

Supplementary Information

Contents

Supplementary Note A	Details of experimental setups	S2
A.1	Statistics of the evaluated datasets	S2
A.2	Implementation details	S2
A.3	Evaluation metrics	S3
Supplementary Note B	Detailed results	S5
B.1	Statistical fidelity results	S5
B.2	Downstream predictive results	S5
B.3	DCR results	S5
B.4	δ -presence results	S5
Supplementary Note C	Visualization of tree-derived variable dependence	S12
Supplementary Note D	Prompt and generated strategy for the <i>Adult</i> dataset	S12
D.1	Original prompt provided to the LLM	S12
D.2	LLM-generated analysis and synthesis strategy	S16
Supplementary Note E	Ablation results	S19
E.1	Contribution of feature-dependence extraction	S19
E.2	Effect of dependence information encoding	S19
E.3	Contribution of prompt guidance	S19
E.4	Sensitivity to different LLM backbones	S20
E.5	Sensitivity to LLM query budget	S20
Supplementary Note F	Detailed runtime of different algorithms on various datasets	S22
Supplementary Note G	Case study of failure modes	S22
Supplementary Note H	Detailed predictive results for clinical data augmentation	S23
Supplementary Note I	Tailored data synthesis strategies	S23

Supplementary Note A Details of experimental setups

A.1 Statistics of the evaluated datasets

We evaluate our method on 13 real-world tabular datasets spanning classification and regression tasks, small-sample settings, and high-dimensional feature spaces. These datasets contain between 4 and 450 features and between 174 and 53,940 samples. Each dataset is divided into training and test sets for downstream machine-learning evaluation. For the Adult dataset, we adopt the official train–test split. For the Thoracic dataset, we use a fixed split of 270 training samples and 200 test samples for the augmentation experiments. The remaining datasets are partitioned using either a 9:1 or 7:3 train–test ratio with a fixed random seed. To evaluate the applicability of our method to high-dimensional data, we additionally apply the greedy set-cover procedure in Algorithm 1 to the two highest-dimensional datasets, *DNA* and *DARWIN*, selecting a compact subset of trees that maximizes variable coverage while controlling prompt length. This procedure retains 19 and 54 trees in total, respectively, including the label tree. Dataset statistics are summarized in Supplementary Table 1.

Supplementary Table 1 Statistics of the evaluated datasets. Dataset names are hyperlinked to their public sources. Num and Cat denote the numbers of numerical and categorical predictor features, respectively. Imb. denotes the majority-to-minority class ratio in the training set.

Dataset	Domain	Source	Samples	Num	Cat	Train	Test	Task	Imb.
Adult	Social Sci.	UCI	48842	6	8	32561	16281	Classification	3.153
Default	Finance	UCI	30000	14	9	27000	3000	Classification	3.523
Magic	Physics	UCI	19019	10	0	17117	1902	Classification	1.847
Diamonds	Commerce	OpenML	53940	8	3	48546	5394	Regression	–
Blood	Medicine	UCI	748	4	0	523	225	Classification	3.023
Car	Decision	UCI	1728	0	6	1209	519	Classification	19.409
Heart	Medicine	UCI	270	5	8	189	81	Classification	1.305
Liver	Medicine	OpenML	345	5	0	241	104	Regression	–
SpeedDating	Social Sci.	OpenML	8378	53	61	7540	838	Classification	5.081
Jasmine	Life Sci.	OpenML	2984	8	136	2685	299	Classification	1.008
DNA	Biology	OpenML	3186	0	180	2230	956	Classification	2.222
DARWIN	Medicine	UCI	174	450	0	121	53	Classification	1.051
Thoracic	Medical	UCI	470	3	13	270	200	Classification	5.585

A.2 Implementation details

For each dataset, we fit all synthesis methods exclusively on the real training split and use each method to generate a synthetic table with the same number of rows as the training split. All methods are evaluated under the same train–test partition and downstream evaluation protocol. Downstream utility is evaluated using the train-on-synthetic, test-on-real protocol with AdaBoost [37], Random Forest [38], and XGBoost [39] as downstream predictors; the reported utility score is the average normalized performance across these models. The reported results are averaged over 20 randomly sampled synthetic data following TABSYN [34]. No information from the held-out test set is used for synthetic-data synthesis, candidate-strategy selection, or hyperparameter selection.

In the main experiments, we use Gemini-2.5-Pro [51] as the LLM backbone. For each dataset, the LLM is queried ten times to generate candidate Python-based synthesis strategies. Sensitivity to the number of LLM queries is analysed in Supplementary Fig. 2. Each candidate strategy is required to follow a unified executable interface, taking the local training table and the target sample size as inputs and saving the generated synthetic table to disk. Candidates that fail to execute, violate the required interface, or produce invalid output tables are discarded before strategy selection. The final strategy is selected using statistics computed only from the real training split, including statistical fidelity, empirical privacy-related risk, and computational efficiency. The held-out test set is used only for final evaluation.

Our framework first extracts dependence information from the original training data using chained trees and then provides only the encoded dependence summaries to the LLM. The original records are used locally for dependence extraction, candidate-strategy execution, and evaluation, but are not transmitted to the LLM during strategy generation. This design separates strategy-level reasoning from record-level data exposure while allowing the generated executable strategy to operate locally on the training data.

The baseline methods include (1) Conditional Tabular Generative Adversarial Network (CTGAN) [31], which uses conditional generation and mode-specific normalization to model multimodal variables, (2) TVAE [31], which learns a probabilistic latent representation of mixed-type tabular data using a variational autoencoder, (3) Generation of Realistic Tabular Data (GReaT) [23], which serializes table rows as text and

generates synthetic records autoregressively using a fine-tuned LLM, (4) diffusion-based methods, which generate mixed-type tabular data by progressively corrupting real samples and learning to reverse the resulting diffusion process, including Tabular Denoising Diffusion Probabilistic Model (TabDDPM) [32], TabDiff [33], and TABSYN [34], (5) Adversarial Random Forests (ARF) [35], a tree-based model that iteratively trains random forests to distinguish real from synthetic records and samples from the resulting leaf-wise distributions, and (6) NRGBoost [36], an energy-based generative model that learns a generative energy function through boosted regression trees and samples from the resulting distribution. In addition to the tabular generative models, we include the Synthetic Minority Oversampling Technique (SMOTE) [7] as a baseline due to its strong performance on synthetic data fidelity. However, SMOTE relies on interpolation between neighbouring records and duplicates specific local geometries. Real records may be reconstructed from synthetic samples [40], incurring severe privacy vulnerabilities that hinder SMOTE’s applicability to secure data-sharing contexts. By benchmarking against SMOTE, we demonstrate that TabSSD achieves competitive synthetic data fidelity and predictive utility but provides a more viable solution for privacy-preserving data synthesis. ARF, TabDiff, and NRGBoost are implemented using their official codebases with the recommended experimental configurations. TABSYN, TabDDPM, and GReaT are implemented using the code and configurations released by the TABSYN authors, whereas CTGAN and TVAE are implemented using the SDV library with its default configurations [52]. Unless otherwise specified, we follow the default or recommended configurations of the corresponding implementations and use the same train–test splits and evaluation protocol for all methods. All experiments are conducted on a workstation equipped with two NVIDIA RTX 4090 GPUs and two Intel Xeon Gold 6430 CPUs. While TABSYN, TabDDPM, GReaT, TabDiff, CTGAN, and TVAE relied on GPU acceleration, our method, TabSSD, required no GPU, demonstrating substantially lower hardware requirements and greater accessibility.

A.3 Evaluation metrics

A.3.1 Distance to closest record (DCR)

Following the DCR evaluation protocol adopted by TABSYN [34], we assess whether synthetic records exhibit excessive proximity to real training records. For each synthetic record, we compare its nearest-neighbour distances to the training and test sets and compute the proportion for which the nearest real neighbour belongs to the training set:

$$\text{DCR} = \frac{1}{|\hat{\mathcal{D}}|} \sum_{i=1}^{|\hat{\mathcal{D}}|} \mathbb{I}[d_{\text{train}}(\hat{\mathbf{x}}_i) < d_{\text{test}}(\hat{\mathbf{x}}_i)]. \quad (5)$$

Because this proportion is affected by unequal train–test split sizes, we centre it relative to the split-induced null expectation. Under the null hypothesis of no preferential proximity to the training data, exchangeability implies that the nearest real neighbour belongs to the training set with probability $|\mathcal{D}_{\text{train}}|/(|\mathcal{D}_{\text{train}}| + |\mathcal{D}_{\text{test}}|)$. We therefore define

$$\text{DCR}_{\text{adjusted}} = \max\left(\text{DCR} - \frac{|\mathcal{D}_{\text{train}}|}{|\mathcal{D}_{\text{train}}| + |\mathcal{D}_{\text{test}}|}, 0\right). \quad (6)$$

The positive-part operation ensures that only proximity exceeding the split-induced expectation contributes to the risk score; values at or below this expectation are assigned a score of zero. Throughout this paper, adjusted DCR refers to this split-ratio-adjusted score. Lower values indicate less excessive proximity to training records and therefore lower potential memorization risk.

A.3.2 δ -presence

We further use δ -presence to evaluate the re-identification risk of the generated synthetic data. Unlike DCR, which measures whether synthetic records are excessively close to real training records, δ -presence characterizes the worst-case imbalance between real and synthetic records across cluster-based partitions. In our experiments, δ -presence is computed following the implementation in SynthCity version 0.2.12 [53]. Specifically, k -means clustering models are fitted to the real training data using the full encoded feature space, and synthetic records are then assigned to the learned clusters using the same clustering models.

Let $c_K(\cdot)$ denote the clustering function learned from $\mathcal{D}_{\text{train}}$ using K clusters. For the q -th cluster, the numbers of real and synthetic records assigned to the same partition are defined as

$$n_{K,q}^{\text{real}} = \sum_{\mathbf{x}_i \in \mathcal{D}_{\text{train}}} \mathbb{I}[c_K(\mathbf{x}_i) = q], \quad (7)$$

113 and

$$114 \quad n_{K,q}^{\text{syn}} = \sum_{\hat{\mathbf{x}}_i \in \hat{\mathcal{D}}} \mathbb{I}[c_K(\hat{\mathbf{x}}_i) = q]. \quad (8)$$

115 For each cluster represented in both the real and synthetic datasets, the cluster-wise δ -presence score is
116 computed as

$$117 \quad \delta_{K,q} = \frac{n_{K,q}^{\text{real}}}{n_{K,q}^{\text{syn}} + \epsilon}, \quad (9)$$

118 where $\epsilon = 10^{-8}$ is a small constant used for numerical stability. Following the SynthCity implementation,
119 we evaluate multiple partition granularities with $K \in \{2, 5, 10, 15\}$ and skip a clustering setting when the
120 average number of real records per cluster is smaller than 10, i.e., $|\mathcal{D}_{\text{train}}|/K < 10$.

121 The final δ -presence score is defined as the maximum cluster-wise ratio over all retained clustering
122 granularities and clusters represented in both datasets:

$$123 \quad \delta\text{-presence} = \max_{\substack{K,q: \\ n_{K,q}^{\text{syn}} > 0}} \delta_{K,q}. \quad (10)$$

124 A larger δ -presence value indicates a larger worst-case real-to-synthetic cluster-count ratio, meaning that
125 at least one partition contains relatively many real records compared with synthetic records. Accordingly, in
126 our privacy evaluation, a lower δ -presence value is regarded as preferable.

127 A.3.3 Downstream utility

128 For downstream utility, we adopt the train-on-synthetic, test-on-real protocol. Predictive models are trained
129 on synthetic data and evaluated on the real test set, and their results are compared with those of the same
130 models trained on real data. We use AdaBoost [37], Random Forest [38], and XGBoost [39] as downstream
131 predictors and average their results. Specifically, the utility score is computed as

$$132 \quad \text{Utility} = \begin{cases} \frac{\text{AUROC}_{\text{syn}}}{\text{AUROC}_{\text{real}}}, & \text{for classification tasks,} \\ \frac{\text{RMSE}_{\text{real}}}{\text{RMSE}_{\text{syn}}}, & \text{for regression tasks,} \end{cases}$$

133 where $\text{AUROC}_{\text{syn}}$ and RMSE_{syn} denote the averaged results obtained by training on synthetic data, while
134 $\text{AUROC}_{\text{real}}$ and $\text{RMSE}_{\text{real}}$ denote the corresponding averaged results obtained by training on real data.
135 Because higher AUROC and lower RMSE indicate better predictive performance, a utility value closer to 1
136 indicates that training on synthetic data achieves performance closer to training on real data.

137 A.3.4 Marginal distribution error and pairwise correlation error

138 Following TABSYN [34], we evaluate the statistical fidelity of the generated synthetic data using marginal
139 distribution error and pairwise correlation error.

140 Marginal distribution error assesses whether the marginal distribution of each individual variable is
141 preserved in the synthetic data. For numerical variables, distributional similarity is evaluated using the
142 Kolmogorov–Smirnov statistic, whereas for categorical variables, total variation distance is used to compare
143 category-frequency distributions.

144 Pairwise correlation error assesses whether relationships between variable pairs are preserved. For
145 numerical variable pairs, similarity is evaluated by comparing the Pearson correlation coefficients computed
146 from the real and synthetic data. For categorical variable pairs, contingency similarity is used to compare
147 their joint-frequency distributions. For mixed-type variable pairs, the numerical variable is first discretized
148 into bins, after which contingency similarity is computed.

149 In our experiments, these metrics are computed using the single-table Quality Report in SDMetrics
150 version 0.9.3 [54]. The marginal distribution error is defined as one minus the Column Shapes score, and the
151 pairwise correlation error is defined as one minus the Column Pair Trends score. The Column Shapes score
152 averages KSComplement scores for numerical variables and TVComplement scores for categorical variables,
153 whereas the Column Pair Trends score averages CorrelationSimilarity scores for numerical–numerical pairs
154 and ContingencySimilarity scores for categorical–categorical and mixed-type pairs.

155 A.3.5 α -precision

156 We use α -precision to evaluate the sample-level fidelity of the generated synthetic data. Unlike the marginal
157 distribution estimation and pairwise correlation estimation metrics, which focus on marginal distributions

and inter-feature relationships, α -precision evaluates whether synthetic samples lie within real-data regions defined by quantile radii around the real-data centre. Following the implementation in SynthCity version 0.2.12 [53], we compute the naive α -precision score, where real and synthetic data are first normalized using a scaler fitted on the real data, and the centre of the normalized real data is used as the reference point. The metric compares the empirical α -precision curve with the ideal curve across different quantile radii. A higher α -precision value indicates closer agreement between the empirical and ideal curves and therefore better sample-level fidelity to the real data distribution.

Supplementary Note B Detailed results

B.1 Statistical fidelity results

Dataset-level statistical fidelity results are reported for marginal distribution error (Supplementary Table 2), pairwise correlation error (Supplementary Table 3), and α -precision (Supplementary Table 4), capturing complementary aspects of marginal, inter-variable, and distributional fidelity.

B.2 Downstream predictive results

Dataset-level downstream predictive performance is reported in Supplementary Table 5, comparing models trained on synthetic data with the corresponding real-data reference performance. Classification datasets report mean AUROC, whereas the regression datasets (*Diamonds* and *Liver*) report mean RMSE. Results are averaged across AdaBoost, Random Forest, and XGBoost.

B.3 DCR results

Dataset-level DCR results are reported in Supplementary Table 6, together with the corresponding train-set proportions used as references for assessing empirical memorization risk.

B.4 δ -presence results

Dataset-level δ -presence results are reported in Supplementary Table 7, providing an empirical assessment of re-identification risk across datasets and synthesis methods.

Supplementary Table 2 Dataset-level marginal distribution errors (%). Results are reported as mean $\pm$ s.d. across 20 runs. Lower values are better; dashes indicate unavailable results. The average row reports the mean of the dataset-level mean errors.

Dataset	SMOTE	ARF	NRGBoost	TVAE	CTGAN	TABSYN	TabDDPM	TabDiff	GReaT	TabSSD
<i>Adult</i>	1.57 $\pm$ 0.030	4.99 $\pm$ 0.000	1.80 $\pm$ 0.001	8.05 $\pm$ 0.000	8.87 $\pm$ 0.040	2.17 $\pm$ 0.056	1.18 $\pm$ 0.044	0.64 $\pm$ 0.047	12.74 $\pm$ 0.079	5.44 $\pm$ 0.001
<i>Blood</i>	5.57 $\pm$ 0.929	6.37 $\pm$ 0.006	8.04 $\pm$ 0.008	12.28 $\pm$ 0.000	19.27 $\pm$ 0.251	5.52 $\pm$ 0.554	5.34 $\pm$ 0.747	5.83 $\pm$ 0.701	10.67 $\pm$ 1.246	3.32 $\pm$ 0.005
<i>Car</i>	1.57 $\pm$ 0.267	1.58 $\pm$ 0.003	2.00 $\pm$ 0.005	3.85 $\pm$ 0.000	15.84 $\pm$ 0.224	1.99 $\pm$ 0.402	1.70 $\pm$ 0.379	1.73 $\pm$ 0.312	3.15 $\pm$ 0.428	1.68 $\pm$ 0.004
<i>Default</i>	1.56 $\pm$ 0.089	4.80 $\pm$ 0.000	3.92 $\pm$ 0.001	9.29 $\pm$ 0.000	7.48 $\pm$ 0.025	1.08 $\pm$ 0.057	1.50 $\pm$ 0.089	1.23 $\pm$ 0.101	20.17 $\pm$ 0.124	1.00 $\pm$ 0.001
<i>Diamonds</i>	1.57 $\pm$ 0.058	1.82 $\pm$ 0.001	5.61 $\pm$ 0.001	8.61 $\pm$ 0.000	5.18 $\pm$ 0.033	3.19 $\pm$ 0.097	0.93 $\pm$ 0.086	0.72 $\pm$ 0.096	4.88 $\pm$ 0.082	3.79 $\pm$ 0.000
<i>Heart</i>	5.40 $\pm$ 0.448	5.78 $\pm$ 0.006	5.87 $\pm$ 0.009	11.90 $\pm$ 0.000	20.23 $\pm$ 0.355	6.65 $\pm$ 0.565	5.03 $\pm$ 0.530	5.70 $\pm$ 0.836	11.42 $\pm$ 0.599	6.08 $\pm$ 0.007
<i>Jasmine</i>	1.38 $\pm$ 0.065	0.97 $\pm$ 0.001	3.09 $\pm$ 0.002	6.77 $\pm$ 0.000	11.15 $\pm$ 0.027	3.57 $\pm$ 0.052	28.16 $\pm$ 0.270	0.82 $\pm$ 0.079	–	0.48 $\pm$ 0.000
<i>Liver</i>	7.56 $\pm$ 0.764	10.55 $\pm$ 0.009	7.44 $\pm$ 0.013	12.17 $\pm$ 0.000	19.43 $\pm$ 0.293	9.14 $\pm$ 0.715	7.60 $\pm$ 0.727	8.61 $\pm$ 0.655	13.60 $\pm$ 0.910	9.20 $\pm$ 0.000
<i>Magic</i>	0.93 $\pm$ 0.070	2.67 $\pm$ 0.001	2.37 $\pm$ 0.002	8.71 $\pm$ 0.000	8.14 $\pm$ 0.055	0.90 $\pm$ 0.102	0.99 $\pm$ 0.109	0.83 $\pm$ 0.095	15.07 $\pm$ 0.097	3.09 $\pm$ 0.000
<i>SpeedDating</i>	1.82 $\pm$ 0.075	6.09 $\pm$ 0.001	12.59 $\pm$ 0.000	12.84 $\pm$ 0.000	10.04 $\pm$ 0.036	12.73 $\pm$ 0.047	53.09 $\pm$ 0.118	5.86 $\pm$ 0.101	–	1.11 $\pm$ 0.000
<i>DNA</i>	1.41 $\pm$ 0.001	0.73 $\pm$ 0.000	0.86 $\pm$ 0.000	13.15 $\pm$ 0.000	11.58 $\pm$ 0.000	8.09 $\pm$ 0.001	0.87 $\pm$ 0.001	2.12 $\pm$ 0.069	–	0.76 $\pm$ 0.000
<i>DARWIN</i>	11.43 $\pm$ 0.004	14.75 $\pm$ 0.002	14.60 $\pm$ 0.003	21.06 $\pm$ 0.000	46.19 $\pm$ 0.001	19.42 $\pm$ 0.003	52.53 $\pm$ 0.001	10.85 $\pm$ 0.204	–	9.67 $\pm$ 0.002
Average	3.48	5.09	5.68	10.72	15.28	6.20	13.24	3.74	11.46	3.80

Supplementary Table 3 Dataset-level pairwise correlation errors (%). Results are reported as mean $\pm$ s.d. across 20 runs. Lower values are better; dashes indicate unavailable results. The average row reports the mean of the dataset-level mean errors.

Dataset	SMOTE	ARF	NRGBoost	TVAE	CTGAN	TABSYN	TabDDPM	TabDiff	GReaT	TabSSD
<i>Adult</i>	2.78 $\pm$ 0.041	7.79 $\pm$ 0.000	2.92 $\pm$ 0.001	10.01 $\pm$ 0.000	13.88 $\pm$ 0.044	3.83 $\pm$ 0.065	2.35 $\pm$ 0.055	1.47 $\pm$ 0.056	18.00 $\pm$ 0.097	3.53 $\pm$ 0.000
<i>Blood</i>	2.83 $\pm$ 0.546	4.65 $\pm$ 0.006	8.25 $\pm$ 0.006	10.35 $\pm$ 0.000	13.35 $\pm$ 0.272	3.65 $\pm$ 0.368	3.46 $\pm$ 0.541	40.86 $\pm$ 0.322	6.58 $\pm$ 0.624	2.68 $\pm$ 0.004
<i>Car</i>	3.70 $\pm$ 0.313	3.68 $\pm$ 0.003	3.95 $\pm$ 0.005	6.94 $\pm$ 0.000	26.25 $\pm$ 0.242	4.12 $\pm$ 0.401	3.75 $\pm$ 0.427	4.15 $\pm$ 0.315	6.10 $\pm$ 0.503	3.99 $\pm$ 0.004
<i>Default</i>	1.68 $\pm$ 0.079	4.58 $\pm$ 0.001	5.00 $\pm$ 0.001	9.63 $\pm$ 0.000	5.34 $\pm$ 0.031	1.17 $\pm$ 0.046	5.92 $\pm$ 0.184	1.38 $\pm$ 0.101	14.40 $\pm$ 0.244	5.09 $\pm$ 0.001
<i>Diamonds</i>	0.77 $\pm$ 0.050	1.09 $\pm$ 0.000	5.09 $\pm$ 0.001	9.25 $\pm$ 0.000	7.70 $\pm$ 0.030	2.89 $\pm$ 0.063	2.93 $\pm$ 0.174	1.04 $\pm$ 0.053	6.15 $\pm$ 0.067	2.11 $\pm$ 0.000
<i>Heart</i>	10.53 $\pm$ 0.660	12.21 $\pm$ 0.006	10.81 $\pm$ 0.010	15.87 $\pm$ 0.000	26.55 $\pm$ 0.375	12.65 $\pm$ 0.710	8.91 $\pm$ 0.638	11.88 $\pm$ 0.845	19.20 $\pm$ 0.626	11.11 $\pm$ 0.006
<i>Jasmine</i>	2.05 $\pm$ 0.102	1.99 $\pm$ 0.001	5.06 $\pm$ 0.003	11.44 $\pm$ 0.000	16.89 $\pm$ 0.041	5.83 $\pm$ 0.089	44.64 $\pm$ 0.406	1.49 $\pm$ 0.123	–	0.96 $\pm$ 0.000
<i>Liver</i>	3.25 $\pm$ 0.809	3.98 $\pm$ 0.008	4.21 $\pm$ 0.013	5.77 $\pm$ 0.000	8.27 $\pm$ 0.579	3.57 $\pm$ 0.774	2.76 $\pm$ 0.701	2.98 $\pm$ 0.631	7.54 $\pm$ 0.940	3.49 $\pm$ 0.000
<i>Magic</i>	0.90 $\pm$ 0.095	1.36 $\pm$ 0.001	1.10 $\pm$ 0.001	4.45 $\pm$ 0.000	6.82 $\pm$ 0.059	0.61 $\pm$ 0.088	0.75 $\pm$ 0.082	0.63 $\pm$ 0.085	7.20 $\pm$ 0.116	0.98 $\pm$ 0.000
<i>SpeedDating</i>	2.92 $\pm$ 0.084	9.02 $\pm$ 0.000	10.42 $\pm$ 0.000	15.45 $\pm$ 0.000	11.02 $\pm$ 0.030	19.08 $\pm$ 0.057	64.17 $\pm$ 0.053	8.22 $\pm$ 0.119	–	5.07 $\pm$ 0.000
<i>DNA</i>	2.37 $\pm$ 0.001	1.61 $\pm$ 0.000	1.59 $\pm$ 0.001	25.67 $\pm$ 0.000	18.50 $\pm$ 0.000	12.60 $\pm$ 0.001	13.77 $\pm$ 0.001	3.43 $\pm$ 0.099	–	1.75 $\pm$ 0.000
<i>DARWIN</i>	4.16 $\pm$ 0.003	6.20 $\pm$ 0.001	7.13 $\pm$ 0.000	5.91 $\pm$ 0.000	16.09 $\pm$ 0.001	6.57 $\pm$ 0.001	7.84 $\pm$ 0.000	5.70 $\pm$ 0.115	–	5.91 $\pm$ 0.001
Average	3.16	4.85	5.46	10.89	14.22	6.38	13.44	6.94	10.65	3.89

Supplementary Table 4 Dataset-level α -precision scores. Results are reported as mean $\pm$ s.d. across 20 runs. Higher values indicate better statistical fidelity; dashes indicate unavailable results. The average row reports the mean of the dataset-level mean scores.

Dataset	SMOTE	ARF	NRGBoost	TVAE	CTGAN	TABSYN	TabDDPM	TabDiff	GReaT	TabSSD
<i>Adult</i>	0.9307 $\pm$ 0.002	0.9917 $\pm$ 0.002	0.9514 $\pm$ 0.003	0.9777 $\pm$ 0.000	0.7666 $\pm$ 0.002	0.9791 $\pm$ 0.002	0.9529 $\pm$ 0.003	0.9914 $\pm$ 0.002	0.5443 $\pm$ 0.001	0.9944 $\pm$ 0.002
<i>Blood</i>	0.9706 $\pm$ 0.009	0.9683 $\pm$ 0.010	0.9105 $\pm$ 0.020	0.8857 $\pm$ 0.000	0.7663 $\pm$ 0.005	0.9638 $\pm$ 0.010	0.9642 $\pm$ 0.012	0.1718 $\pm$ 0.008	0.9413 $\pm$ 0.015	0.9627 $\pm$ 0.011
<i>Car</i>	0.9860 $\pm$ 0.006	0.9808 $\pm$ 0.006	0.9688 $\pm$ 0.019	0.9751 $\pm$ 0.000	0.9319 $\pm$ 0.005	0.9825 $\pm$ 0.008	0.9826 $\pm$ 0.006	0.9825 $\pm$ 0.007	0.9603 $\pm$ 0.014	0.9848 $\pm$ 0.005
<i>Default</i>	0.9790 $\pm$ 0.003	0.9427 $\pm$ 0.002	0.9406 $\pm$ 0.003	0.8576 $\pm$ 0.000	0.9179 $\pm$ 0.001	0.9885 $\pm$ 0.002	0.9893 $\pm$ 0.002	0.9872 $\pm$ 0.003	0.8611 $\pm$ 0.002	0.9790 $\pm$ 0.001
<i>Diamonds</i>	0.9954 $\pm$ 0.002	0.9962 $\pm$ 0.002	0.9553 $\pm$ 0.001	0.9442 $\pm$ 0.000	0.8667 $\pm$ 0.002	0.9743 $\pm$ 0.003	0.9956 $\pm$ 0.002	0.9964 $\pm$ 0.001	0.8522 $\pm$ 0.003	0.9957 $\pm$ 0.001
<i>Heart</i>	0.9042 $\pm$ 0.036	0.9527 $\pm$ 0.016	0.9462 $\pm$ 0.020	0.7023 $\pm$ 0.000	0.8516 $\pm$ 0.018	0.8934 $\pm$ 0.033	0.9340 $\pm$ 0.029	0.9504 $\pm$ 0.021	0.6085 $\pm$ 0.032	0.9442 $\pm$ 0.021
<i>Jasmine</i>	0.8616 $\pm$ 0.009	0.9403 $\pm$ 0.007	0.7595 $\pm$ 0.012	0.4716 $\pm$ 0.000	0.7703 $\pm$ 0.003	0.7499 $\pm$ 0.010	0.2855 $\pm$ 0.012	0.9792 $\pm$ 0.007	–	0.9391 $\pm$ 0.007
<i>Liver</i>	0.8878 $\pm$ 0.032	0.8791 $\pm$ 0.036	0.9071 $\pm$ 0.031	0.7834 $\pm$ 0.000	0.7473 $\pm$ 0.012	0.9404 $\pm$ 0.019	0.9646 $\pm$ 0.011	0.9386 $\pm$ 0.017	0.6761 $\pm$ 0.028	0.9659 $\pm$ 0.000
<i>Magic</i>	0.9814 $\pm$ 0.003	0.9865 $\pm$ 0.002	0.9916 $\pm$ 0.002	0.8365 $\pm$ 0.000	0.8547 $\pm$ 0.001	0.9918 $\pm$ 0.003	0.9827 $\pm$ 0.004	0.9941 $\pm$ 0.002	0.8610 $\pm$ 0.005	0.9925 $\pm$ 0.000
<i>SpeedDating</i>	0.9391 $\pm$ 0.006	0.9323 $\pm$ 0.003	0.8807 $\pm$ 0.004	0.7526 $\pm$ 0.000	0.9683 $\pm$ 0.003	0.6902 $\pm$ 0.006	0.0832 $\pm$ 0.001	0.7229 $\pm$ 0.007	–	0.8948 $\pm$ 0.002
<i>DNA</i>	0.8145 $\pm$ 0.014	0.9669 $\pm$ 0.009	0.8654 $\pm$ 0.007	0.0771 $\pm$ 0.000	0.6164 $\pm$ 0.004	0.3043 $\pm$ 0.007	0.9491 $\pm$ 0.011	0.6674 $\pm$ 0.012	–	0.9550 $\pm$ 0.009
<i>DARWIN</i>	0.5071 $\pm$ 0.050	0.6772 $\pm$ 0.018	0.1892 $\pm$ 0.004	0.0000 $\pm$ 0.000	0.0000 $\pm$ 0.000	0.0002 $\pm$ 0.000	0.0000 $\pm$ 0.000	0.2002 $\pm$ 0.016	–	0.6623 $\pm$ 0.019
Average	0.8964	0.9346	0.8555	0.6887	0.7548	0.7882	0.7570	0.7985	0.7881	0.9392

Supplementary Table 5 Dataset-level downstream predictive performance. Real denotes the reference performance obtained by training on real data. Classification datasets report mean AUROC, whereas the regression datasets (*Diamonds* and *Liver*) report mean RMSE. Results are averaged across AdaBoost, Random Forest, and XGBoost. Synthetic-data results are reported as mean $\pm$ s.d. across 20 runs. Higher AUROC and lower RMSE indicate better performance; dashes indicate unavailable results.

Dataset	Real	SMOTE	ARF	NRGBoost	TVAE	CTGAN	TABSYN	TabDDPM	TabDiff	GReaT	TabSSD
<i>Adult</i>	0.9209	0.9003 $\pm$ 0.001	0.9065 $\pm$ 0.001	0.9112 $\pm$ 0.001	0.8889 $\pm$ 0.000	0.8767 $\pm$ 0.004	0.9063 $\pm$ 0.001	0.9100 $\pm$ 0.001	0.9107 $\pm$ 0.001	0.9062 $\pm$ 0.002	0.9001 $\pm$ 0.001
<i>Blood</i>	0.7353	0.7026 $\pm$ 0.023	0.7169 $\pm$ 0.045	0.5786 $\pm$ 0.089	0.7935 $\pm$ 0.000	0.7959 $\pm$ 0.017	0.7229 $\pm$ 0.046	0.6860 $\pm$ 0.059	0.7235 $\pm$ 0.032	0.6127 $\pm$ 0.072	0.7036 $\pm$ 0.026
<i>Car</i>	0.9789	0.9772 $\pm$ 0.006	0.9084 $\pm$ 0.010	0.9790 $\pm$ 0.006	0.9305 $\pm$ 0.000	0.7924 $\pm$ 0.040	0.9637 $\pm$ 0.004	0.9780 $\pm$ 0.007	0.8994 $\pm$ 0.016	0.9521 $\pm$ 0.006	0.9577 $\pm$ 0.004
<i>Default</i>	0.7717	0.7492 $\pm$ 0.004	0.7514 $\pm$ 0.002	0.7566 $\pm$ 0.002	0.7319 $\pm$ 0.000	0.7278 $\pm$ 0.003	0.7619 $\pm$ 0.002	0.7607 $\pm$ 0.002	0.7617 $\pm$ 0.004	0.7494 $\pm$ 0.004	0.7520 $\pm$ 0.002
<i>Diamonds</i>	836.93	860.13 $\pm$ 0.52	1027.78 $\pm$ 2.78	956.81 $\pm$ 7.24	1544.25 $\pm$ 0.00	1521.09 $\pm$ 0.47	927.47 $\pm$ 1.21	949.41 $\pm$ 16.25	891.76 $\pm$ 39.39	984.00 $\pm$ 26.59	934.09 $\pm$ 5.11
<i>Heart</i>	0.8498	0.8417 $\pm$ 0.024	0.8022 $\pm$ 0.031	0.8030 $\pm$ 0.031	0.8467 $\pm$ 0.000	0.7910 $\pm$ 0.021	0.8348 $\pm$ 0.025	0.8416 $\pm$ 0.020	0.8409 $\pm$ 0.023	0.7698 $\pm$ 0.038	0.8233 $\pm$ 0.028
<i>Jasmine</i>	0.8860	0.8704 $\pm$ 0.010	0.8518 $\pm$ 0.009	0.8668 $\pm$ 0.008	0.8556 $\pm$ 0.000	0.6231 $\pm$ 0.047	0.8456 $\pm$ 0.013	0.5185 $\pm$ 0.077	0.8549 $\pm$ 0.009	—	0.8622 $\pm$ 0.009
<i>Liver</i>	3.4243	3.4096 $\pm$ 0.090	3.5508 $\pm$ 0.102	3.5743 $\pm$ 0.184	3.4199 $\pm$ 0.000	3.8220 $\pm$ 0.099	3.3874 $\pm$ 0.147	3.5228 $\pm$ 0.153	3.5010 $\pm$ 0.186	4.1169 $\pm$ 0.209	3.1812 $\pm$ 0.000
<i>Magic</i>	0.9362	0.9320 $\pm$ 0.001	0.9137 $\pm$ 0.002	0.9252 $\pm$ 0.002	0.8934 $\pm$ 0.000	0.8832 $\pm$ 0.002	0.9293 $\pm$ 0.002	0.9256 $\pm$ 0.002	0.9281 $\pm$ 0.002	0.9012 $\pm$ 0.003	0.9206 $\pm$ 0.000
<i>SpeedDating</i>	0.8515	0.8357 $\pm$ 0.006	0.7995 $\pm$ 0.012	0.5051 $\pm$ 0.025	0.7640 $\pm$ 0.000	0.7755 $\pm$ 0.011	0.8137 $\pm$ 0.007	0.4952 $\pm$ 0.044	0.8251 $\pm$ 0.006	—	0.8010 $\pm$ 0.005
<i>DNA</i>	0.9848	0.9847 $\pm$ 0.002	0.9437 $\pm$ 0.008	0.9833 $\pm$ 0.002	0.9655 $\pm$ 0.000	0.4791 $\pm$ 0.020	0.9292 $\pm$ 0.010	0.4965 $\pm$ 0.031	0.9758 $\pm$ 0.003	—	0.9623 $\pm$ 0.006
<i>DARWIN</i>	0.9667	0.9081 $\pm$ 0.035	0.8007 $\pm$ 0.055	0.6241 $\pm$ 0.111	0.8758 $\pm$ 0.000	0.4977 $\pm$ 0.081	0.8470 $\pm$ 0.052	0.4580 $\pm$ 0.071	0.8633 $\pm$ 0.026	—	0.8805 $\pm$ 0.022

Supplementary Table 6 Dataset-level DCR values. DCR is the proportion of synthetic records closer to the training set than to the held-out test set. Results are reported as mean $\pm$ s.d. across 20 runs where available. Values closer to the train-set proportion indicate lower empirical memorization risk; dashes indicate unavailable results.

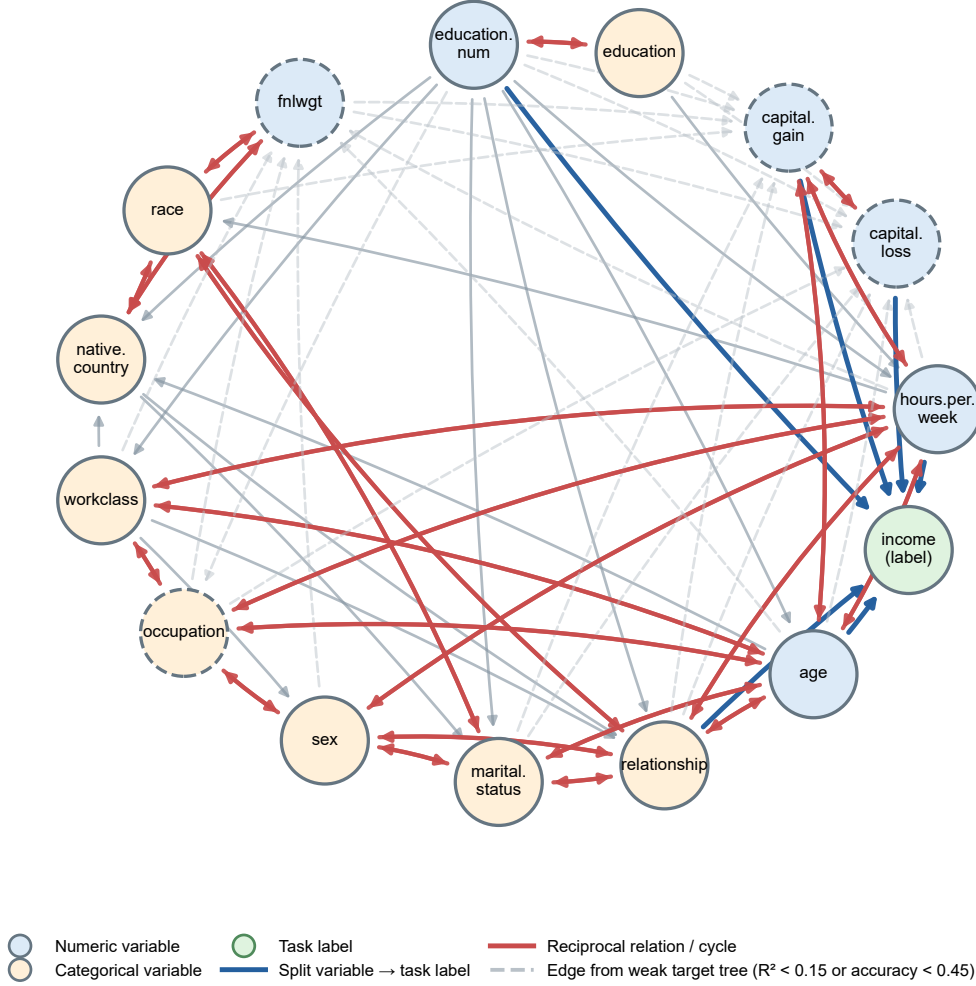
Dataset	Train prop.	SMOTE	ARF	NRGBoost	TVAE	CTGAN	TABSYN	TabDDPM	TabDiff	GReaT	TabSSD
<i>Adult</i>	0.67	0.9438 $\pm$ 0.001	0.7231 $\pm$ 0.003	0.7046 $\pm$ 0.003	0.6685 $\pm$ 0.000	0.6748 $\pm$ 0.002	0.6687 $\pm$ 0.003	0.6798 $\pm$ 0.002	0.7057 $\pm$ 0.003	0.6785 $\pm$ 0.004	0.6715 $\pm$ 0.002
<i>Blood</i>	0.70	0.8732 $\pm$ 0.011	0.7528 $\pm$ 0.021	0.9444 $\pm$ 0.005	0.7354 $\pm$ 0.000	0.7258 $\pm$ 0.007	0.7342 $\pm$ 0.022	0.7676 $\pm$ 0.014	0.7326 $\pm$ 0.016	0.7569 $\pm$ 0.012	0.7543 $\pm$ 0.017
<i>Car</i>	0.70	0.9972 $\pm$ 0.001	0.8099 $\pm$ 0.012	0.9973 $\pm$ 0.001	0.7438 $\pm$ 0.000	0.7419 $\pm$ 0.009	0.7574 $\pm$ 0.012	0.7644 $\pm$ 0.010	0.7156 $\pm$ 0.015	0.7186 $\pm$ 0.013	0.7090 $\pm$ 0.009
<i>Default</i>	0.90	0.9856 $\pm$ 0.001	0.9073 $\pm$ 0.002	0.9022 $\pm$ 0.002	0.8950 $\pm$ 0.000	0.9009 $\pm$ 0.002	0.9031 $\pm$ 0.002	0.9039 $\pm$ 0.002	0.9158 $\pm$ 0.002	0.9004 $\pm$ 0.002	0.9027 $\pm$ 0.002
<i>Diamonds</i>	0.90	0.9813 $\pm$ 0.001	0.9176 $\pm$ 0.001	0.9047 $\pm$ 0.002	0.8984 $\pm$ 0.000	0.9040 $\pm$ 0.001	0.9013 $\pm$ 0.002	0.9009 $\pm$ 0.001	0.9119 $\pm$ 0.001	0.9040 $\pm$ 0.001	0.9020 $\pm$ 0.001
<i>Heart</i>	0.70	0.9958 $\pm$ 0.004	0.7616 $\pm$ 0.031	0.9995 $\pm$ 0.002	0.8254 $\pm$ 0.000	0.7561 $\pm$ 0.027	0.9328 $\pm$ 0.017	0.9873 $\pm$ 0.007	0.8651 $\pm$ 0.020	0.7543 $\pm$ 0.021	0.7616 $\pm$ 0.035
<i>Jasmine</i>	0.90	1.0000 $\pm$ 0.000	0.9291 $\pm$ 0.004	0.9200 $\pm$ 0.005	0.9363 $\pm$ 0.000	0.9142 $\pm$ 0.003	0.9125 $\pm$ 0.006	0.9421 $\pm$ 0.004	0.9129 $\pm$ 0.005	–	0.9199 $\pm$ 0.006
<i>Liver</i>	0.70	0.9749 $\pm$ 0.011	0.7562 $\pm$ 0.025	1.0000 $\pm$ 0.000	0.7427 $\pm$ 0.000	0.7114 $\pm$ 0.016	0.7618 $\pm$ 0.028	1.0000 $\pm$ 0.000	0.7205 $\pm$ 0.026	0.7326 $\pm$ 0.036	0.7178 $\pm$ 0.000
<i>Magic</i>	0.90	0.9945 $\pm$ 0.001	0.9105 $\pm$ 0.002	0.9116 $\pm$ 0.002	0.8964 $\pm$ 0.000	0.8999 $\pm$ 0.002	0.9044 $\pm$ 0.002	0.9052 $\pm$ 0.002	0.9054 $\pm$ 0.002	0.9051 $\pm$ 0.003	0.8993 $\pm$ 0.000
<i>SpeedDating</i>	0.90	1.0000 $\pm$ 0.000	0.9419 $\pm$ 0.003	0.9056 $\pm$ 0.002	0.9188 $\pm$ 0.000	0.9111 $\pm$ 0.003	0.9082 $\pm$ 0.004	0.7022 $\pm$ 0.004	0.9716 $\pm$ 0.002	–	0.9090 $\pm$ 0.003
<i>DNA</i>	0.70	0.9830 $\pm$ 0.001	0.8537 $\pm$ 0.006	0.7999 $\pm$ 0.010	0.9350 $\pm$ 0.000	0.7985 $\pm$ 0.006	0.8773 $\pm$ 0.007	0.7541 $\pm$ 0.007	0.9533 $\pm$ 0.004	–	0.7687 $\pm$ 0.006
<i>DARWIN</i>	0.70	1.0000 $\pm$ 0.000	0.6583 $\pm$ 0.044	0.6215 $\pm$ 0.046	0.5455 $\pm$ 0.000	0.6810 $\pm$ 0.024	0.9256 $\pm$ 0.024	0.9146 $\pm$ 0.024	0.5983 $\pm$ 0.045	–	0.7287 $\pm$ 0.050

Supplementary Table 7 Dataset-level δ -presence scores. Results are reported as mean $\pm$ s.d. across 20 runs. Lower values indicate lower empirical re-identification risk; dashes indicate unavailable results. The average row reports the mean of the dataset-level mean scores.

Dataset	SMOTE	ARF	NRGBoost	TVAE	CTGAN	TABSYN	TabDDPM	TabDiff	GReaT	TabSSD
<i>Adult</i>	5.5076 $\pm$ 5.042	1.4935 $\pm$ 0.473	1.8159 $\pm$ 0.309	25.0000 $\pm$ 0.000	468.2476 $\pm$ 153.135	5.9231 $\pm$ 1.395	1.5995 $\pm$ 0.277	1.1333 $\pm$ 0.103	2.8423 $\pm$ 0.341	1.3100 $\pm$ 0.297
<i>Blood</i>	2.9354 $\pm$ 1.013	2.7067 $\pm$ 0.884	2.4535 $\pm$ 1.023	7.0000 $\pm$ 0.000	3.4894 $\pm$ 0.195	10.0354 $\pm$ 5.799	3.3739 $\pm$ 1.408	8.0617 $\pm$ 4.713	6.3392 $\pm$ 2.833	2.5710 $\pm$ 0.968
<i>Car</i>	1.2454 $\pm$ 0.091	1.2465 $\pm$ 0.111	1.2899 $\pm$ 0.136	1.6444 $\pm$ 0.000	11.1778 $\pm$ 6.092	1.2876 $\pm$ 0.080	1.2679 $\pm$ 0.090	1.2646 $\pm$ 0.102	1.4477 $\pm$ 0.159	1.2093 $\pm$ 0.069
<i>Default</i>	2.7467 $\pm$ 1.176	1.9903 $\pm$ 1.094	1.5135 $\pm$ 0.117	2.2105 $\pm$ 0.000	2.1417 $\pm$ 0.116	1.3138 $\pm$ 0.176	1.2053 $\pm$ 0.040	1.5399 $\pm$ 0.615	5.3066 $\pm$ 0.654	1.6712 $\pm$ 0.209
<i>Diamonds</i>	1.1531 $\pm$ 0.059	1.0982 $\pm$ 0.033	2.0167 $\pm$ 0.101	1.8487 $\pm$ 0.000	1.3069 $\pm$ 0.009	1.1529 $\pm$ 0.013	1.0989 $\pm$ 0.020	1.0721 $\pm$ 0.019	1.0446 $\pm$ 0.016	4.1604 $\pm$ 0.275
<i>Heart</i>	4.1689 $\pm$ 2.114	2.2363 $\pm$ 0.770	2.8924 $\pm$ 1.355	6.0000 $\pm$ 0.000	26.0000 $\pm$ 0.000	6.1792 $\pm$ 3.822	2.5880 $\pm$ 1.086	3.1417 $\pm$ 1.073	11.1667 $\pm$ 4.065	2.9773 $\pm$ 1.528
<i>Jasmine</i>	3.0117 $\pm$ 0.670	1.4277 $\pm$ 0.125	2.2063 $\pm$ 0.160	5.1667 $\pm$ 0.000	179.2250 $\pm$ 74.877	4.4896 $\pm$ 4.162	336.2175 $\pm$ 204.257	1.3864 $\pm$ 0.137	–	1.5088 $\pm$ 0.172
<i>Liver</i>	3.4452 $\pm$ 1.116	2.5318 $\pm$ 0.914	2.6779 $\pm$ 0.849	4.6000 $\pm$ 0.000	14.3500 $\pm$ 5.776	3.8750 $\pm$ 1.907	2.7193 $\pm$ 0.976	3.2900 $\pm$ 1.557	12.4000 $\pm$ 4.659	2.3333 $\pm$ 0.000
<i>Magic</i>	1.2153 $\pm$ 0.133	1.1471 $\pm$ 0.038	1.0934 $\pm$ 0.025	9.0645 $\pm$ 0.000	2.0099 $\pm$ 0.234	1.1348 $\pm$ 0.086	1.4130 $\pm$ 0.186	1.2422 $\pm$ 0.179	112.3600 $\pm$ 71.578	1.1601 $\pm$ 0.000
<i>SpeedDating</i>	2.3761 $\pm$ 0.324	1.8952 $\pm$ 0.229	2.5269 $\pm$ 0.294	47.7500 $\pm$ 0.000	2.4987 $\pm$ 0.291	5.6823 $\pm$ 0.781	654.5083 $\pm$ 381.861	2.1643 $\pm$ 0.200	–	1.7783 $\pm$ 0.129
<i>DNA</i>	1.3362 $\pm$ 0.090	1.5581 $\pm$ 0.298	2.7525 $\pm$ 4.189	3.7500 $\pm$ 0.000	38.5008 $\pm$ 24.136	4.5089 $\pm$ 1.957	3.6837 $\pm$ 0.539	2.7289 $\pm$ 1.154	–	1.9511 $\pm$ 0.312
<i>DARWIN</i>	3.9152 $\pm$ 1.944	2.8598 $\pm$ 1.908	5.1759 $\pm$ 2.720	7.0000 $\pm$ 0.000	13.0917 $\pm$ 5.863	3.6742 $\pm$ 1.005	24.1524 $\pm$ 9.551	2.5042 $\pm$ 0.878	–	5.5708 $\pm$ 2.564
Average	2.7547	1.8493	2.3679	10.0862	63.5033	4.1047	86.1523	2.4608	19.1134	2.3501

Supplementary Note C Visualization of tree-derived variable dependence

Supplementary Fig. 1 visualizes the raw variable-dependence signals extracted by the chained trees for the *Adult* dataset. The graph contains overlapping, reciprocal, and cyclic associations and should not be interpreted as a coherent causal structure.



Supplementary Fig. 1 Raw variable-dependence signals extracted by the chained trees for the *Adult* dataset. Nodes represent the input variables and task label. A directed edge $X_j \rightarrow X_i$ indicates that X_j was selected as a splitting variable by the CART fitted to predict X_i . Because variables can appear in one another’s prediction trees, the resulting graph may contain reciprocal, redundant, or cyclic dependence signals. Grey dashed edges denote signals extracted from trees with weak predictive performance, defined as regression trees with $R^2 < 0.15$ or classification trees with accuracy < 0.45 . These edges represent predictive associations rather than causal relationships.

Supplementary Note D Prompt and generated strategy for the *Adult* dataset

This section presents the original prompt and the corresponding LLM-generated synthesis strategy for the *Adult* dataset. Both are reproduced verbatim to preserve the actual strategy-generation process and may differ from the terminology used in the manuscript.

D.1 Original prompt provided to the LLM

According to the prompt construction described in our framework, the prompt is conceptually composed of:

$$p = p_{\text{task}} \oplus p_{\text{metric}} \oplus p_{\text{requirement}} \oplus p_{\text{dep}} \oplus p_{\text{output}}. \quad (11)$$

Here, p_{dep} contains both the feature-wise structural summaries and the complete label tree structure. The component symbols are provided only for explanation and are not inserted into the original prompt text.

Prompt Part I: Task Specification and Evaluation Criteria ($p_{\text{task}} \oplus p_{\text{metric}}$)

```

1 Your task is to generate a tabular data synthesis strategy by inferring the
2 original data's intrinsic characteristics from the provided pre-trained
3 decision trees and feature information. The synthesized data will replace the
4 original data for model training, with the core goal of making the performance
5 of models trained on synthetic data as close as possible to those trained on
6 the original data.
7
8 # Metrics
9 When evaluating synthetic data, typically considering the following metrics:
10
11 1. ML Efficiency:
12    A core metric reflecting synthetic data's support for downstream ML tasks,
13    evaluated via the TSTR (Train on Synthetic, Test on Real) protocol.
14
15 2. Fidelity:
16    Measures preservation of raw data statistical properties from two aspects:
17    - Pairwise: Uses DPCM (numerical features), DCSM (categorical features),
18      and contingency similarity matrices (mixed features) to assess
19      feature-pair dependencies.
20    - Column-wise: Compares individual features; KST is commonly used for
21      numerical features.
22
23 3. Privacy:
24    Privacy is evaluated using split-ratio-adjusted DCR risk, which measures
25    whether synthetic records exhibit excessive proximity to training records
26    beyond the expected preference induced by the train--test split ratio.
27    A lower adjusted DCR value indicates lower potential memorization risk.
28
29 The synthetic data you generate should perform well on these metrics.

```

Prompt Part II: Core Requirements for Strategy Generation ($p_{\text{requirement}}$)

```

1 # Core Requirements for Your Strategy
2
3 Multiple decision tree models are provided. The final decision tree takes the
4 original label of the raw data as the prediction target. All preceding decision
5 trees are trained with a single data feature as the target and all other
6 features, excluding the original label, as inputs. Their outputs are restricted
7 to structured summaries, for example:
8
9 {feature_0}:{type:continuous,min:0.0,max:1.0,
10 associated_features:[feature_1,feature_3,feature_89],
11 performance:{R2:0.992,RMSE:0.0379}}
12
13 The features listed in the associated_features field participate in node
14 splitting in the decision tree trained to predict the corresponding target
15 feature. Their order follows the top-down splitting order of the original tree.
16
17 You need to analyse the underlying feature dependency patterns and distribution
18 characteristics based on the provided decision tree information. You should
19 then construct a routing function to assign each sample to an appropriate
20 subspace. The routing function should reflect your interpretation of the
21 data structure rather than merely reproducing tree rules.
22
23 Within each routed subspace, use suitable model fitting procedures to learn the
24 distribution characteristics of real data and support the generation of
25 high-quality synthetic samples.
26
27 Attention:
28 Decision tree rules are not intended for direct record sampling. They should be
29 analysed to infer feature dependencies and data distribution characteristics.
30 Your output must incorporate this analysis when designing the synthesis
31 procedure.

```

Prompt Part III: Feature inter-dependency for *Adult* (p_{dep})

```

1
2 {feature_0}:{type:continuous,min:17.0,max:90.0,

```

```

3 associated_features:[feature_5,feature_12,feature_7,feature_1,feature_6,feature_4,feature_10
4 ],
5 performance:{R2:0.4227,RMSE:10.3665}}
6 {feature_1}:{type:categorical,ordinal_encoding_range:[0,8],class_num:9,
7 associated_features:[feature_6,feature_0,feature_4,feature_12],
8 performance:{accuracy:0.7553}}
9
10 {feature_2}:{type:continuous,min:12285.0,max:1484705.0,
11 associated_features:[feature_13,feature_8,feature_0,feature_9,feature_1,feature_6,feature_12
12 ],
13 performance:{R2:0.0549,RMSE:102680.6073}}
14
15 {feature_3}:{type:categorical,ordinal_encoding_range:[0,15],class_num:16,
16 associated_features:[feature_4],
17 performance:{accuracy:1.0}}
18
19 {feature_4}:{type:continuous,min:1.0,max:16.0,
20 associated_features:[feature_3],
21 performance:{R2:1.0,RMSE:0.0}}
22
23 {feature_5}:{type:categorical,ordinal_encoding_range:[0,6],class_num:7,
24 associated_features:[feature_7,feature_0,feature_9,feature_4,feature_8,feature_13],
25 performance:{accuracy:0.8419}}
26
27 {feature_6}:{type:categorical,ordinal_encoding_range:[0,14],class_num:15,
28 associated_features:[feature_1,feature_4,feature_9,feature_0,feature_12],
29 performance:{accuracy:0.3717}}
30
31 {feature_7}:{type:categorical,ordinal_encoding_range:[0,5],class_num:6,
32 associated_features:[feature_5,feature_9,feature_0,feature_12,feature_4,feature_1,feature_13,
33 feature_8],
34 performance:{accuracy:0.7891}}
35
36 {feature_8}:{type:categorical,ordinal_encoding_range:[0,4],class_num:5,
37 associated_features:[feature_13,feature_7,feature_12,feature_2,feature_5],
38 performance:{accuracy:0.8752}}
39
40 {feature_9}:{type:categorical,ordinal_encoding_range:[0,1],class_num:2,
41 associated_features:[feature_7,feature_6,feature_1,feature_5,feature_12],
42 performance:{accuracy:0.8336}}
43
44 {feature_10}:{type:continuous,min:0.0,max:99999.0,
45 associated_features:[feature_4,feature_0,feature_12,feature_11,feature_2,feature_3,feature_8,
46 feature_7,feature_5],
47 performance:{R2:0.1142,RMSE:6927.3273}}
48
49 {feature_11}:{type:continuous,min:0.0,max:4356.0,
50 associated_features:[feature_4,feature_7,feature_10,feature_5,feature_2,feature_0,feature_12,
51 feature_3,feature_6],
52 performance:{R2:0.0354,RMSE:394.4608}}
53
54 {feature_12}:{type:continuous,min:1.0,max:99.0,
55 associated_features:[feature_0,feature_3,feature_4,feature_9,feature_10,feature_1,feature_7,
56 feature_6],
57 performance:{R2:0.2214,RMSE:10.9254}}
58
59 {feature_13}:{type:categorical,ordinal_encoding_range:[0,41],class_num:42,
60 associated_features:[feature_8,feature_2,feature_4,feature_0,feature_1],
61 performance:{accuracy:0.9031}}

```

205

Prompt Part IV: Complete Label-Tree Structure (p_{dep})

```

1 feature_14 (label) (categorical) ordinal encoding range is integers in [0, 1]:
2 accuracy: 0.8545
3
4 --- feature_7 <= 0.5
5 |   --- feature_4 <= 12.5
6 |   |   --- feature_10 <= 5095.5
7 |   |   |   --- feature_11 <= 1782.5
8 |   |   |   |   --- feature_4 <= 8.5
9 |   |   |   |   |   --- class: 0 [leaf_id:1]
10 |   |   |   |   |   Sample Count 1247, Label Ratio {'0': 0.901, '1': 0.099}
11 |   |   |   |   |   --- feature_4 > 8.5
12 |   |   |   |   |   |   --- feature_0 <= 35.5
13 |   |   |   |   |   |   |   --- class: 0 [leaf_id:2]
14 |   |   |   |   |   |   |   Sample Count 2042, Label Ratio {'0': 0.800, '1': 0.200}

```

206

```

15 | | | | | --- feature_0 > 35.5
16 | | | | | --- feature_12 <= 34.5
17 | | | | | | --- class: 0 [leaf_id:3]
18 | | | | | | Sample Count 376, Label Ratio {'0': 0.899, '1': 0.101}
19 | | | | | --- feature_12 > 34.5
20 | | | | | | --- feature_4 <= 9.5
21 | | | | | | | --- class: 0 [leaf_id:4]
22 | | | | | | | Sample Count 2136, Label Ratio {'0': 0.659, '1': 0.341}
23 | | | | | | --- feature_4 > 9.5
24 | | | | | | | --- class: 0 [leaf_id:5]
25 | | | | | | | Sample Count 1716, Label Ratio {'0': 0.538, '1': 0.462}
26 | | | | | --- feature_11 > 1782.5
27 | | | | | | --- feature_11 <= 1989.5
28 | | | | | | | --- class: 1 [leaf_id:6]
29 | | | | | | | Sample Count 209, Label Ratio {'0': 0.053, '1': 0.947}
30 | | | | | | --- feature_11 > 1989.5
31 | | | | | | | --- class: 0 [leaf_id:7]
32 | | | | | | | Sample Count 82, Label Ratio {'0': 0.829, '1': 0.171}
33 | | | | | --- feature_10 > 5095.5
34 | | | | | | --- class: 1 [leaf_id:8]
35 | | | | | | Sample Count 403, Label Ratio {'0': 0.020, '1': 0.980}
36 | | | | | --- feature_4 > 12.5
37 | | | | | --- feature_10 <= 5095.5
38 | | | | | | --- feature_11 <= 1782.5
39 | | | | | | | --- feature_12 <= 31.0
40 | | | | | | | | --- class: 0 [leaf_id:9]
41 | | | | | | | | Sample Count 207, Label Ratio {'0': 0.691, '1': 0.309}
42 | | | | | | | --- feature_12 > 31.0
43 | | | | | | | | --- feature_0 <= 28.5
44 | | | | | | | | | --- class: 0 [leaf_id:10]
45 | | | | | | | | | Sample Count 149, Label Ratio {'0': 0.624, '1': 0.376}
46 | | | | | | | | --- feature_0 > 28.5
47 | | | | | | | | | --- class: 1 [leaf_id:11]
48 | | | | | | | | | Sample Count 2321, Label Ratio {'0': 0.318, '1': 0.682}
49 | | | | | | | --- feature_11 > 1782.5
50 | | | | | | | | --- class: 1 [leaf_id:12]
51 | | | | | | | | Sample Count 309, Label Ratio {'0': 0.036, '1': 0.964}
52 | | | | | | | --- feature_10 > 5095.5
53 | | | | | | | | --- class: 1 [leaf_id:13]
54 | | | | | | | | Sample Count 533, Label Ratio {'0': 0.004, '1': 0.996}
55 | | | | | --- feature_7 > 0.5
56 | | | | | --- feature_10 <= 7073.5
57 | | | | | | --- feature_7 <= 4.5
58 | | | | | | | --- feature_4 <= 13.5
59 | | | | | | | | --- feature_12 <= 42.5
60 | | | | | | | | | --- class: 0 [leaf_id:14]
61 | | | | | | | | | Sample Count 11927, Label Ratio {'0': 0.979, '1': 0.021}
62 | | | | | | | | --- feature_12 > 42.5
63 | | | | | | | | | --- class: 0 [leaf_id:15]
64 | | | | | | | | | Sample Count 2793, Label Ratio {'0': 0.890, '1': 0.110}
65 | | | | | | | --- feature_4 > 13.5
66 | | | | | | | | --- feature_12 <= 43.5
67 | | | | | | | | | --- class: 0 [leaf_id:16]
68 | | | | | | | | | Sample Count 494, Label Ratio {'0': 0.836, '1': 0.164}
69 | | | | | | | | --- feature_12 > 43.5
70 | | | | | | | | | --- class: 0 [leaf_id:17]
71 | | | | | | | | | Sample Count 318, Label Ratio {'0': 0.594, '1': 0.406}
72 | | | | | | | --- feature_7 > 4.5
73 | | | | | | | | --- feature_4 <= 10.5
74 | | | | | | | | | --- class: 0 [leaf_id:18]
75 | | | | | | | | | Sample Count 784, Label Ratio {'0': 0.684, '1': 0.316}
76 | | | | | | | | --- feature_4 > 10.5
77 | | | | | | | | | --- class: 1 [leaf_id:19]
78 | | | | | | | | | Sample Count 503, Label Ratio {'0': 0.364, '1': 0.636}
79 | | | | | | | --- feature_10 > 7073.5
80 | | | | | | | | --- class: 1 [leaf_id:20]
81 | | | | | | | | Sample Count 395, Label Ratio {'0': 0.033, '1': 0.967}

```

207

Prompt Part V: Output and Implementation Constraints

```

1 # Output Requirements
2
3 Please carefully examine the decision tree rules and adhere to the core
4 requirements.
5
6 You must write a complete and executable function to generate samples covering
7 all features. The resulting synthetic data will replace the original dataset.

```

208

```

8
9 Additional requirements:
10 1. The generated synthetic data must be saved in CSV format.
11 2. Only output the function generate_synthesis_data.
12 3. All library imports and auxiliary functions must be contained within
13    generate_synthesis_data.
14 4. Categorical features must use ordinal encoding and be represented directly
15    as integers rather than strings.
16 5. If probabilities are used, they must sum to 1.
17 6. Generated values must not exceed the predefined feature ranges.
18 7. You may select different distribution-fitting models for different feature
19    types or subspaces, provided that feature dependencies are appropriately
20    considered.
21
22 # Output Format
23
24 def generate_synthesis_data(save_path: str, n_sample: int,
25                             X_train: np.ndarray, y_train: np.ndarray) -> None:
26     import pandas as pd
27     import csv
28     ...
29
30     fieldnames = [f"feature_{i}" for i in range(14)] + ["label"]
31     with open(save_path, "w", newline="") as csv_file:
32         writer = csv.DictWriter(csv_file, fieldnames=fieldnames)
33         writer.writeheader()
34         writer.writerows(synthesis_samples)

```

209

Prompt Part VI: Reference Analysis Provided to the LLM (p_{output})

```

1 # Reference Analysis
2
3 An analysis of the provided decision tree rules reveals a complex web of
4 inter-feature dependencies. The goal is to construct a generative process that
5 honors these relationships to produce high-fidelity synthetic data. The strategy
6 is founded on the principle of divide and conquer: partitioning the data space
7 into smaller, more homogeneous subspaces and then modelling the distribution
8 within each subspace.
9
10 1. Feature Dependency Analysis
11 - Hub Features: feature_4, feature_7, and feature_0 appear frequently as
12   splitting variables, indicating that they are central to the data structure.
13 - Strong Co-dependency: feature_3 and feature_4 exhibit a near-deterministic
14   relationship. Once feature_4 is generated, feature_3 can be generated through
15   an explicit mapping.
16 - Cyclic Dependencies: dependencies such as feature_5 relying on feature_7 and
17   feature_7 relying on feature_5 imply that the generation strategy must adopt
18   a practical ordering to break cycles.
19
20 2. Subspace-based Synthesis Strategy
21 - Partition the data using major task-relevant attributes identified from the
22   label tree, especially feature_7 and feature_4.
23 - Fit conditional generative models within each subspace rather than learning
24   one global distribution.
25 - Use probabilistic classification models for categorical attributes.
26 - Use regression models with residual sampling for continuous attributes.
27 - Explicitly preserve the deterministic mapping from feature_4 to feature_3.
28
29 3. Data Generation and Validation
30 - Allocate generated samples according to the empirical sizes of routed
31   subspaces.
32 - Aggregate all generated records into a complete synthetic dataset.
33 - Enforce feature type, range, and encoding constraints before saving the CSV.

```

210

211 D.2 LLM-generated analysis and synthesis strategy

Listing 1 Original LLM output containing the analysis and generated synthesis strategy for the *Adult* dataset.

```

212 To synthesize the data effectively, I will follow a "Divide and Conquer" approach based on the
213 provided decision tree rules.
214
215 ### Analysis of Decision Tree Rules and Feature Dependencies
216
217 1. **Hub Features and Partitioning:**
218

```

218

```

219 * **Feature 7:** This is the root of the label decision tree. It also appears as a primary
220 split for features 5 and 9. This is the most natural candidate for the primary routing
221 split.
222 * **Feature 4 and Feature 3:** These features have a perfect deterministic relationship (
223  $\$R^2=1$ , Accuracy=1). They are essentially two representations of the same information. In
224 the label tree, feature 4 is a critical secondary splitter.
225 * **Feature 10 and Feature 11:** These features show high sparsity (min=0) and appear as
226 important split features in the label tree.
227
228 2. **Dependency Chain:**
229 * **Root Features:** feature 0, feature 9, and feature 13 tend to be independent or act as
230 root nodes.
231 * **Related Features:** feature 4 influences feature 1, feature 6, and feature 12.
232 * **Target:** Label is heavily conditioned on feature 7, feature 4, features 10 and
233 feature 11.
234
235 ### Synthesis Strategy
236
237 1. **Routing Function:** I will partition the data into 4 major subspaces based on the primary
238 splits of the label tree: feature 7 ( $\leq 0.5$  vs  $> 0.5$ ) and feature 4 ( $\leq 12.5$  vs  $> 12.5$ ).
239 This ensures that the most important decision boundaries for ML Efficiency are preserved.
240 2. **Generative Chain within Subspaces:**
241 * **Step 1:** Fit and sample "Root Features" (feature 7, feature 4, feature 0) using their
242 empirical joint distribution in the subspace.
243 * **Step 2:** Generate feature 3 via a direct mapping from feature 4.
244 * **Step 3:** Generate remaining categorical features (feature 1, feature 5, feature 6,
245 feature 8, feature 9, feature 13) using DecisionTreeClassifier models conditioned on the
246 root features. To maintain fidelity, I will sample from the predicted probability
247 distributions rather than taking the argmax.
248 * **Step 4:** Generate continuous features (feature 2, feature 10, feature 11, feature 12)
249 using DecisionTreeRegressor models. To preserve variance and privacy, I will add sampled
250 residuals from the training data to the model predictions.
251 * **Step 5:** Generate the label using a DecisionTreeClassifier conditioned on all
252 significant predictors identified in the rules.
253
254 3. **Refining Constraints:** I will strictly enforce the min and max values provided in the
255 rules for continuous features and use ordinal_encoding_range for categorical features.
256
257 '''python
258 import numpy as np
259 import pandas as pd
260 from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
261 import csv
262
263 def generate_synthesis_data(save_path: str, n_sample: int, X_train: np.ndarray, y_train: np.
264 ndarray) -> None:
265     # Combine X and y for training modeling
266     full_data = np.column_stack((X_train, y_train))
267     feat_cols = ["feature_{i}" for i in range(14)]
268     all_cols = feat_cols + ["label"]
269     df = pd.DataFrame(full_data, columns=all_cols)
270
271     # Feature metadata derived from rules
272     continuous_features = {
273         'feature_0': (17.0, 90.0),
274         'feature_2': (12285.0, 1484705.0),
275         'feature_4': (1.0, 16.0),
276         'feature_10': (0.0, 99999.0),
277         'feature_11': (0.0, 4356.0),
278         'feature_12': (1.0, 99.0)
279     }
280     categorical_features = {
281         'feature_1': 9, 'feature_3': 16, 'feature_5': 7, 'feature_6': 15,
282         'feature_7': 6, 'feature_8': 5, 'feature_9': 2, 'feature_13': 42,
283         'label': 2
284     }
285
286     # Define subspaces based on the label decision tree splits
287     df['subspace'] = 0
288     df.loc[(df['feature_7'] <= 0.5) & (df['feature_4'] <= 12.5), 'subspace'] = 0
289     df.loc[(df['feature_7'] <= 0.5) & (df['feature_4'] > 12.5), 'subspace'] = 1
290     df.loc[(df['feature_7'] > 0.5) & (df['feature_4'] <= 12.5), 'subspace'] = 2
291     df.loc[(df['feature_7'] > 0.5) & (df['feature_4'] > 12.5), 'subspace'] = 3
292
293     subspace_counts = df['subspace'].value_counts(normalize=True)
294
295     synthetic_data = []
296
297     for sub_id in range(4):
298         sub_df = df[df['subspace'] == sub_id]

```

```

299         if len(sub_df) < 5: continue # Safety for empty subspaces
300
301         n_sub_samples = int(n_sample * subspace_counts.get(sub_id, 0))
302         if n_sub_samples <= 0: continue
303
304         # Local Models for this subspace
305         # Order: 7, 4, 0 -> 3 -> 9, 5 -> 1, 6, 12 -> 10, 11 -> 2, 8, 13 -> label
306
307         # 1. Roots: 7, 4, 0 (Sampled directly to maintain core structure)
308         indices = np.random.choice(sub_df.index, size=n_sub_samples, replace=True)
309         roots = sub_df.loc[indices, ['feature_7', 'feature_4', 'feature_0']].values
310
311         sub_gen = pd.DataFrame(roots, columns=['feature_7', 'feature_4', 'feature_0'])
312
313         # 2. Map 3 from 4 (Deterministic rule)
314         # We learn the most frequent mapping in this subspace
315         map_3_4 = sub_df.groupby('feature_4')['feature_3'].agg(lambda x: x.value_counts().index
316 [0]).to_dict()
317         sub_gen['feature_3'] = sub_gen['feature_4'].map(map_3_4).fillna(sub_df['feature_3'].mode
318 ([0]))
319
320         # 3. Conditional Generators
321         def fit_sample_cat(target, predictors, current_gen_df):
322             clf = DecisionTreeClassifier(max_depth=5)
323             clf.fit(sub_df[predictors], sub_df[target])
324             probs = clf.predict_proba(current_gen_df[predictors])
325             # Sample from probabilities
326             return [np.random.choice(clf.classes_, p=p) for p in probs]
327
328         def fit_sample_cont(target, predictors, current_gen_df):
329             reg = DecisionTreeRegressor(max_depth=5)
330             reg.fit(sub_df[predictors], sub_df[target])
331             preds = reg.predict(current_gen_df[predictors])
332             # Add residuals
333             residuals = sub_df[target] - reg.predict(sub_df[predictors])
334             sampled_res = np.random.choice(residuals, size=len(preds), replace=True)
335             res = preds + sampled_res
336             # Clip to range
337             low, high = continuous_features[target]
338             return np.clip(res, low, high)
339
340         # Sequence of generation
341         sub_gen['feature_9'] = fit_sample_cat('feature_9', ['feature_7', 'feature_0'], sub_gen)
342         sub_gen['feature_5'] = fit_sample_cat('feature_5', ['feature_7', 'feature_0', 'feature_9
343 '], sub_gen)
344         sub_gen['feature_1'] = fit_sample_cat('feature_1', ['feature_4', 'feature_0', 'feature_5
345 '], sub_gen)
346         sub_gen['feature_6'] = fit_sample_cat('feature_6', ['feature_1', 'feature_4', 'feature_9
347 '], sub_gen)
348         sub_gen['feature_12'] = fit_sample_cont('feature_12', ['feature_0', 'feature_4', '
349 feature_6'], sub_gen)
350         sub_gen['feature_10'] = fit_sample_cont('feature_10', ['feature_4', 'feature_0', '
351 feature_12'], sub_gen)
352         sub_gen['feature_11'] = fit_sample_cont('feature_11', ['feature_4', 'feature_7', '
353 feature_10'], sub_gen)
354         sub_gen['feature_2'] = fit_sample_cont('feature_2', ['feature_0', 'feature_7', '
355 feature_12'], sub_gen)
356         sub_gen['feature_8'] = fit_sample_cat('feature_8', ['feature_7', 'feature_9', 'feature_2
357 '], sub_gen)
358         sub_gen['feature_13'] = fit_sample_cat('feature_13', ['feature_8', 'feature_2', '
359 feature_0'], sub_gen)
360         sub_gen['label'] = fit_sample_cat('label', ['feature_7', 'feature_4', 'feature_10', '
361 feature_11', 'feature_0', 'feature_12'], sub_gen)
362
363         synthetic_data.append(sub_gen)
364
365     # Combine and shuffle
366     final_df = pd.concat(synthetic_data).sample(frac=1).reset_index(drop=True)
367
368     # Final data type correction
369     for col in categorical_features:
370         final_df[col] = final_df[col].astype(int)
371
372     # Reorder columns to match fieldnames
373     final_df = final_df[feat_cols + ["label"]]
374
375     # Save to CSV
376     final_df.to_csv(save_path, index=False)
377
378

```

Supplementary Note E Ablation results

The ablation experiments are conducted on eight datasets for which all evaluated variants can be accommodated within the LLM context window. In particular, the Full Feature Trees variant retains the complete feature-tree structures and therefore produces substantially longer prompts; for the remaining high-dimensional datasets, these prompts exceed the context-window limit. We therefore restrict the ablation analysis to datasets on which all variants can be evaluated under the same experimental protocol, ensuring a consistent and fair comparison. Across the following ablations, we focus on overall performance trends across datasets rather than requiring each design choice to improve every individual metric on every dataset.

E.1 Contribution of feature-dependence extraction

This section reports detailed ablation results on feature-dependence extraction. We compare the proposed tree-based extraction mechanism with two alternative mechanisms based on linear models and mutual information. To isolate the effect of feature-wise dependence extraction, the complete label tree and all other prompt components are kept unchanged across all variants. Across the evaluated datasets, the chained-tree mechanism provides a stronger overall balance of marginal fidelity, pairwise-dependence preservation, and downstream task performance than the alternative extraction mechanisms, although the relative advantage varies across individual datasets and metrics. Detailed per-dataset results are reported in Supplementary Table 8.

Supplementary Table 8 Ablation results for feature-dependence extraction mechanisms. Values are reported as mean $\pm$ s.d. Bold values indicate the best result for each dataset.

Dataset	Linear Models			Mutual Information			Chained Trees (Ours)		
	Marginal distribution error ($\downarrow$)	Pairwise correlation error ($\downarrow$)	Task ^a	Marginal distribution error ($\downarrow$)	Pairwise correlation error ($\downarrow$)	Task ^a	Marginal distribution error ($\downarrow$)	Pairwise correlation error ($\downarrow$)	Task ^a
<i>Adult</i>	12.31% $\pm$ 0.000	9.78% $\pm$ 0.000	0.903$\pm$0.001	9.93% $\pm$ 0.031	11.90% $\pm$ 0.000	0.892 $\pm$ 0.002	5.44%$\pm$0.001	3.53%$\pm$0.000	0.900 $\pm$ 0.001
<i>Blood</i>	8.64% $\pm$ 0.000	2.91% $\pm$ 0.000	0.712$\pm$0.000	8.38% $\pm$ 0.004	3.21% $\pm$ 0.004	0.699 $\pm$ 0.055	3.32%$\pm$0.005	2.68%$\pm$0.004	0.704 $\pm$ 0.026
<i>Car</i>	8.67% $\pm$ 0.003	17.70% $\pm$ 0.003	0.969$\pm$0.004	1.64%$\pm$0.003	3.84%$\pm$0.003	0.961 $\pm$ 0.006	1.68% $\pm$ 0.004	3.99% $\pm$ 0.004	0.958 $\pm$ 0.006
<i>Default</i>	10.57% $\pm$ 0.001	4.74%$\pm$0.001	0.751 $\pm$ 0.004	8.48% $\pm$ 0.000	5.72% $\pm$ 0.000	0.735 $\pm$ 0.005	1.00%$\pm$0.001	5.09% $\pm$ 0.001	0.752$\pm$0.002
<i>Diamonds</i>	3.50%$\pm$0.001	3.18% $\pm$ 0.000	1287.506 $\pm$ 125.968	4.29% $\pm$ 0.000	3.58% $\pm$ 0.000	1262.624 $\pm$ 13.715	3.79% $\pm$ 0.000	2.11%$\pm$0.000	934.090$\pm$15.111
<i>Heart</i>	5.94%$\pm$0.004	11.17% $\pm$ 0.004	0.783 $\pm$ 0.029	8.42% $\pm$ 0.007	13.96% $\pm$ 0.008	0.852$\pm$0.019	6.08% $\pm$ 0.007	1.11%$\pm$0.006	0.823 $\pm$ 0.028
<i>Liver</i>	14.94% $\pm$ 0.000	5.21% $\pm$ 0.000	3.291 $\pm$ 0.000	11.62% $\pm$ 0.007	3.15%$\pm$0.006	3.421 $\pm$ 0.088	9.20%$\pm$0.000	3.49% $\pm$ 0.000	3.181$\pm$0.000
<i>Magic</i>	3.02%$\pm$0.000	0.98%$\pm$0.000	0.920$\pm$0.000	3.57% $\pm$ 0.000	1.08% $\pm$ 0.000	0.918 $\pm$ 0.001	3.09% $\pm$ 0.000	0.98%$\pm$0.000	0.920$\pm$0.000

Task^a is AUROC ($\uparrow$) for classification and RMSE ($\downarrow$) for regression. The complete label tree and all other prompt components are retained across all variants.

E.2 Effect of dependence information encoding

This section examines how the representation and ordering of dependence information affect synthesis performance. We compare the proposed encoding with three variants: retaining the full feature-tree structures, replacing the complete label tree with a compact summary, and randomly shuffling the order of associated features. Overall, the proposed representation provides a favourable balance across the evaluated metrics and datasets, while individual variants can perform better on particular datasets or metrics. These results indicate that compact representation and structured ordering are useful at the aggregate level rather than uniformly improving every individual evaluation measure. The corresponding results are reported in Supplementary Table 9.

Supplementary Table 9 Ablation results for dependence-information representation and ordering. Values are reported as mean $\pm$ s.d. Bold values indicate the best result for each dataset.

Dataset	Full Feature Trees			Condensed Label Tree			Shuffled Dependence Order			Proposed Representation (Ours)		
	Marginal distribution error ($\downarrow$)	Pairwise correlation error ($\downarrow$)	Task ^a	Marginal distribution error ($\downarrow$)	Pairwise correlation error ($\downarrow$)	Task ^a	Marginal distribution error ($\downarrow$)	Pairwise correlation error ($\downarrow$)	Task ^a	Marginal distribution error ($\downarrow$)	Pairwise correlation error ($\downarrow$)	Task ^a
<i>Adult</i>	6.21% $\pm$ 0.000	4.65% $\pm$ 0.000	0.897 $\pm$ 0.002	3.14%$\pm$0.000	5.99% $\pm$ 0.000	0.887 $\pm$ 0.003	9.09% $\pm$ 0.000	5.53% $\pm$ 0.000	0.907$\pm$0.001	5.44% $\pm$ 0.001	3.53%$\pm$0.000	0.900 $\pm$ 0.001
<i>Blood</i>	3.77% $\pm$ 0.006	4.02% $\pm$ 0.007	0.696 $\pm$ 0.035	5.92% $\pm$ 0.006	3.53% $\pm$ 0.005	0.636 $\pm$ 0.050	7.99% $\pm$ 0.007	3.39% $\pm$ 0.003	0.713$\pm$0.033	3.32%$\pm$0.005	2.68%$\pm$0.004	0.704 $\pm$ 0.026
<i>Car</i>	4.21% $\pm$ 0.000	7.37% $\pm$ 0.000	0.952 $\pm$ 0.000	1.68% $\pm$ 0.003	4.47% $\pm$ 0.003	0.943 $\pm$ 0.013	1.29%$\pm$0.002	3.44%$\pm$0.002	0.964$\pm$0.005	1.68% $\pm$ 0.004	3.99% $\pm$ 0.004	0.958 $\pm$ 0.006
<i>Default</i>	9.25% $\pm$ 0.001	5.48% $\pm$ 0.001	0.756$\pm$0.003	11.78% $\pm$ 0.000	9.54% $\pm$ 0.000	0.730 $\pm$ 0.000	9.99% $\pm$ 0.000	7.37% $\pm$ 0.000	0.750 $\pm$ 0.000	1.00%$\pm$0.001	5.09%$\pm$0.001	0.752 $\pm$ 0.002
<i>Diamonds</i>	1.40%$\pm$0.000	2.18% $\pm$ 0.000	1419.174 $\pm$ 5.703	3.92% $\pm$ 0.000	2.31% $\pm$ 0.000	1477.964 $\pm$ 13.909	2.36% $\pm$ 0.000	2.46% $\pm$ 0.000	1406.246 $\pm$ 5.114	3.79% $\pm$ 0.000	2.11%$\pm$0.000	934.090$\pm$15.111
<i>Heart</i>	5.46% $\pm$ 0.004	10.76% $\pm$ 0.004	0.845 $\pm$ 0.016	5.67% $\pm$ 0.005	10.77% $\pm$ 0.004	0.853$\pm$0.013	5.42%$\pm$0.004	10.08%$\pm$0.004	0.837 $\pm$ 0.033	6.08% $\pm$ 0.007	11.11% $\pm$ 0.006	0.823 $\pm$ 0.028
<i>Liver</i>	9.26% $\pm$ 0.010	3.32% $\pm$ 0.008	3.550 $\pm$ 0.119	8.83%$\pm$0.006	3.77% $\pm$ 0.007	3.448 $\pm$ 0.185	10.30% $\pm$ 0.000	2.16%$\pm$0.000	3.378 $\pm$ 0.000	9.20% $\pm$ 0.000	3.49% $\pm$ 0.000	3.181$\pm$0.000
<i>Magic</i>	2.98% $\pm$ 0.000	0.96%$\pm$0.000	0.918 $\pm$ 0.000	3.41% $\pm$ 0.000	0.96%$\pm$0.000	0.917 $\pm$ 0.001	2.95%$\pm$0.001	2.12% $\pm$ 0.001	0.914 $\pm$ 0.002	3.09% $\pm$ 0.000	0.98% $\pm$ 0.000	0.920$\pm$0.000

Task^a is AUROC ($\uparrow$) for classification and RMSE ($\downarrow$) for regression. The complete label tree is retained in all variants except the Condensed Label Tree variant.

E.3 Contribution of prompt guidance

We further examine whether the prompt components used to guide LLM reasoning contribute to synthesis quality. In this experiment, the task definition and the feature-dependence information extracted from the chained trees are retained in all variants, while metric guidance, routing guidance, or the reference analysis is removed individually. Across datasets, the full prompt provides a more balanced overall synthesis

performance than the ablated variants, although removing an individual component can occasionally improve a particular metric on a particular dataset. The ablation therefore supports the overall contribution of the prompt-guidance components without implying that each component improves every metric individually. Detailed results are reported in Supplementary Table 10.

Supplementary Table 10 Ablation results for prompt-guidance components. Values are reported as mean $\pm$ s.d. Bold values indicate the best result for each dataset.

Dataset	w/o Routing			w/o Metric			w/o Reference			Full Prompt (Ours)		
	Marginal distribution error ($\downarrow$)	Pairwise correlation error ($\downarrow$)	Task ^a	Marginal distribution error ($\downarrow$)	Pairwise correlation error ($\downarrow$)	Task ^a	Marginal distribution error ($\downarrow$)	Pairwise correlation error ($\downarrow$)	Task ^a	Marginal distribution error ($\downarrow$)	Pairwise correlation error ($\downarrow$)	Task ^a
<i>Adult</i>	5.85% $\pm$ 0.054	5.07% $\pm$ 0.038	0.904 $\pm$ 0.001	8.29% $\pm$ 0.064	4.70% $\pm$ 0.064	0.905$\pm$0.001	11.92% $\pm$ 0.059	10.82% $\pm$ 0.052	0.888 $\pm$ 0.002	5.44%$\pm$0.001	3.53%$\pm$0.000	0.900 $\pm$ 0.001
<i>Blood</i>	6.97% $\pm$ 1.180	3.94% $\pm$ 0.493	0.702 $\pm$ 0.034	8.14% $\pm$ 0.146	3.50% $\pm$ 0.232	0.713$\pm$0.054	8.83% $\pm$ 0.404	3.78% $\pm$ 0.597	0.700 $\pm$ 0.075	3.32%$\pm$0.005	2.68%$\pm$0.004	0.704 $\pm$ 0.026
<i>Car</i>	1.70% $\pm$ 0.380	3.92% $\pm$ 0.403	0.915 $\pm$ 0.017	1.37%$\pm$0.224	3.40%$\pm$0.292	0.974$\pm$0.005	1.68% $\pm$ 0.382	3.92% $\pm$ 0.320	0.963 $\pm$ 0.004	1.68% $\pm$ 0.004	3.99% $\pm$ 0.004	0.958 $\pm$ 0.006
<i>Default</i>	9.58% $\pm$ 0.091	3.54%$\pm$0.046	0.756$\pm$0.003	11.67% $\pm$ 0.078	5.89% $\pm$ 0.079	0.751 $\pm$ 0.003	13.89% $\pm$ 0.041	10.42% $\pm$ 0.028	0.723 $\pm$ 0.004	1.00%$\pm$0.001	5.09% $\pm$ 0.001	0.752 $\pm$ 0.002
<i>Diamonds</i>	4.91% $\pm$ 0.000	2.97% $\pm$ 0.000	889.692$\pm$6.476	4.64% $\pm$ 0.000	2.51% $\pm$ 0.000	1195.031 $\pm$ 0.000	6.17% $\pm$ 0.000	5.06% $\pm$ 0.000	1011.838 $\pm$ 0.000	3.79%$\pm$0.000	2.11%$\pm$0.000	934.090 $\pm$ 15.111
<i>Heart</i>	7.25% $\pm$ 1.046	12.57% $\pm$ 0.864	0.817 $\pm$ 0.031	6.23% $\pm$ 0.873	11.21% $\pm$ 0.808	0.817 $\pm$ 0.029	5.70%$\pm$0.462	11.08%$\pm$0.609	0.849$\pm$0.022	6.08% $\pm$ 0.007	11.11% $\pm$ 0.006	0.823 $\pm$ 0.028
<i>Liver</i>	14.19% $\pm$ 0.729	4.84% $\pm$ 1.205	3.413 $\pm$ 0.085	9.67% $\pm$ 1.019	2.59% $\pm$ 0.357	3.484 $\pm$ 0.086	13.34% $\pm$ 0.946	2.55%$\pm$0.482	3.546 $\pm$ 0.103	9.20%$\pm$0.000	3.49% $\pm$ 0.000	3.181$\pm$0.000
<i>Magic</i>	3.29% $\pm$ 0.026	1.19% $\pm$ 0.033	0.915 $\pm$ 0.002	3.39% $\pm$ 0.000	0.98% $\pm$ 0.000	0.921$\pm$0.000	2.75%$\pm$0.011	0.84%$\pm$0.007	0.898 $\pm$ 0.002	3.09% $\pm$ 0.000	0.98% $\pm$ 0.000	0.920 $\pm$ 0.000

Task^a is AUROC ($\uparrow$) for classification and RMSE ($\downarrow$) for regression. The task definition and tree-derived dependence information are retained in all variants.

E.4 Sensitivity to different LLM backbones

We evaluate the sensitivity of the proposed framework to the choice of LLM backbone. We replace Gemini-2.5-Pro [51] with GLM-5 [55] and Qwen3-Max [56], while retaining the same prompt, number of candidate queries, and strategy-selection protocol. The results show that the overall TabSSD pipeline remains applicable across all three evaluated LLM backbones, indicating that the framework is not restricted to a single specific model. However, synthesis performance varies across backbones and datasets, suggesting that the choice of backbone can still affect the quality of the generated strategies. Detailed results are reported in Supplementary Table 11.

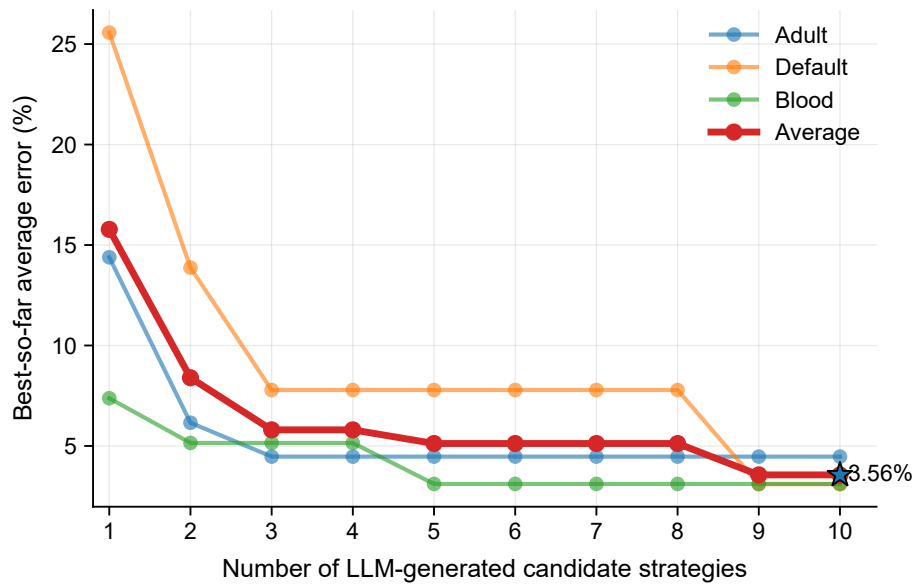
Supplementary Table 11 Sensitivity of TabSSD to different LLM backbones. Values are reported as mean $\pm$ s.d. Bold values indicate the best result for each dataset.

Dataset	GLM-5			Qwen3-Max			Gemini-2.5-Pro		
	Marginal distribution error ($\downarrow$)	Pairwise correlation error ($\downarrow$)	Task ^a	Marginal distribution error ($\downarrow$)	Pairwise correlation error ($\downarrow$)	Task ^a	Marginal distribution error ($\downarrow$)	Pairwise correlation error ($\downarrow$)	Task ^a
<i>Adult</i>	11.66% $\pm$ 0.000	11.89% $\pm$ 0.000	0.898 $\pm$ 0.000	6.33% $\pm$ 0.000	4.35% $\pm$ 0.000	0.894 $\pm$ 0.000	5.44%$\pm$0.001	3.53%$\pm$0.000	0.900$\pm$0.001
<i>Blood</i>	9.84% $\pm$ 0.008	5.01% $\pm$ 0.005	0.722$\pm$0.033	5.54% $\pm$ 0.000	3.11% $\pm$ 0.000	0.703 $\pm$ 0.000	3.32%$\pm$0.005	2.68%$\pm$0.004	0.704 $\pm$ 0.026
<i>Car</i>	1.48% $\pm$ 0.002	3.70% $\pm$ 0.003	0.963$\pm$0.005	1.30%$\pm$0.003	3.49%$\pm$0.003	0.961 $\pm$ 0.006	1.68% $\pm$ 0.004	3.99% $\pm$ 0.004	0.958 $\pm$ 0.006
<i>Default</i>	12.42% $\pm$ 0.000	7.03% $\pm$ 0.000	0.752$\pm$0.000	12.54% $\pm$ 0.000	6.64% $\pm$ 0.000	0.724 $\pm$ 0.000	1.00%$\pm$0.001	5.09%$\pm$0.001	0.752$\pm$0.002
<i>Diamonds</i>	4.62% $\pm$ 0.000	2.54% $\pm$ 0.000	881.093$\pm$18.327	2.97%$\pm$0.000	2.54% $\pm$ 0.000	903.082 $\pm$ 0.000	3.79% $\pm$ 0.000	2.11%$\pm$0.000	934.090 $\pm$ 15.111
<i>Heart</i>	5.23%$\pm$0.005	10.83%$\pm$0.005	0.852 $\pm$ 0.017	5.93% $\pm$ 0.000	11.36% $\pm$ 0.000	0.873$\pm$0.000	6.08% $\pm$ 0.007	11.11% $\pm$ 0.006	0.823 $\pm$ 0.028
<i>Liver</i>	10.79% $\pm$ 0.000	2.51%$\pm$0.000	3.537 $\pm$ 0.000	15.01% $\pm$ 0.000	4.41% $\pm$ 0.000	3.362 $\pm$ 0.000	9.20%$\pm$0.000	3.49% $\pm$ 0.000	3.181$\pm$0.000
<i>Magic</i>	3.94% $\pm$ 0.000	1.23% $\pm$ 0.000	0.902 $\pm$ 0.000	2.60%$\pm$0.000	1.69% $\pm$ 0.000	0.910 $\pm$ 0.000	3.09% $\pm$ 0.000	0.98%$\pm$0.000	0.920$\pm$0.000

Task^a is AUROC ($\uparrow$) for classification and RMSE ($\downarrow$) for regression. All three LLM backbones use the same prompt, number of candidate queries, and strategy-selection protocol.

E.5 Sensitivity to LLM query budget

Because LLM-generated strategies are stochastic, we further examine whether the final performance depends strongly on a single LLM output. We conduct this query-budget analysis on three datasets, *Adult*, *Default*, and *Blood*. For each query budget k , we select the best-performing strategy among the first k candidate strategies according to the average statistical error, defined as the mean of marginal distribution error and pairwise correlation error. As shown in Supplementary Fig. 2, the best-so-far error generally decreases and then stabilizes as the number of candidate strategies increases. This suggests that querying the LLM multiple times improves strategy-selection robustness and reduces reliance on a single stochastic LLM output, while avoiding the need for extensive search.



Supplementary Fig. 2 Effect of LLM query budget on strategy selection. Best-so-far average statistical error, defined as the mean of marginal distribution error and pairwise correlation error, is shown as a function of the number of LLM-generated candidate strategies. For each query budget k , the selected strategy is the one with the lowest average statistical error among the first k candidates. Thin curves show results for *Adult*, *Default*, and *Blood*, and the bold curve shows their mean.

Supplementary Note F Detailed runtime of different algorithms on various datasets

Dataset-level runtime results are reported in Supplementary Table 12, including the total runtime of each baseline and the decomposition of TabSSD runtime into feature-CART extraction, LLM-based synthesis-strategy design, and execution of the generated strategy.

Supplementary Table 12 Runtime of different algorithms across datasets, including single-run and 10-run parallel TabSSD runtimes, in seconds.

Dataset	ARF	NRGB	CTGAN	TVAE	GReaT	TDDPM	TabDiff	TSYN	Tree	Query	Synth.	Total	Parallel query	Parallel synth.	Parallel total
Adult	586.29	1145.87	3950.74	1361.28	12492.07	2794.10	4449.82	2306	11.47	30.06	2.50	44.03	77.83	35.90	125.20
Default	881.77	1191.42	4172.98	1954.85	23840.83	2414.62	4849.87	3110	29.39	82.38	73.65	185.42	168.70	93.16	291.25
Magic	342.90	587.95	1846.34	883.19	9752.93	1504.44	3276.78	5518	7.91	48.62	2.57	59.10	85.09	13.79	106.79
Diamonds	911.43	764.76	4823.80	1952.56	14456.91	1697.19	5217.43	2997	5.48	34.71	12.12	52.31	90.55	43.04	139.07
Jasmine	233.54	1987.75	954.65	365.96	—	7830.38	4488.51	2651	13.92	97.88	28.19	139.99	169.99	1.29	185.20
SpeedDating	668.07	12392.89	2241.91	962.28	—	5865.19	4415.97	2309	36.65	73.06	5.91	115.62	378.45	2.45	417.55
DNA	322.20	1251.77	551.40	210.51	—	7833.71	4728.01	3872	88.61	78.93	5.12	172.66	120.66	4.95	214.22
DARWIN	14.54	8530.16	437.17	380.86	—	1177.65	2941.14	1226	129.78	289.65	10.02	429.45	289.65	3.56	422.99
Blood	7.05	327.37	58.93	44.44	190.21	1622.75	2980.23	1304	0.14	74.08	0.29	74.51	53.81	5.93	59.88
Car	9.25	415.17	88.48	42.11	384.97	1733.42	1663.72	1619	0.20	86.41	0.12	86.73	197.71	13.14	211.05
Heart	6.54	334.39	73.97	30.23	116.48	2018.31	2621.01	1257	0.37	96.54	5.10	102.01	100.06	5.37	105.80
Liver	7.54	388.90	50.88	27.09	94.72	1390.71	1662.97	1489	0.08	56.62	0.04	56.74	112.35	3.68	116.11

“—” indicates that the corresponding method could not be successfully executed on the dataset. NRGB, TDDPM, and TSYN denote NRGBBoost, TabDDPM, and TABSYN, respectively.

For TabSSD, Tree, Query, and Synth. denote the runtimes required for feature-CART extraction, generation of the selected synthesis strategy, and execution of the selected strategy, respectively. Total is the sum of Tree, Query, and Synth.

The parallel columns correspond to 10 independent candidate runs. Parallel query and Parallel synth. denote the maximum single-query time and single-synthesis time, respectively, across these runs. Parallel total is the sum of Tree, Parallel query, and Parallel synth. The runtime analyses in Fig. 3 use Parallel total as the TabSSD end-to-end runtime.

Supplementary Note G Case study of failure modes

Not all LLM-generated synthesis strategies are reliable. Supplementary Table 13 presents representative failure cases in terms of fidelity, privacy, and efficiency. Unlike black-box generators, our framework produces inspectable executable strategies, allowing failure sources to be traced to specific code-level design choices and unreliable strategies to be excluded through multi-objective evaluation before final selection and deployment.

Supplementary Table 13 Representative failure modes in LLM-generated synthesis strategies. The displayed code fragments highlight the operations directly responsible for each failure.

Dataset	Observed failure	Failure-inducing design and key code fragment
Blood	Low fidelity Marginal distribution error: 28.80% Pairwise correlation error: 23.82%	Subspace-specific Gaussian mixture models are used to generate the strongly dependent continuous variables <code>feature_1</code> and <code>feature_2</code> . Repeated single-sample generation with a fixed random state concentrates the outputs on only a few recurring value pairs, causing distribution collapse. The regenerated routing attributes are also not constrained to remain in their assigned subspaces. <pre>gmm = GaussianMixture(..., random_state=42) gmm.fit(subspace_data[['feature_1', 'feature_2']]) f1_f2_sample, _ = subspace_model['gmm'].sample(1) f0 = f0_pred + f0_residual</pre>
Car	Privacy leakage DCR: 0.994 Train-set proportion: 0.700	The strategy stores complete feature-label records within each subspace and directly resamples an observed row with probability 0.98. Consequently, most synthetic records are copied from the training support rather than newly generated. Laplace smoothing changes only sampling probabilities and cannot prevent exact matches. <pre>self.unique_vectors = unique_rows if np.random.rand() > diversity_rate: # diversity_rate = 0.02 idx = np.random.choice(len(self.unique_vectors), ...) return self.unique_vectors[idx]</pre>
Jasmine	Excessive cost Runtime: 1,888.74 s	The strategy partitions a 144-feature dataset into multiple subspaces and trains a separate conditional boosting model for nearly every feature in each subspace. During synthesis, it further generates each record feature by feature, producing substantial nested training and sampling overhead. <pre>for space_id in sorted(df['subspace_id'].unique()): for target_feature in GENERATION_ORDER: model.fit(X_fit, y_fit) for _ in range(num_samples_for_subspace): for target_feature in GENERATION_ORDER:</pre>

Supplementary Note H Detailed predictive results for clinical data augmentation

Detailed predictive results under the 2× augmentation setting are reported in Supplementary Table 14. Balanced accuracy and minority-class accuracy are evaluated on the real test set.

Supplementary Table 14 Predictive utility under the 2× augmentation setting.

Method	Balanced accuracy	Minority-class accuracy
TabSSD	0.5785 ± 0.0162	0.5874 ± 0.0331
SMOTE	0.5638 ± 0.0158	0.2557 ± 0.0303
ARF	0.5499 ± 0.0251	0.3259 ± 0.0495
CTGAN	0.5106 ± 0.0130	0.1052 ± 0.0254
TVAE	0.5197 ± 0.0051	0.2586 ± 0.0102
GReaT	0.5229 ± 0.0199	0.2236 ± 0.0460
TabDDPM	0.5550 ± 0.0141	0.2224 ± 0.0290
TabDiff	0.5634 ± 0.0203	0.3236 ± 0.0459
TABSYN	0.5360 ± 0.0168	0.2460 ± 0.0306
NRGBoost	0.5525 ± 0.0108	0.1707 ± 0.0202

Supplementary Note I Tailored data synthesis strategies

This section presents the concrete Python synthesis strategies generated by TabSSD for the selected datasets. For clarity, only the executable strategy code is shown, without reproducing the corresponding LLM analyses. Owing to its substantially greater length, the strategy for the *DARWIN* dataset is presented in an abridged form, with part of the embedded feature-dependence information omitted for brevity.

Listing 2 Generated synthesis strategy for the Blood dataset.

```
def generate_synthesis_data(save_path: str, n_sample: int, X_train: np.ndarray, y_train: np.
ndarray) -> None:
    """
    Generates synthetic tabular data by inferring characteristics from pre-trained decision
    trees.

    The strategy involves:
    1. Analyzing feature dependencies from provided tree performance metrics.
    2. Using the main label-prediction decision tree to define subspaces (leaves).
    3. Fitting a Gaussian Copula model to the continuous features within each subspace to
    capture local correlations.
    4. Modeling the categorical label based on its empirical distribution in each subspace.
    5. Generating new data by sampling from these fitted models, ensuring generated values
    respect subspace boundaries.
    """
    # All necessary imports are contained within the function.
    import numpy as np
    import pandas as pd
    from sklearn.preprocessing import QuantileTransformer
    import csv
    import warnings

    # Suppress warnings from QuantileTransformer for small n_quantiles
    warnings.filterwarnings("ignore", category=UserWarning)

    feature_names = [f"feature_{i}" for i in range(4)]
    all_feature_names = feature_names + ["label"]

    # Define global feature ranges for final clamping
    feature_ranges = {
        0: (0.0, 74.0),
        1: (1.0, 50.0),
        2: (250.0, 12500.0),
        3: (2.0, 98.0)
    }

    # Helper function to get leaf boundaries for clamping. This is critical for fidelity.
    def get_leaf_bounds():
        bounds = {
            1: {0: (-np.inf, 6.5), 2: (-np.inf, 1125.0), 3: (-np.inf, 3.0)},
            2: {0: (-np.inf, 5.0), 2: (-np.inf, 1125.0), 3: (3.0, 12.0)},
            3: {0: (5.0, 6.5), 2: (-np.inf, 1125.0), 3: (3.0, 12.0)},
            4: {0: (-np.inf, 6.5), 2: (-np.inf, 625.0), 3: (12.0, np.inf)},
            5: {0: (-np.inf, 6.5), 2: (625.0, 1125.0), 3: (12.0, 24.5)},
            6: {0: (-np.inf, 6.5), 2: (625.0, 1125.0), 3: (24.5, np.inf)},
            7: {0: (-np.inf, 6.5), 2: (1125.0, np.inf), 3: (-np.inf, 22.5)},
            8: {0: (-np.inf, 6.5), 2: (1125.0, np.inf), 3: (22.5, 31.0)},
            9: {0: (-np.inf, 6.5), 2: (1125.0, np.inf), 3: (31.0, 33.5)},
            10: {0: (-np.inf, 6.5), 1: (-np.inf, 5.5), 2: (1125.0, np.inf), 3: (33.5, 50.0)},
            11: {0: (-np.inf, 6.5), 1: (5.5, np.inf), 2: (1125.0, np.inf), 3: (33.5, 50.0)},
            12: {0: (-np.inf, 6.5), 1: (-np.inf, 11.5), 2: (1125.0, np.inf), 3: (50.0, np.inf)},
            13: {0: (-np.inf, 6.5), 1: (-np.inf, 15.0), 2: (1125.0, 4500.0), 3: (50.0, 78.5)},
            14: {0: (-np.inf, 3.5), 1: (15.0, np.inf), 2: (1125.0, 4500.0), 3: (50.0, 78.5)},
            15: {0: (3.5, 5.0), 1: (15.0, np.inf), 2: (1125.0, 4500.0), 3: (50.0, 78.5)},
            16: {0: (5.0, 6.5), 1: (15.0, np.inf), 2: (1125.0, 4500.0), 3: (50.0, 78.5)},
            17: {0: (-np.inf, 6.5), 2: (1125.0, 4500.0), 3: (78.5, np.inf)},
            18: {0: (-np.inf, 6.5), 2: (4500.0, np.inf), 3: (50.0, np.inf)},
            19: {0: (6.5, np.inf), 3: (-np.inf, 58.5)},
            20: {0: (6.5, np.inf), 3: (58.5, np.inf)}
        }
    }
    return bounds

LEAF_BOUNDS = get_leaf_bounds()

# Step 1: Define the Routing Function based on the decision tree for the label.
def get_leaf_id(sample):
    f0, f1, f2, f3 = sample['feature_0'], sample['feature_1'], sample['feature_2'], sample['
feature_3']
    if f0 <= 6.5:
        if f2 <= 1125.0:
            if f3 <= 3.0: return 1
            else:
                if f3 <= 12.0:
                    if f0 <= 5.0: return 2
                    else: return 3
                else:
                    if f2 <= 625.0: return 4
                    else:
```

```

534         if f3 <= 24.5: return 5
535         else: return 6
536     else:
537         if f3 <= 50.0:
538             if f3 <= 22.5: return 7
539             else:
540                 if f3 <= 31.0: return 8
541                 else:
542                     if f3 <= 33.5: return 9
543                     else:
544                         if f1 <= 5.5: return 10
545                         else: return 11
546         else:
547             if f1 <= 11.5: return 12
548             else:
549                 if f2 <= 4500.0:
550                     if f3 <= 78.5:
551                         if f1 <= 15.0: return 13
552                         else:
553                             if f0 <= 3.5: return 14
554                             else:
555                                 if f0 <= 5.0: return 15
556                                 else: return 16
557                     else: return 17
558                 else: return 18
559     else:
560         if f3 <= 58.5: return 19
561         else: return 20
562     return -1 # Should not be reached with valid data
563
564 # Step 2: Partition the original data into subspaces using the routing function.
565 X_df = pd.DataFrame(X_train, columns=feature_names)
566 data_df = X_df.copy()
567 data_df['label'] = y_train
568
569 subspaces = {i: [] for i in range(1, 21)}
570 for _, row in data_df.iterrows():
571     leaf_id = get_leaf_id(row)
572     if leaf_id != -1:
573         subspaces[leaf_id].append(row.values)
574
575 for leaf_id in subspaces:
576     subspaces[leaf_id] = pd.DataFrame(subspaces[leaf_id], columns=all_feature_names)
577
578 # Step 3: Learn a generative model for each subspace.
579 models = {}
580 for leaf_id, subspace_df in subspaces.items():
581     n_points = len(subspace_df)
582     if n_points < 4: # Need a few points to fit a model reliably.
583         models[leaf_id] = None
584         continue
585
586     continuous_data = subspace_df[feature_names]
587     labels = subspace_df['label']
588
589     qt = QuantileTransformer(output_distribution='normal', n_quantiles=max(2, min(n_points
590 // 2, 100)), subsample=min(n_points, int(1e5)))
591     transformed_data = qt.fit_transform(continuous_data)
592     transformed_data = np.nan_to_num(transformed_data) # Handle constant columns
593
594     copula_corr = np.corrcoef(transformed_data, rowvar=False)
595     copula_corr = np.nan_to_num(copula_corr, nan=0.0) # Handle NaN in correlation matrix
596     np.fill_diagonal(copula_corr, 1.0)
597
598     label_dist = labels.value_counts(normalize=True).to_dict()
599     if 0 not in label_dist: label_dist[0] = 0.0
600     if 1 not in label_dist: label_dist[1] = 0.0
601
602     models[leaf_id] = {
603         "qt": qt,
604         "copula_corr": copula_corr,
605         "label_dist": label_dist,
606         "n_points": n_points
607     }
608
609 # Step 4: Generate new data by sampling from the fitted models.
610 total_original_points = len(data_df) if len(data_df) > 0 else 1
611 synthesis_samples_list = []
612
613 for leaf_id, model in models.items():

```

```

614         if model is None:
615             continue
616
617         n_samples_leaf = int(np.round(n_sample * model['n_points'] / total_original_points))
618         if n_samples_leaf == 0:
619             continue
620
621         mean = np.zeros(len(feature_names))
622         try:
623             # Add a small ridge to ensure matrix is positive semi-definite
624             cov_matrix = model['copula_corr'] + np.eye(len(feature_names)) * 1e-6
625             mvn_samples = np.random.multivariate_normal(mean, cov_matrix, n_samples_leaf)
626         except np.linalg.LinAlgError:
627             continue # Skip subspace if generation fails
628
629         synthetic_continuous = model['qt'].inverse_transform(mvn_samples)
630
631         leaf_bounds = LEAF_BOUNDS.get(leaf_id, {})
632         for i in range(len(feature_names)):
633             global_min, global_max = feature_ranges[i]
634             leaf_min, leaf_max = leaf_bounds.get(i, (-np.inf, np.inf))
635             final_min = max(global_min, leaf_min)
636             final_max = min(global_max, leaf_max)
637             synthetic_continuous[:, i] = np.clip(synthetic_continuous[:, i], final_min,
638 final_max)
639
640         labels_to_gen = list(model['label_dist'].keys())
641         label_probs = list(model['label_dist'].values())
642         synthetic_labels = np.random.choice(labels_to_gen, size=n_samples_leaf, p=label_probs)
643
644         for j in range(n_samples_leaf):
645             record = {fname: val for fname, val in zip(feature_names, synthetic_continuous[j])}
646             record['label'] = int(synthetic_labels[j])
647             synthesis_samples_list.append(record)
648
649         # Adjust sample count to exactly match n_sample
650         if len(synthesis_samples_list) > n_sample:
651             synthesis_samples_list = synthesis_samples_list[:n_sample]
652         elif len(synthesis_samples_list) < n_sample and synthesis_samples_list:
653             diff = n_sample - len(synthesis_samples_list)
654             extras = np.random.choice(synthesis_samples_list, size=diff)
655             synthesis_samples_list.extend(extras)
656
657         # Step 5: Save the generated data to a CSV file.
658         with open(save_path, "w", newline="") as csv_file:
659             writer = csv.DictWriter(csv_file, fieldnames=all_feature_names)
660             writer.writeheader()
661             if synthesis_samples_list:
662                 writer.writerows(synthesis_samples_list)

```

Listing 3 Generated synthesis strategy for the Car dataset.

```

664 def generate_synthesis_data(save_path: str, n_sample: int, X_train: np.ndarray, y_train: np.
665 ndarray) -> None:
666     import pandas as pd
667     import numpy as np
668     import csv
669
670     # 1. Data Preparation
671     # Map X_train to features 0-5 and y_train to label
672     feat_cols = [f"feature_{i}" for i in range(6)]
673     df = pd.DataFrame(X_train, columns=feat_cols)
674     df['label'] = y_train
675
676     # 2. Routing Function: Partitioning into Subspaces based on Decision Tree logic
677     def get_subspace(data):
678         f3 = data['feature_3']
679         f5 = data['feature_5']
680         # Condition 1: feature_3 <= 0.5 (Significant split for Class 2)
681         # Condition 2: feature_3 > 0.5 and feature_5 <= 0.5
682         # Condition 3: feature_3 > 0.5 and 0.5 < feature_5 <= 1.5
683         # Condition 4: feature_3 > 0.5 and feature_5 > 1.5
684         conditions = [
685             (f3 <= 0.5),
686             (f3 > 0.5) & (f5 <= 0.5),
687             (f3 > 0.5) & (f5 > 0.5) & (f5 <= 1.5),
688             (f3 > 0.5) & (f5 > 1.5)
689         ]
690         choices = [1, 2, 3, 4]

```

```

692     # Use select to assign each row to a subspace ID
693     return np.select(conditions, choices, default=4)
694
695 df['subspace'] = get_subspace(df)
696
697 # 3. Fit: Learning distribution characteristics within each subspace
698 subspace_stats = {}
699 subspace_weights_series = df['subspace'].value_counts(normalize=True)
700
701 for s_id in df['subspace'].unique():
702     s_df = df[df['subspace'] == s_id]
703
704     # Model P(Label | Subspace)
705     l_counts = s_df['label'].value_counts(normalize=True).to_dict()
706     l_vals = list(l_counts.keys())
707     l_weights = np.array(list(l_counts.values()), dtype='float64')
708     l_weights /= l_weights.sum() # Normalize for safety
709
710     # Model P(Feature_i | Label, Subspace)
711     f_probs_given_l = {}
712     for l_val in s_df['label'].unique():
713         sl_df = s_df[s_df['label'] == l_val]
714         f_probs_given_l[l_val] = {}
715         for col in feat_cols:
716             counts = sl_df[col].value_counts(normalize=True).to_dict()
717             f_v = list(counts.keys())
718             f_w = np.array(list(counts.values()), dtype='float64')
719             f_w /= f_w.sum()
720             f_probs_given_l[l_val][col] = (f_v, f_w)
721
722     subspace_stats[s_id] = {
723         'label_dist': (l_vals, l_weights),
724         'feature_dists': f_probs_given_l
725     }
726
727 # 4. Synthesize Data
728 synthesis_samples = []
729 s_ids = list(subspace_stats.keys())
730 s_probs = [subspace_weights_series.get(sid, 0) for sid in s_ids]
731 s_probs = np.array(s_probs) / sum(s_probs)
732
733 for _ in range(n_sample):
734     # Sample Subspace
735     sid = np.random.choice(s_ids, p=s_probs)
736     stats = subspace_stats[sid]
737
738     # Sample Label within Subspace
739     l_vals, l_weights = stats['label_dist']
740     l_choice = np.random.choice(l_vals, p=l_weights)
741
742     # Sample Features conditioned on Label and Subspace
743     row = {'label': int(l_choice)}
744     for col in feat_cols:
745         f_vals, f_weights = stats['feature_dists'][l_choice][col]
746         f_choice = np.random.choice(f_vals, p=f_weights)
747         row[col] = int(f_choice)
748
749     synthesis_samples.append(row)
750
751 # 5. Output to CSV
752 fieldnames = feat_cols + ["label"]
753 with open(save_path, "w", newline="") as csv_file:
754     writer = csv.DictWriter(csv_file, fieldnames=fieldnames)
755     writer.writeheader()
756     writer.writerows(synthesis_samples)
757

```

Listing 4 Abridged excerpt from the synthesis strategy generated for the *DARWIN* dataset.

```

758 def generate_synthesis_data(save_path: str, n_sample: int, X_train:
759                             np.ndarray, y_train: np.ndarray) -> None:
760
761     """
762     Generates synthetic tabular data based on inferred characteristics from pre-trained decision
763     trees.
764
765     The strategy involves:
766     1. Parsing decision tree summaries to build a feature dependency graph.
767     2. Performing a topological sort to establish a valid, sequential generation order.
768     3. Using the final label-prediction tree to define a "routing function" that partitions
769     the data space into subspaces with homogeneous labels.

```

```

770 4. Fitting models within each subspace to capture local data distributions:
771 - Linear Regression + Residual Sampling for features with strong predictors.
772 - Empirical Distribution Sampling for root/un-modeled features.
773 5. Sequentially generating new samples according to the determined order and fitted models,
774 ensuring values are within known bounds.
775 6. Saving the generated data to a CSV file.
776 """
777 import pandas as pd
778 import numpy as np
779 import csv
780 import re
781 from collections import defaultdict, Counter
782 from sklearn.linear_model import LinearRegression
783 import warnings
784
785 warnings.filterwarnings("ignore", category=UserWarning)
786
787 # 1. Parse feature information and define constants
788 # The decision tree rule strings are embedded for self-containment.
789 feature_info_str = """
790 {feature_49}:{type:continuous,min:0.0,max:14.0,associated_features:[feature_36,feature_38,
791 feature_334,feature_105],performance:{R2:0.9911,RMSE:0.2114}}
792 {feature_94}:{type:continuous,min:736.0,max:9830.0,associated_features:[feature_251,
793 feature_95,feature_201,feature_100],performance:{R2:0.9752,RMSE:199.1262}}
794
795 ... additional feature summaries omitted for brevity ...
796
797 {feature_41}:{type:continuous,min:0.0,max:4456.0,associated_features:[feature_53,feature_49,
798 feature_48,feature_227],performance:{R2:0.9838,RMSE:87.5136}}
799 {feature_88}:{type:continuous,min:0.0,max:340361.9688,associated_features:[feature_87,
800 feature_122,feature_248,feature_222],performance:{R2:0.9922,RMSE:4425.3103}}
801 """
802
803 def parse_feature_info(info_str):
804     feature_models = {}
805     feature metas = {}
806     pattern = re.compile(
807         r"\{feature_(\d+)\}:\{type:([\^,]+),min:([\^,]+),max:([\^,]+),associated_features"
808         :\[([\^,]]*)\},performance:\{R2:([\^,]+),RMSE:([\^,]]+)\}\}"
809     )
810     for line in info_str.strip().split('\n'):
811         match = pattern.match(line.strip())
812         if match:
813             target_id = int(match.group(1))
814             assoc_str = match.group(5)
815             predictors = []
816             if assoc_str:
817                 predictors = [int(f.replace('feature_', '')) for f in assoc_str.split(',')]
818
819             feature_models[target_id] = predictors
820             feature metas[target_id] = {
821                 'type': match.group(2),
822                 'min': float(match.group(3)),
823                 'max': float(match.group(4)),
824                 'r2': float(match.group(6)),
825                 'rmse': float(match.group(7))
826             }
827     return feature_models, feature metas
828
829 feature_models, feature metas = parse_feature_info(feature_info_str)
830
831 NUM_FEATURES = 450
832 all_feature_indices = list(range(NUM_FEATURES))
833 feature_names = [f"feature_{i}" for i in all_feature_indices]
834
835 CONSTANT_FEATURES = {k for k, v in feature metas.items() if v['min'] == v['max']}
836
837 # 2. Define the Subspace Routing Logic from the label decision tree
838 def get_subspace_id(row):
839     if row['feature_413'] <= 8690.0:
840         if row['feature_446'] <= 54097.5:
841             if row['feature_204'] <= 1.61:
842                 return 0 # Leaf 1 -> Label 0
843             else:
844                 return 1 # Leaf 2 -> Label 1
845         else: # feature_446 > 54097.5
846             return 2 # Leaf 3 -> Label 1
847     else: # feature_413 > 8690.0
848         if row['feature_305'] <= 28607.5:
849             if row['feature_202'] <= 763.0:

```

```

850         return 3 # Leaf 4 -> Label 0
851     else: # feature_202 > 763.0
852         if row['feature_171'] <= 0.03:
853             return 4 # Leaf 5 -> Label 0
854         else: # feature_171 > 0.03
855             return 5 # Leaf 6 -> Label 1
856     else: # feature_305 > 28607.5
857         if row['feature_329'] <= 11045.5:
858             return 6 # Leaf 7 -> Label 1
859         else: # feature_329 > 11045.5
860             return 7 # Leaf 8 -> Label 0
861
862     subspace_labels = {0: 0, 1: 1, 2: 1, 3: 0, 4: 0, 5: 1, 6: 1, 7: 0}
863
864     # 3. Determine Generation Order via Topological Sort
865     dependencies = {k: v for k, v in feature_models.items()}
866     all_features_set = set(all_feature_indices)
867
868     generation_order = []
869     generated_features = set()
870
871     # Start with features that are not targets of any model (root nodes)
872     unmodeled_targets = all_features_set - set(dependencies.keys())
873     generation_order.extend(list(unmodeled_targets))
874     generated_features.update(unmodeled_targets)
875
876     # Iteratively add features whose dependencies are met
877     while len(generated_features) < len(all_features_set):
878         features_to_add = set()
879         for f in (all_features_set - generated_features):
880             if f in dependencies:
881                 preds = dependencies.get(f, [])
882                 if all(p in generated_features for p in preds):
883                     features_to_add.add(f)
884
885         if not features_to_add:
886             # Cycle detected. Break it by finding the feature with the lowest R2 in the cycle.
887             cyclic_features = all_features_set - generated_features
888             feature_to_break = min(cyclic_features, key=lambda f: feature metas.get(f, {'r2':
889             999}))['r2'])
890             features_to_add.add(feature_to_break)
891
892         # Sort to make order deterministic, then add
893         for f in sorted(list(features_to_add)):
894             generation_order.append(f)
895             generated_features.add(f)
896
897     # 4. Synthesizer Class
898     class SubspaceSynthesizer:
899         def __init__(self, order, metas, models, constants, num_features):
900             self.order = order
901             self.metas = metas
902             self.models = models
903             self.constants = constants
904             self.num_features = num_features
905
906             self.subspace_models = defaultdict(dict)
907             self.subspace_residuals = defaultdict(dict)
908             self.subspace_marginals = defaultdict(dict)
909             self.subspace_proportions = {}
910             self.global_marginals = {} # Fallback
911
912         def fit(self, df):
913             df['subspace_id'] = df.apply(get_subspace_id, axis=1)
914             self.subspace_proportions = df['subspace_id'].value_counts(normalize=True).to_dict()
915
916             # Store global marginals as a fallback
917             for feature_id in self.order:
918                 self.global_marginals[feature_id] = df[f'feature_{feature_id}'].values
919
920             grouped = df.groupby('subspace_id')
921             for subspace_id, subspace_df in grouped:
922                 if subspace_df.empty: continue
923
924                 for feature_id in self.order:
925                     fname = f'feature_{feature_id}'
926
927                     if feature_id in self.constants: continue
928
929                     # Decide if this feature is modeled conditionally or marginally

```

```

930         is_conditionally_modeled = feature_id in self.models and all(
931             p in self.order[:self.order.index(feature_id)] for p in self.models[
932 feature_id]
933         )
934
935         if is_conditionally_modeled:
936             predictors = self.models[feature_id]
937             pred_names = [f'feature_{p}' for p in predictors]
938
939             # Ensure all predictor columns exist in the subspace df
940             if not all(p in subspace_df.columns for p in pred_names):
941                 is_conditionally_modeled = False
942             else:
943                 X_sub = subspace_df[pred_names]
944                 y_sub = subspace_df[fname]
945
946                 if X_sub.empty or y_sub.empty:
947                     is_conditionally_modeled = False
948                 else:
949                     try:
950                         model = LinearRegression()
951                         model.fit(X_sub, y_sub)
952                         predictions = model.predict(X_sub)
953                         residuals = y_sub - predictions
954
955                         self.subspace_models[subspace_id][feature_id] = model
956                         self.subspace_residuals[subspace_id][feature_id] = residuals.
957 values
958
959                     except ValueError: # E.g., if X_sub has no variation
960                         is_conditionally_modeled = False
961
962             if not is_conditionally_modeled:
963                 self.subspace_marginals[subspace_id][feature_id] = subspace_df[fname].
964 values
965
966         def sample(self, n_samples):
967             all_samples = []
968
969             subspace_ids = list(self.subspace_proportions.keys())
970             subspace_probs = list(self.subspace_proportions.values())
971
972             if not subspace_ids: # Handle case where training data is empty
973                 return []
974
975             # Assign samples to subspaces based on proportions
976             assignments = np.random.choice(subspace_ids, size=n_samples, p=subspace_probs)
977
978             for i in range(n_samples):
979                 subspace_id = assignments[i]
980                 new_sample = {}
981
982                 for feature_id in self.order:
983                     fname = f'feature_{feature_id}'
984                     fmeta = self.metas.get(feature_id, {})
985                     fmin = fmeta.get('min', -np.inf)
986                     fmax = fmeta.get('max', np.inf)
987
988                     val = 0.0 # Default value
989                     if feature_id in self.constants:
990                         val = 0.0
991                     elif feature_id in self.subspace_models.get(subspace_id, {}):
992                         model = self.subspace_models[subspace_id][feature_id]
993                         residuals = self.subspace_residuals[subspace_id][feature_id]
994                         predictors = self.models[feature_id]
995                         pred_names = [f'feature_{p}' for p in predictors]
996
997                         pred_vals_list = [new_sample[p_name] for p_name in pred_names]
998                         pred_df = pd.DataFrame([pred_vals_list], columns=pred_names)
999
1000                         base_val = model.predict(pred_df)[0]
1001                         residual = np.random.choice(residuals) if len(residuals) > 0 else 0
1002                         val = base_val + residual
1003                     else:
1004                         # Use subspace marginal if available, else global marginal as fallback
1005                         marginal_vals = self.subspace_marginals.get(subspace_id, {}).get(
1006 feature_id)
1007
1008                         if marginal_vals is None or len(marginal_vals) == 0:
1009                             marginal_vals = self.global_marginals.get(feature_id)
1010
1011                         if marginal_vals is not None and len(marginal_vals) > 0:

```

```

1010         val = np.random.choice(marginal_vals)
1011     else: # Ultimate fallback for features with no data
1012         val = np.random.uniform(fmin if np.isfinite(fmin) else 0, fmax if np.
1013 isfinite(fmax) else 1)
1014
1015     # Clip to ensure validity
1016     new_sample[fname] = np.clip(val, fmin, fmax)
1017
1018     new_sample['label'] = subspace_labels[subspace_id]
1019     all_samples.append(new_sample)
1020
1021     return all_samples
1022
1023 # 5. Main execution logic
1024 df_train = pd.DataFrame(X_train, columns=feature_names)
1025 df_train['label'] = y_train
1026
1027 # Instantiate, fit, and sample
1028 synthesizer = SubspaceSynthesizer(generation_order, feature metas, feature_models,
1029 CONSTANT_FEATURES, NUM_FEATURES)
1030 synthesizer.fit(df_train)
1031 synthetic_samples = synthesizer.sample(n_sample)
1032
1033 # 6. Save to CSV
1034 if not synthetic_samples:
1035     print("Warning: No synthetic samples were generated.")
1036     return
1037
1038 output_fieldnames = [f"feature_{i}" for i in all_feature_indices] + ["label"]
1039 with open(save_path, "w", newline="") as csv_file:
1040     writer = csv.DictWriter(csv_file, fieldnames=output_fieldnames)
1041     writer.writeheader()
1042     writer.writerows(synthetic_samples)
1043
1044 print(f"Successfully generated {len(synthetic_samples)} synthetic samples and saved to {
1045 save_path}")
1046

```