# AI in Industry: Synthetic Data Generation

Giovanni Grotto, Tancredi Bosi, Francesco Farneti

April 5, 2025

## Contents

# 1 Data Cleaning

The initial dataset presented a lot of difficulties and, thus, the data cleaning step was crucial in our pipeline. The cleaning is performed in three steps: Data Organization, Tag Clustering and Invalid Rows Removal.

## 1.1 Data Organization

The `organize_data` function is designed to clean and structure a dataset, ensuring it is ready for analysis:

- **Strip Whitespace from Column Names and String Values:** the function removes leading and trailing whitespace from column names to avoid inconsistencies in data access. Similarly, it strips spaces from all string values within the dataset to standardize textual entries.
- **Remove Duplicate Rows:** duplicate rows are dropped to prevent redundant information, ensuring each record in the dataset is unique.
- **Clean the 'Overall' Column:** the function trims any leading tilde (`~`) characters in the 'Overall' column, which might indicate formatting artifacts or parsing issues.
- **Split the 'Residence' Column:** the 'Residence' column, which contains city, province, and region information, is split into three separate columns: 'City', 'Province', and 'Region'. The split is performed using the delimiters `»` and `~`, which are assumed to separate the components of the residence string.
- **Convert Year Columns to Integers:** the 'Year of insertion' and 'Year of Recruitment' columns are converted to integers for proper date-based calculations. Any non-numeric values or formatting characters (like brackets) are handled, and invalid entries are coerced to `NaN` before casting to integers.
- **Filter Out Invalid 'Last Role' Values:** rows with invalid entries in the 'Last Role' column (like `????`, `-`, `.`, or `/`) are set to `NaN` to eliminate unreliable data points.
- **Group Rows by ID and Keep the Most Complete Record:** when multiple rows share the same ID, the function retains the row with the highest count of non-null (non-`NaN`) values. This approach ensures the most complete and informative entry is preserved for each unique ID.
- **Drop Unnecessary Columns:** finally, columns deemed irrelevant for analysis — such as `linked_search__key`, `Years Experience.1`, `Study Area.1`, `Residence`, and `Recruitment Request` — are dropped to streamline the dataset.

## 1.2 Tag Clustering

The `cluster_tag` function serves as the primary tool for clustering and mapping values in the 'Last Role' and 'TAG' columns of the dataset. This process involves several steps, including text preprocessing, hierarchical clustering, and the assignment of cluster names based on common words. Below is a detailed explanation of how this process works.

The first step in the process is handled by the `preprocess_text` function. This function prepares the text for clustering by removing special characters and converting it to lowercase.

Next, the `cluster_and_map_roles` function performs the actual clustering. It takes a list of unique values, such as job roles or tags, and preprocesses them using the `preprocess_text` function. A TF-IDF matrix is then created using `TfidfVectorizer`, which transforms the text data into a numerical format suitable for clustering. Hierarchical clustering is applied using `AgglomerativeClustering`, with a distance threshold of 1.5 and 'ward' linkage to determine the optimal number of clusters. Once the clustering is complete, the function maps each cluster ID to its corresponding values and assigns a cluster name based on the most common words within that cluster.

To map individual values to their respective cluster names, the `map_to_cluster_name` function is used. This function takes a value, a list of unique values, the cluster assignments, and the cluster names as inputs. If the value is found in the list of unique values, it retrieves the corresponding cluster ID and returns the associated cluster name, otherwise, it returns the original value.

Finally, the `cluster_tag` function orchestrates the entire process: it extracts unique values from the 'Last Role' and 'TAG' columns of the dataset and applies the `cluster_and_map_roles` function to perform clustering and assign cluster names. The original values in the 'Last Role' and 'TAG' columns are then mapped to their respective cluster names using the `map_to_cluster_name` function. The modified dataset, with clustered values, is returned as the final output.

### 1.3 Invalid Rows Removal and Validity Check

The `filter_minor_workers` function is designed to remove rows from the dataset that contain inconsistencies between the 'Age Range' and 'Years Experience' columns. Specifically, it filters out workers whose age and experience combinations are logically invalid. For example, a worker with an age range of "$<$ 20 years" should not have years of experience such as "[+10]", "[7-10]", "[5-7]", or "[3-5]", as these values imply a level of experience that is inconsistent with their age.

Once the dataset had been cleaned and filtered, including the removal of invalid rows, we checked that all remaining entries were valid. This step was essential because adding constraints to the synthesizer (see Section 3) required the input dataset to fully comply with those constraints. To address this, we developed a dedicated function called `check_constraint`, designed to verify data validity at every stage of the process. This function helped ensure that the dataset consistently met the necessary requirements, preventing potential errors and maintaining the overall quality of the data pipeline.

## 2 Synthesizer

The synthesizer used is the Gaussian Copula, which is one of the models available within the Synthetic Data Vault (SDV) library.

### 2.1 How Synthesizer Works

The operation of the model is built on the definition of a copula, a statistical function that describes the dependence structure between multiple variables, allowing each to re-

tain its own marginal distribution. The synthesizer in question is more specifically based on a Gaussian Copula, meaning that the data is assumed to follow a multivariate normal distribution, regardless of the original distributions of the individual variables. The Gaussian Copula process involves transforming each column's value series into a standard normal distribution and estimating the dependence structure between these variables using a covariance matrix.

The synthesizer follows a structured process, divided into four main phases:

### 2.1.1 Estimating Marginal Distributions

Each column of the dataset is modeled separately to determine its distribution. This can be assigned manually or determined by default by the synthesizer using Kolmogorov-Smirnov tests, which compare the empirical distribution with possible theoretical distributions. The possible distributions are: 'norm', 'beta', 'truncnorm', 'uniform', 'gamma', and 'gaussian kde'.

### 2.1.2 Transforming Data into Gaussian Space

Once the marginal distributions are estimated, the values of the variables are transformed so that they follow a standard normal distribution. **Transformation Process:** For a value $x_i$ in column $X$:

- The cumulative distribution function (CDF) $F_x(x_i)$ is calculated based on the estimated distribution.
- The value is transformed into Gaussian space by applying the inverse cumulative distribution function of the standard normal distribution (probit function):

$$y_i = \Phi^{-1}(F_x(x_i))$$

  where $\Phi^{-1}$ is the inverse of the standard normal CDF.

After this transformation, all variables follow a multivariate normal distribution.

### 2.1.3 Estimating the Covariance Structure

After the transformation, the dependence between variables is modeled through a covariance matrix. Assuming we are dealing with two columns, $X$ and $Y$, the covariance between them is given by:

$$\text{Cov}(X, Y) = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \bar{X})(Y_i - \bar{Y})$$

where $n$ represents the total number of samples considered, and $\bar{X}$ and $\bar{Y}$ are the means of $X$ and $Y$, respectively.

### 2.1.4 Generating Synthetic Data and Inverse Transformation

The generation of synthetic samples follows these steps:

- New values are drawn from a multivariate normal distribution based on the covariance matrix.
- The inverse transformation is applied to return the values to the original domain:

This ensures that the generated data follow the same marginal distributions as the original dataset.

## 2.2 Why This Synthesizer?

This synthesizer has proven to be excellent as it can maintain the dependence relationships between variables, and in terms of computational efficiency, it has proven to be less resource-intensive compared to other generative models.
We also tested two other synthesizers available in SVD, namely *CTGAN* and *TVAE*, but we encountered some significant issues compared to *GaussianCopulaSynthesizer*.

### 2.2.1 CTGAN

It proved to be computationally too heavy, even with a small number of epochs. The training time was significantly longer than the other two methods, making it impractical in this context. Although its GAN-based approach is theoretically more powerful, the high computational cost was not justified by a significant improvement in performance compared to the other models.

### 2.2.2 TVAE

*TVAE*, on the other hand, did not have speed issues, but even after optimizing the number of training epochs, its performance remained consistently lower than that of *GaussianCopula*. This could be due to several factors, such as an insufficient amount of data to effectively model the dependencies between variables. Since *TVAE* attempts to reconstruct these dependencies through the latent space, there is a risk of information loss or a latent representation that fails to fully capture the structure of the original dataset.

Given the available data, the most effective method turned out to be *GaussianCopulaSynthesizer*, thanks to its ability to directly and accurately model the data distribution without the computational or representational limitations of the other two approaches.

## 2.3 Comparison of Synthesizers and Evaluation Metrics

Below, we present a comparison of the results obtained by generating 10,000 samples using Gaussian Copula and TVAE. For TVAE, training was conducted for 200 epochs, as this number provided the best results according to the evaluation metrics used. We excluded GAN because its training required excessively long times without achieving performance comparable to the other two models.

| Synthesizer | Training Time (s) | Data Validity | Data Structure | Column Shapes | Column Pair Trends |
|---|---|---|---|---|---|
| GaussianCopula | 11.63 | 0.9740 | 1.0000 | 0.9459 | 0.8908 |
| TVAE | 97.46 | 0.9740 | 1.0000 | 0.7668 | 0.8092 |

Table 1: Comparison of Different Synthesizers

The following is a detailed explanation of the evaluation metrics used to analyze the generated synthetic data.

### 2.3.1 Data Validity

The data validity metric examines the conformity of synthetic data to fundamental properties expected from real data. Several aspects are verified using these validity metrics:

- **Uniqueness of Primary Keys**: Each record should have a unique primary key (such as an ID) without duplicates. This property is essential to ensure data integrity.
- **Adherence to Categories**: Columns containing discrete values (e.g., categories like "Room Type" or "Gender") in synthetic data should match the categories present in the real data.
- **Adherence to Numerical Boundaries**: Continuous values in synthetic data should stay within the minimum and maximum ranges observed in the real data, avoiding any out-of-bounds values.

### 2.3.2 Data Structure

This metric focuses on the similarity of the data structure. Synthetic data should follow the same structure as real data, meaning it should have the same columns and data types. Additionally, column names should be consistent between real and synthetic data. If, for example, an important column is missing from the synthetic data or if column names do not match, this suggests an issue in data generation. The main structure metric is **Column Names**: it verifies that the column names are the same between real and synthetic data.

### 2.3.3 Data Quality

Data quality metrics examine the statistical similarity between real and synthetic data, assessing how well synthetic data can reproduce the statistical characteristics of real data. These metrics focus on two main aspects:

- **Column Shapes**: Analyzes the similarity in the marginal distributions of individual columns. In other words, it compares the distribution of values in each column between real and synthetic data. A common example is to compare whether a column that follows a normal distribution in real data also follows a similar distribution in synthetic data. The metric used for this analysis is **KSComplement (Kolmogorov-Smirnov Complement)**, it measures the distance between the cumulative distributions of real and synthetic data. A score close to 1 indicates that the distributions are very similar, while a score close to 0 indicates a significant difference.
- **Column Pair Trends**: Examines the statistical relationships between pairs of columns. For example, it checks the correlation between "Price" and "Room Type" in both real and synthetic data. If the correlation between these two columns is similar in synthetic data as in real data, the metric will score high. The metric used for this analysis is **Column Correlation**, it measures the statistical relationship (e.g., Pearson correlation) between pairs of columns in real and synthetic data.

# 3 Applying Constraints

In this project, constraints were used to guide the model during training to ensure that certain conditions or properties were respected. These constraints helped to improve the stability of the model and the interpretability of the results.

## 3.1 How Constraints Were Applied

The constraints were applied by defining custom rules that the synthetic data must follow. These rules were implemented using the `sdv.constraints` module, which allows for the creation of both built-in and custom constraints. The constraints were added to the synthesizer using the `add_constraints` method, which ensures that the generated synthetic data adheres to the specified rules.

In the `main.py` file, the function `set_constraints` is responsible for loading and applying these constraints. The constraints include both fixed combinations of columns and custom constraints. For example, the custom constraint `CustomYearsHired` ensures that the relationship between the columns `Candidate State`, `Year of insertion`, and `Year of Recruitment` is valid. This constraint checks two conditions:

- If `Candidate State` is `Hired`, then `Year of insertion` must be less than or equal to `Year of Recruitment`.
- If `Candidate State` is not `Hired`, then `Year of Recruitment` must be `NaN`.

This custom constraint is defined in the `custom_constraint_years.py` file, where the function `is_valid_YearsHired` checks the validity of the data based on the specified conditions.

Additionally, several constraints were applied using the `FixedCombinations` class, which ensures that certain combinations of column values remain consistent in the synthetic data:

- The combination of `Candidate State` and `Year of Recruitment` must remain fixed, ensuring that the recruitment year is consistent with the candidate's state.
- The combination of `Age Range` and `Years Experience` must remain fixed, ensuring that the years of experience are logically consistent with the age range.
- The combination of `City`, `Province`, and `Region` must remain fixed, ensuring that geographical data is consistent.
- The combination of `event_type__val` and `event_feedback` must remain fixed, ensuring that event types and their corresponding feedback are consistent.

## 3.2 Why a Custom Constraint Was Needed

We needed a custom constraint because the standard constraints provided by the SDV library did not support the implementation of a certain level of logical complexity. Specifically, we defined a custom constraint to manage the relationship between the columns `Candidate State`, `Year of insertion`, and `Year of Recruitment`. The constraint includes a validity function that enforces the desired custom logic. This function generates

two masks: one for the `Hired` state and one for the other possible values of `Candidate State`. Then, a pandas series with a length equal to the input data index is initialized with `True` values. Finally, the series is updated to `False` at the indexes where the custom logic is not satisfied, using the two masks and applying the appropriate condition for each mask (as explained in the previous section).

The resulting validity pandas series is then passed as input to the `create_custom_constraint_class` function from the `sdv.constraints` module, ensuring that the validity check is applied during the synthesizer construction.

# 4 Polarized Data Generation

The SDV library offers two methods to generate polarized data using a trained synthesizer: `sample_from_conditions` and `sample_remaining_columns`.

- `sample_from_conditions:` This function takes as input **Conditions**, which specify the number of rows to generate and the column values to be used.
  The advantage is that it is very fast; however, it does not check for distribution constraints.
  As a result, when multiple conditions are applied, there may be collisions with the distribution constraints.
  For example: If we want to generate 10 rows where Sex = Female and 10 rows where State = Hired, some of the "Hired" rows may also have Sex = Female, leading to more than 10 Female rows in total.

- `sample_remaining_columns:` This function starts with a dataset containing some columns and all the required rows.
  It then generates the remaining columns while ensuring that all constraints are maintained.
  The benefit of this approach is that it checks for and resolves distribution collisions, ensuring the correct distribution. However, this increases computation time, and with very complex distribution constraints, the function may fail to generate the desired data.

## 4.1 Our Solution

To design our final solution, we started with the built-in function `sample_from_conditions`, as it was the fastest and easiest to build upon.
We generate data for each desired condition, producing more data than required. Then, we filter out the data that conflicts with other constraints, ensuring that the final distribution matches the intended one.
If the filtered data falls short of the required amount, we use a retry function that generates additional data and filters it again, repeating the process until the maximum retry limit is reached.
In such cases, the function will fail, but we have never encountered a failure during our experiments.

# 5 Performance comparison

## Experiment 1

In this experiment we want to have 25% of rows with: Sex=Female.

| Group | Field | Percentage |
|---|---|---|
| **1** | **Sex:** Female | 25% |

Table 2: Polarization conditions for Experiment 1.

## Experiment 2

In this experiment we want to have 25% of rows with: Sex=Female ∧ Candidate State=Hired .

| Group | Field | Percentage |
|---|---|---|
| **1** | **Sex:** Female | 25% |
| | **Candidate State:** Hired | 25% |

Table 3: Polarization conditions for Experiment 2.

## Experiment 3

In this experiment we want to have 25% of rows with: Sex=Female ∧ Candidate State=Hired. We also want to have 10% of rows with: Study Title=Five-year degree ∧ Assumption Headquarters=Milan ∧ English Level=3.

| Group | Field | Percentage |
|---|---|---|
| **1** | **Sex:** Female | 25% |
| | **Candidate State:** Hired | 25% |
| **2** | **Study Title:** Five-year degree | 10% |
| | **Assumption Headquarters:** Milan | 10% |
| | **English Level:** 3 | 10% |

Table 4: Polarization conditions for Experiment 3.

## Experiment Performances

| Experiment | From Conditions (Ours) | Remaining Columns |
|---|---|---|
| 1 | **1.00 sec** | 1.46 sec |
| 2 | **3.52 sec** | 7.96 sec |
| 3 | **9.97 sec** | Failed |

Table 5: Polarization function performances