dimensions, a list to store generator models in self.models = [] and a list to store the corresponding optimisers in self.optimizers = []. The criterion is a loss function BCELoss(). This discriminator is trained to distinguish between real and synthetic data

The load_model method initialises a set of models and optimisers. The method loops through a list of self.masking_strategy, each item in this list is used to create a new instance of MaskGenerator. For each mask, a MaskedGenerator model is instantiated with self.input_dim, self.output_dim, and the mask as parameters. An Adam optimizer is created for each model. The optimizer's learning rate is set to 0.0002. Each model and its corresponding optimizer are stored in self.models and self.optimizers.

This load_data method prepares data for training and testing a machine learning model, particularly in PyTorch. X = data.drop('Purchase', axis=1) drops the target column Purchase from the dataset and y = data['Purchase'] selects the Purchase data column for prediction. The data is then split into a training set of 80% and a testing set of 20%. The features are normalised using StandardScaler. The scaler is fit to the training data and transformed and the fitted scaler is used to transform the test data. Training and testing data is then converted to PyTorch tensors. X_train_scaled and X_test_scaled are converted to torch.float32 tensors and y_train.values and y_test.values are converted to tensors with unsqueeze(1) to make it a single column dimension.

The train method implements a loop where multiple generator models work against a single discriminator. The inputs are the number of epochs for training and the number of samples for each training batch. The loop iterates through the number of epochs training each generator and the discriminator at every epoch. Real data is sampled from X_train_tensor using torch.randperm in

variable idx. Real labels are set to 1 for real data. Fake data is then prepared with noise being passed through each generator. Discriminator outputs for both real and fake data are calculated.

The generate method creates synthetic data using multiple models with an initial input of the number of synthetic samples to generate. Self.models is looped through generating synthetic parts. Random noise is generated as input for each generator and each generator processes it to produce a synthetic part, output is detached and the result is converted to a NumPy array. Synthetic parts by all models are combined in combined_data. This is then converted into a Pandas DataFrame.

The Discriminator Class is a neural network taking in the nn.Module from PyTorch. It is a classifier to distinguish between real and fake data. The initialisation method takes in the number of features in each data sample as input_dim. There is an input layer mapping the input to 128 neurons. For non linearity nn.ReLu() is used. Nn.Linear(128, 1) is a fully connected layer mapping 128 neurons to a single output. Nn.Sigmoid() converts the output into a range between 0 and 1 for binary classification.

The MaskedGenerator class plays a key role by applying masks to limit output generation to specific dimensions. This approach ensures that each generator contributes only to a specific portion of the synthetic dataset, providing modular control. The generator consists of a two-layer network with ReLU activation and a Tanh output layer. These activation functions are commonly used in GAN architectures, with Tanh producing normalized outputs between -1 and 1, which can be scaled or processed further as needed. The masking strategy not only adds structure to the model but also enhances interpretability by constraining each generator to particular data attributes

The MEG_SPI.py file defines two abstract base classes (ABCs) using Python's abc module

MEGModelSPI and MEGMetricSPI. These classes serve as templates for specific implementations of synthetic data generation models and evaluation metrics in the MEG framework. The __init__ method is defined as an abstract method using the @abstractmethod decorator, which makes it mandatory for any subclass of MEGModelSPI to implement its own version of this method. The rest are abstract methods serving as templates for implementing specific functionalities related to model operations and data evaluation.

Use Cases

pip install -U scikit-learn

pip install pyitlib

pip install tensorflow

pip install pgmpy

pip install sdv

installs all the necessary packages

from katabatic.katabatic import Katabatic

import pandas as pd

import numpy as np

imports the katabatic framework, pandas and numpy

```
from katabatic.models.meg_DGEK.utils import get_demo_data

df = get_demo_data('adult-raw')

df.head()
```

this demonstrates how to load and preview a demo dataset

```
from katabatic.models.meg_DGEK.utils import get_demo_data

df = get_demo_data('adult-raw')

df.head()
```

```
from sklearn.model_selection import train_test_split

x, y = df.values[:,:-1], df.values[:,-1]

X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.5)
```

This code loads a dataset named 'adult-raw' using the get_demo_data function, storing it in a DataFrame (df), and displays its first few rows using .head(). It then splits the DataFrame into features (x) and target labels (y), where x contains all columns except the last, and y contains only the last column. The train_test_split function is used to split the data into training and testing sets, allocating 50% of the data to each, resulting in X_train, X_test, y_train, and y_test. This prepares the data for machine learning workflows.

```
from katabatic.models.meg_DGEK.meg_adapter import MegAdapter

adapter = MegAdapter()

adapter.load_model()

adapter.fit(X_train, y_train, epochs=5)
```

This code initializes an instance of the MegAdapter class. The load_model() method is called to load or initialize the model, and then the fit() method is used to train the model on the training data (X_train, y_train) for 5 epochs. This process sets up and trains the model using the provided data.

```
adapter.generate(size=5)
```

This code calls the generate method on the adapter object, instructing the model to generate 5 samples of synthetic data

```
import katabatic as kb

from katabatic.models import meg

from katabatic.models import ganblr

from katabatic.evaluate import eval_method1
```

from katabatic.utils.preprocessing import data_processing_method1

This code imports various modules and functions from the katabatic library. It imports the main katabatic module as kb, specific model-related submodules (meg and ganblr) for working with different machine learning or synthetic data models, an evaluation method (eval_method1) for assessing model performance, and a preprocessing utility (data_processing_method1) for preparing data before training or evaluation. These imports set up the environment for tasks like model training, evaluation, and data preparation within the katabatic framework.

User Interface

There is a file called katabatic_logic.py. It starts by loading a demo dataset (adult-raw) using get_demo_data, returning a DataFrame where the last column is assumed to be the target variable. The data is split into features (x) and target (y), and a training subset is created using an 80-20 split (default test size of 50%). The MegAdapter model is loaded and trained on the training data for a specified number of epochs (default: 5). Finally, the trained model generates synthetic data of a specified size (default: 5 rows

App.py is the main file where a flask web application is set up that integrates Katabatic's logic to generate synthetic data based on user input. The application includes two primary routes: the root route (/) renders the index.html template as the main interface, while the /process route handles POST requests to process user inputs. Users can optionally specify the size of the synthetic dataset they want to generate, with a default size of 5. Upon receiving the request, the load_demo_data function loads a demo dataset, and train_model trains the model using the MEG adapter. The generate_data function then produces synthetic data based on the trained model, which is converted into JSON format for easy rendering on the UI. Finally, the JSON response containing the synthetic data is sent back to the front end, where it can be displayed to the user. The application runs locally in debug mode, enabling easy testing and development. The app is run using the command python app.py

The script in index.html is written in javascript and it adds interactivity to a web page by dynamically submitting a form, retrieving data, and rendering it as an HTML table. When a user submits the form (#data-form), the submit event is intercepted using preventDefault() to prevent the default page reload behavior. The form data, specifically the dataset size entered by the user (#size), is sent to the server's /process route using an asynchronous POST request with fetch(). The server responds with a JSON object containing synthetic data, which is parsed and used to generate a dynamic HTML table. First, the table headers are generated based on the keys of the first object in the returned data. Then, each row of the table is populated using the values from the JSON response. The resulting table replaces any previous content within the target element (#synthetic-table), effectively displaying the synthetic data in a clean, tabular format on the webpage. The UI will display a “Generating Data…” message when the data is being generated as this takes several minutes, the dots are animated to show the user that something is happening on the backend. The app is run by cd into Katabatic_UI and running the command python app.py

9. Conclusion

The MEG algorithm's masking and ensemble generator architecture make it a robust solution for generating synthetic tabular data. It balances accuracy, scalability, and feature-specific generation, paving the way for its application in diverse data domains.