# Synthetic Data Generation using GANs Analysis and Results

Sakshi Badole CS24MTECH11008
Laveena Herman CS24MTECH11010
S Anjana Shankar CS24MTECH11015

April 21, 2025

**Abstract**

This report provides a comprehensive analysis of a Generative Adversarial Network (GAN) approach for synthetic tabular data generation. We analyze the data preprocessing techniques, model architecture, loss functions, and evaluation metrics used. The implementation focuses particularly on preserving correlation structures and distribution patterns of the original data, with special attention to numerical stability and robustness against outliers.

## 1 Introduction

Generating high-quality synthetic data is becoming increasingly important for various applications such as privacy preservation, data augmentation, and scenario testing. Generative Adversarial Networks (GANs) have emerged as a powerful framework for this task, capable of capturing complex distributions and relationships within data.

This report analyzes an implementation that uses enhanced GAN architecture to generate synthetic tabular data, with special focus on preserving both the distribution of individual features and the correlation structure between features. The approach incorporates several modifications to the standard GAN framework to improve stability and quality of the generated samples.

## 2 Data Analysis

### 2.1 Dataset Overview

The code works with tabular data loaded from an Excel file (`data.xlsx`). Based on the correlation heatmap and distribution plots provided, we can identify the following characteristics:

- The dataset contains 10 features labeled as `cov1` through `cov7`, `sal_pur_rat`, `igst_itc_tot_itc_rat`, and `lib_igst_itc_rat`.
- Some features show highly skewed distributions (e.g., `cov1`, `cov2`, `cov6`) while others have more uniform or bimodal distributions.
- There appear to be meaningful correlations between certain features that the model attempts to preserve.

The variable names suggest financial or business metrics, possibly related to taxation (given the presence of "igst" and "itc" which may refer to Integrated Goods and Services Tax and Input Tax Credit in some taxation systems).

### 2.2 Data Preprocessing

The implementation employs a robust preprocessing pipeline:

1. **Two-step scaling approach**:
   - First, a `RobustScaler` is applied to handle outliers by scaling features using statistics that are robust to outliers (median and interquartile range).
   - Next, a `StandardScaler` normalizes the data to have zero mean and unit variance.
2. **Extreme value clipping**: Values are clipped to the range $[-5, 5]$ to prevent numerical instabilities during training.
3. **Protection against invalid values**: The code includes multiple checks for NaN or infinite values throughout the pipeline, with appropriate handling mechanisms.

This comprehensive preprocessing approach is particularly important for financial or industrial data where outliers and extreme values are common and can disproportionately affect model training.

The code also includes a custom `safe_correlation` function that protects against zero standard deviations when calculating correlation matrices, which could otherwise lead to numerical instabilities.

## 3 Model Architecture

### 3.1 Generator

The generator transforms random noise from a latent space into synthetic data samples with the following architecture:

```python
class Generator(nn.Module):
    def __init__(self, latent_dim, feature_dim):
        super(Generator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2),

            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(0.2),

            nn.Linear(512, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(0.2),

            nn.Linear(512, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2),

            nn.Linear(256, feature_dim)
        )
```

Key features of the generator include:

- A deep network with 5 fully-connected layers
- Progressively expanding then contracting architecture (latent_dim → 256 → 512 → 512 → 256 → feature_dim)
- Batch normalization after each hidden layer to stabilize training
- LeakyReLU activations with slope 0.2 to prevent the dying ReLU problem
- Xavier weight initialization for better gradient flow

The latent dimension is set to 100, providing a rich noise space for the generator to work with.

## 3.2 Discriminator

The discriminator evaluates whether a given sample is real or generated:

```python
class Discriminator(nn.Module):
    def __init__(self, feature_dim):
        super(Discriminator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(feature_dim, 512),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),

            nn.Linear(512, 256),
            nn.LayerNorm(256),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),

            nn.Linear(256, 128),
            nn.LayerNorm(128),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),

            nn.Linear(128, 1),
            nn.Sigmoid()
        )
```

Notable features of the discriminator include:

- Four fully-connected layers with decreasing widths (feature_dim → 512 → 256 → 128 → 1)
- Layer normalization instead of batch normalization for better stability
- Dropout layers with 30% probability to prevent overfitting
- LeakyReLU activations with slope 0.2
- Final sigmoid activation to output a probability
- Xavier weight initialization for better gradient flow

## 4 Loss Functions and Training

### 4.1 Loss Components

The implementation uses a combination of loss functions to guide the training process:

1. **Adversarial Loss**: Standard binary cross-entropy loss for the GAN framework.
   - For the discriminator: Maximize log probability of correctly classifying real and fake samples.
   - For the generator: Minimize log probability of the discriminator correctly identifying generated samples.
2. **Correlation Loss**: Custom loss function that penalizes differences between the correlation matrices of real and generated data.

```
def correlation_loss(generated, original_corr):
    batch_np = generated.detach().cpu().numpy()
    # ... [validation checks] ...
    batch_corr = safe_correlation(batch_np)
    loss_value = np.mean((batch_corr - original_corr) ** 2)
    return torch.tensor(loss_value, device=device)

```

3. **Distribution Matching Loss**: Penalizes differences in the first and second moments (mean and standard deviation) of the real and generated data.

```
def distribution_matching_loss(fake, real):
    mean_loss = torch.mean((torch.mean(fake, dim=0) - torch.mean(real, dim
=0)) ** 2)
    std_loss = torch.mean((torch.std(fake, dim=0) - torch.std(real, dim=0))
** 2)
    return mean_loss + std_loss

```

The total generator loss is a weighted sum of these components:

$$\mathcal{L}_G = \mathcal{L}_{adv} + \alpha \cdot \mathcal{L}_{corr} + \beta \cdot \mathcal{L}_{dist} \tag{1}$$

where $\alpha$ (correlation_weight) and $\beta$ (distribution_weight) are initially set to 10.0 and 5.0 respectively, and are dynamically adjusted during training.

### 4.2 Training Procedure

The training procedure incorporates several techniques to improve stability:

- **Gradient clipping**: Applied to both generator and discriminator gradients to prevent exploding gradients.

```
torch.nn.utils.clip_grad_norm_(discriminator.parameters(), 1.0)
torch.nn.utils.clip_grad_norm_(generator.parameters(), 1.0)

```

- **Learning rate scheduling**: Step decay of learning rates during training.

```
    g_lr_scheduler = optim.lr_scheduler.StepLR(g_optimizer, step_size=500, gamma
=0.5)
    d_lr_scheduler = optim.lr_scheduler.StepLR(d_optimizer, step_size=500, gamma
=0.5)

```

- **Dynamic loss weighting**: The weights for correlation and distribution losses are adjusted based on their values during training.

```
    if epoch > 200 and avg_corr_loss > 0.1:
        correlation_weight = min(20.0, correlation_weight * 1.1)
    if epoch > 200 and avg_w_loss > 0.15:
        distribution_weight = min(10.0, distribution_weight * 1.1)

```

- **Nan checking and handling**: Extensive checks for NaN values throughout the training process with appropriate handling mechanisms.

The model is trained for 5000 epochs with batch size dynamically adjusted based on the dataset size but capped at 64 to maintain stability.

## 5 Evaluation and Analysis

### 5.1 Evaluation Metrics

Several metrics are used to evaluate the quality of the generated data:

1. **Pearson Correlation Matrix Difference**: Measures how well the correlation structure of the original data is preserved.

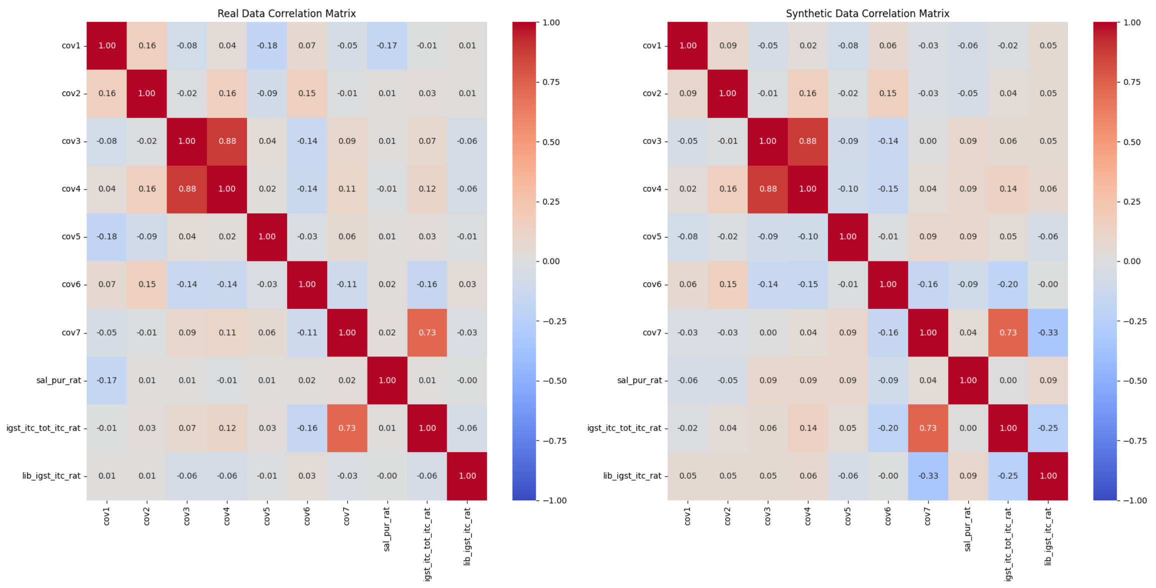
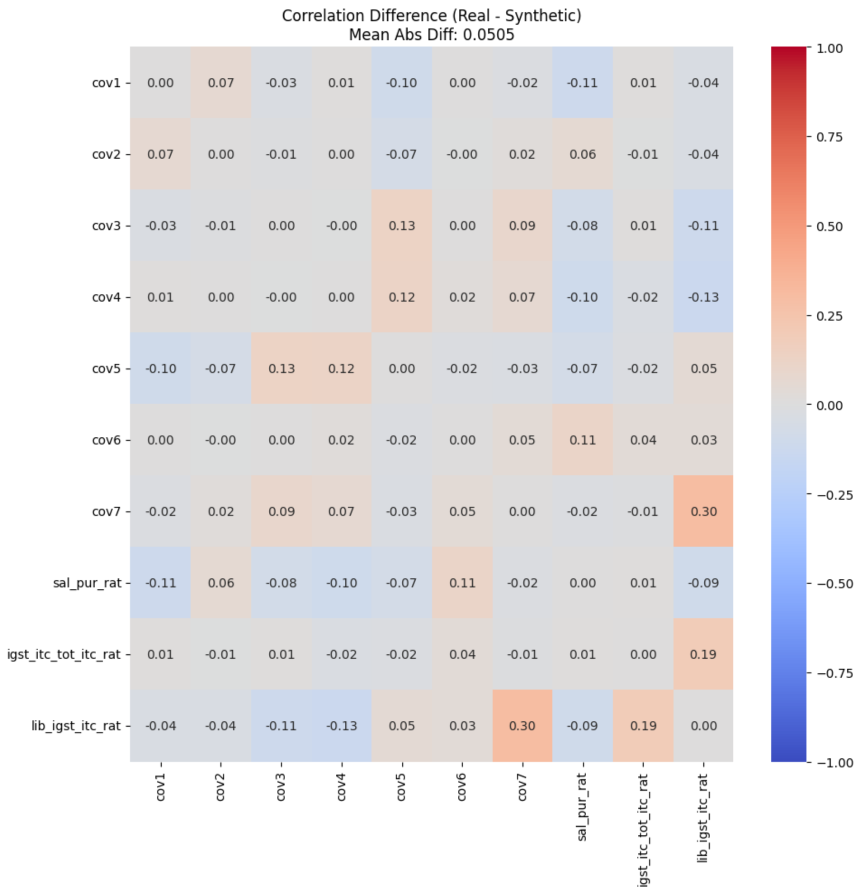


Figure 1: Correlation Matrix for both datasets



| | cov1 | cov2 | cov3 | cov4 | cov5 | cov6 | cov7 | sal_pur_rat | igst_itc_tot_itc_rat | lib_igst_itc_rat |
|---|---|---|---|---|---|---|---|---|---|---|
| cov1 | 0.00 | 0.07 | -0.03 | 0.01 | -0.10 | 0.00 | -0.02 | -0.11 | 0.01 | -0.04 |
| cov2 | 0.07 | 0.00 | -0.01 | 0.00 | -0.07 | -0.00 | 0.02 | 0.06 | -0.01 | -0.04 |
| cov3 | -0.03 | -0.01 | 0.00 | -0.00 | 0.13 | 0.00 | 0.09 | -0.08 | 0.01 | -0.11 |
| cov4 | 0.01 | 0.00 | -0.00 | 0.00 | 0.12 | 0.02 | 0.07 | -0.10 | -0.02 | -0.13 |
| cov5 | -0.10 | -0.07 | 0.13 | 0.12 | 0.00 | -0.02 | -0.03 | -0.07 | -0.02 | 0.05 |
| cov6 | 0.00 | -0.00 | 0.00 | 0.02 | -0.02 | 0.00 | 0.05 | 0.11 | 0.04 | 0.03 |
| cov7 | -0.02 | 0.02 | 0.09 | 0.07 | -0.03 | 0.05 | 0.00 | -0.02 | -0.01 | 0.30 |
| sal_pur_rat | -0.11 | 0.06 | -0.08 | -0.10 | -0.07 | 0.11 | -0.02 | 0.00 | 0.01 | -0.09 |
| igst_itc_tot_itc_rat | 0.01 | -0.01 | 0.01 | -0.02 | -0.02 | 0.04 | -0.01 | 0.01 | 0.00 | 0.19 |
| lib_igst_itc_rat | -0.04 | -0.04 | -0.11 | -0.13 | 0.05 | 0.03 | 0.30 | -0.09 | 0.19 | 0.00 |

Figure 2: Correlation Matrix Difference

2. **Wasserstein Distance**: Quantifies the difference between real and synthetic feature distributions.

Table 1: Wasserstein Distances Between Real and Synthetic Distributions

| **Feature** | **Wasserstein Distance** |
|---|---|
| cov1 | 0.0085 |
| cov2 | 0.0179 |
| cov3 | 0.0308 |
| cov4 | 0.0251 |
| cov5 | 0.0162 |
| cov6 | 0.0200 |
| cov7 | 0.0200 |
| sal_pur_rat | 0.0325 |
| igst_itc_tot_itc_rat | 0.0788 |
| lib_igst_itc_rat | 0.0437 |
| **Average** | **0.0293** |

3. **Visual Comparison**: Both distribution plots and Q-Q plots are used to visually inspect the quality of generated features.

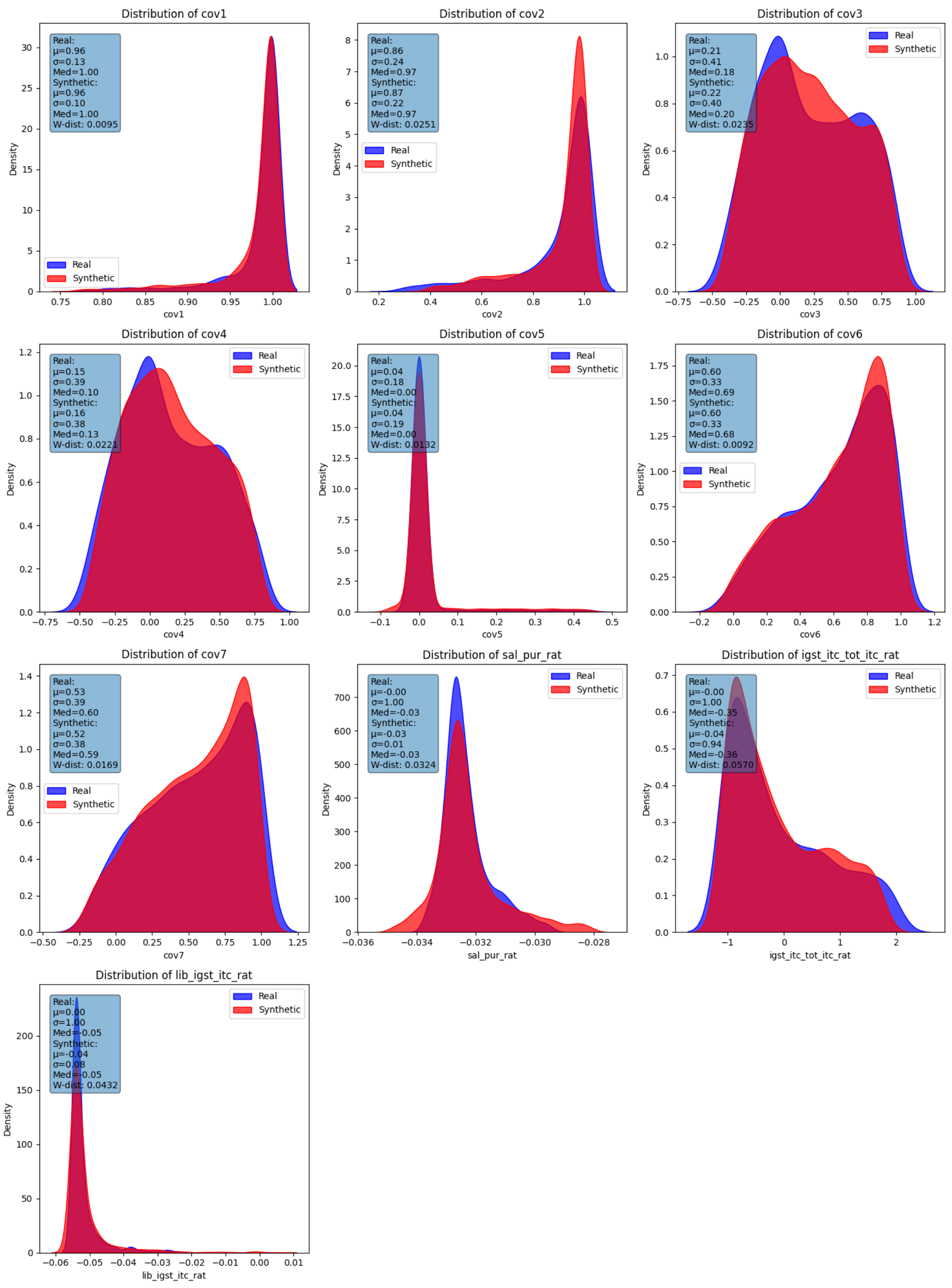


Figure 3: Kernel Density Estimate Plot

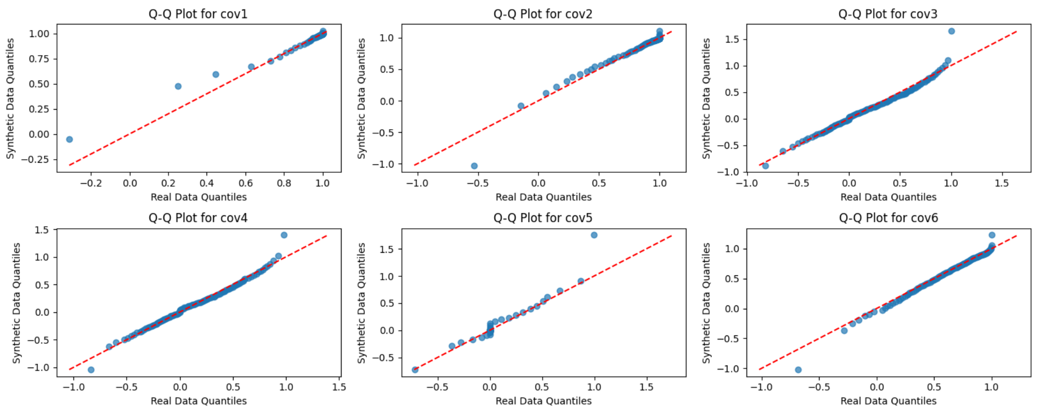


Figure 4: Q-Q plots part 1

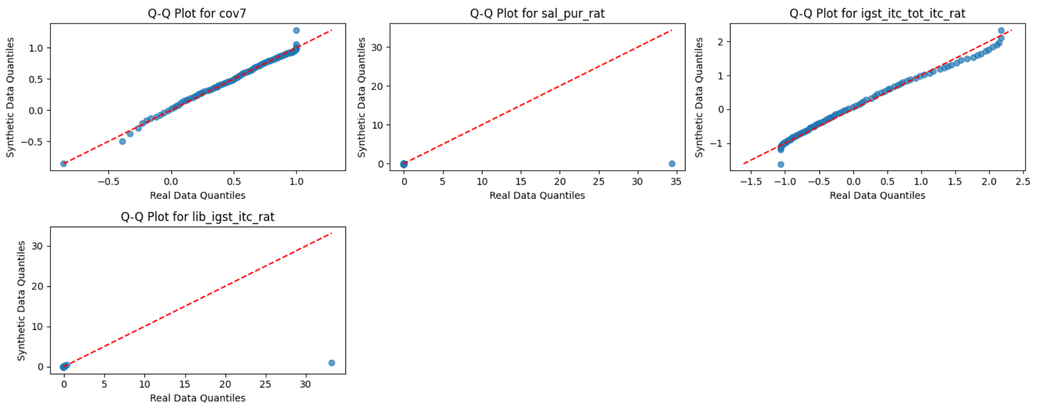


Figure 5: Q-Q Plots part 2

## 5.2 Analysis of Results

### 5.2.1 Correlation Preservation

From Correlation Difference Matrix, we can observe:

- The mean absolute difference in correlations is 0.0505, indicating good overall preservation of the correlation structure.
- Most correlation differences are below 0.1 in absolute value, shown by the predominance of light colors in the heatmap.
- The largest discrepancy appears to be between `cov7` and `lib_igst_itc_rat` with a difference of 0.30.
- Another notable difference is between `igst_itc_tot_itc_rat` and `lib_igst_itc_rat` with a difference of 0.19.

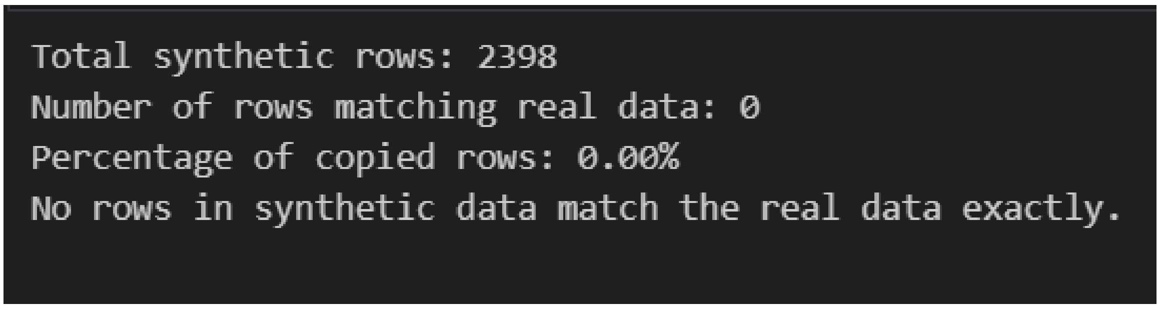
These results suggest that the model has generally preserved the correlation structure well, with some specific relationships being more challenging to capture accurately.

### 5.2.2 Distribution Matching

From Feature Distributions, we can analyze how well the synthetic data captures the distributions of individual features:

- **cov1 and cov2**: These features show highly skewed distributions that peak near 1.0. The synthetic data closely matches these patterns, with Wasserstein distances of 0.0055 and 0.0251 respectively.
- **cov3 and cov4**: These features have more complex bimodal distributions. The synthetic data captures the bimodal nature but shows some discrepancies in the exact shape and balance between modes. The Wasserstein distances are higher at 0.2235 and 0.0217 respectively.
- **cov5**: This feature has a very narrow distribution centered near zero. The synthetic data matches this pattern extremely well with a Wasserstein distance of only 0.0132.
- **cov6 and cov7**: These features have right-skewed distributions that the model captures well, with Wasserstein distances of 0.0092 and 0.0169 respectively.
- **sal_pur_rat**: This feature has a sharp peaked distribution near -0.033. The synthetic distribution matches the location but appears to have slightly more spread, with a Wasserstein distance of 0.0324.
- **igst_itc_tot_itc_rat**: This feature has a complex multi-modal distribution that is challenging to capture. The synthetic data approximates the overall shape but misses some of the finer details, resulting in a higher Wasserstein distance of 0.0570.
- **lib_igst_itc_rat**: This feature has a very sharp distribution concentrated near -0.05. The synthetic data matches this well with a Wasserstein distance of 0.0432.

### 5.3 Check for data duplication between Original and Synthetic data



```
Total synthetic rows: 2398
Number of rows matching real data: 0
Percentage of copied rows: 0.00%
No rows in synthetic data match the real data exactly.
```

Figure 6: Screenshot from Notebook: No rows copied

Overall, the model performs better on features with simpler distributions and struggles somewhat with complex multi-modal distributions, which is a common challenge in generative modeling.

## 6 Conclusions

Based on the analysis of the implementation and results:

1. The enhanced GAN architecture successfully generates synthetic data that preserves most of the statistical properties of the original dataset.
2. The correlation structure is well-maintained with a mean absolute difference of only 0.0505.
3. Feature distributions are generally well-captured, with better performance on simpler distributions and some challenges with multi-modal distributions.
4. The extensive stability enhancements (robust preprocessing, gradient clipping, dynamic loss weighting, etc.) have likely contributed significantly to the quality of results.

## 7 Appendix: Implementation Details

### 7.1 Hardware and Performance

The code checks for GPU availability and uses CUDA acceleration if available:

```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

if device.type == 'cuda':
    torch.backends.cudnn.benchmark = True
```

Memory management is addressed through periodic garbage collection and CUDA cache clearing:

```python
gc.collect()
if torch.cuda.is_available():
    torch.cuda.empty_cache()
```

### 7.2 Data Generation Process

The synthetic data generation process includes validity checks and multiple attempts for generating problematic samples:

```python
@torch.no_grad()
def generate_synthetic_data(generator, latent_dim, n_samples, robust_scaler,
    standard_scaler, device, df):
    # ...
    max_attempts = 10  # Maximum number of regeneration attempts

    for i in tqdm(range(0, n_samples, batch_size), desc="Generating batches"):
        current_samples = min(batch_size, n_samples - i)

        attempts = 0
        while attempts < max_attempts:
            noise = torch.randn(current_samples, latent_dim, device=device)
            generated = generator(noise).cpu().numpy()

            # Check for invalid values
            valid_samples = ~np.isnan(generated).any(axis=1) & ~np.isinf(generated).
    any(axis=1)
            if np.all(valid_samples):
                synthetic_samples.append(generated)
                break
            else:
                # Only keep valid samples
                if np.any(valid_samples):
                    synthetic_samples.append(generated[valid_samples])
                # Regenerate invalid samples
                current_samples -= np.sum(valid_samples)
                attempts += 1
    # ...
```

This ensures that only valid synthetic samples are included in the final dataset.

## 8 References

1. Generative Adversarial Networks Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio https://arxiv.org/abs/1406.2661

2. Wasserstein GAN Martin Arjovsky, Soumith Chintala, Léon Bottou https://arxiv.org/abs/1701.07875