Modeling Click Through Rate

Hanxi Chen, Manuela Lozano, Harshdeep Sidhu, Alton Law

Abstract

This project addresses the challenges of nonlinear user behaviour and extreme class imbalance in digital advertising by combining predictive modeling and generative artificial intelligence to predict click-through rates. We analyze a large-scale dataset containing advertising interactions and news feed engagement, and use CatBoost as the primary predictive model after evaluating a logistic regression baseline. Catboost demonstrates strong performance and reveals key behavioural and ad-level predictors of click behaviour. To address the rarity of positive click events, we train a Conditional Tabular GAN to generate principal component projections. The synthetic samples preserve important statistical properties of the real minority class and highlight the potential of generative artificial intelligence for augmentation and analysis in rare event prediction. Together, the predictive and generative models provide complementary insights into user engagement patterns and offer practical avenues for improving click-through rate modeling in digital advertising.

Introduction

Predictive analytics plays an essential role in digital marketing by enabling advertisers to anticipate user behavior, improve audience targeting, and optimize campaign performance. Machine learning models allow marketers to analyze historical interactions and forecast the likelihood of a click. Since advertisers rely on click-through rate to evaluate ad effectiveness and make decisions about pricing, placement, and bidding strategies, accurate CTR prediction directly influences revenue and campaign success. Understanding and predicting click behavior is therefore fundamental to effective advertising technology.

Click-through rate measures how often users click on an advertisement relative to how many times it is displayed. It reflects the extent to which an ad captures user attention and aligns with user interests. A higher CTR indicates strong engagement and often suggests that the ad content, creative design, and targeting strategies are performing effectively. Because CTR is a direct measure of user responsiveness, it is one of the most widely used indicators of ad performance. CTR is computed by dividing the number of clicks an ad receives by the number of impressions it generates, then expressing the result as a percentage. For example, fifty clicks out of ten thousand impressions correspond to a CTR of zero point five percent. Although CTR benchmarks vary by industry, ad format, and platform, tracking this metric over time helps advertisers refine their strategies, improve personalization, and allocate budgets more efficiently. Generative AI refers to a class of machine learning models that learn the underlying structure of data and create new synthetic samples that resemble the original dataset. Unlike traditional discriminative models, which focus on predicting labels, generative models aim to capture the full joint distribution of the features. This makes them particularly powerful for data

augmentation, privacy-preserving modeling, and tasks involving underrepresented minority classes.

Our motivation for this project was to explore how predictive modeling and generative artificial intelligence can work together in this setting. We aimed to evaluate the strengths and limitations of a traditional predictive model and then investigate whether synthetic data generated from a generative model can help represent rare click events. This approach mirrors real problems faced in digital advertising and provides insights into the role of these models in improving click-through rate prediction.

Literature Review

Early click-through rate (CTR) prediction methods relied on traditional machine learning models that used manually engineered features. These models incorporated information from different stages of user interaction, including current impressions, past impressions within a session, and contextual features describing the browsing environment. Techniques such as Support Vector Machines, Conditional Random Fields, Decision Trees, and Back Propagation Neural Networks were commonly used. Research consistently showed that modeling user intent, session context, and behavioral history improved predictive accuracy.

Modern approaches shift toward representation learning through deep neural architectures. Two tower models encode user behavior and item characteristics separately and project them into a shared latent space to measure compatibility. Extensions of these architectures introduce context encoders that capture temporal and device-level factors that influence engagement. More recent developments incorporate multi-domain information, multi-task learning, and attention-based models that prioritize the most relevant parts of a user's interaction history. These advances demonstrate a clear evolution from manually engineered features toward deep learning systems capable of automatically extracting complex patterns, reducing sparsity, and improving personalization.

Initial applications of generative models for tabular data focused on adapting GAN architectures originally developed for images to structured datasets. Initial approaches demonstrated that adversarial training could synthesize minority-class examples that resembled real observations, helping address severe class imbalance without relying on simple resampling techniques. Surveys across multiple domains, including fraud detection, medical diagnosis, and churn prediction, consistently showed that GAN-based augmentation improved minority-class recall and provided more realistic samples than traditional oversamplers such as SMOTE. These early models established the foundation for generative AI as a viable method for improving prediction quality in rare-event settings where observational data are limited.

Modern generative approaches introduced specialized architectures tailored to the unique properties of tabular data. Conditional Tabular GAN (CTGAN) emerged as a leading model by

incorporating techniques such as mode-specific normalization, conditional sampling, and mixed-type feature handling. These innovations allow CTGAN to learn complex joint distributions involving categorical and continuous variables while generating high-quality synthetic samples targeted toward the minority class. Parallel developments led to the Tabular Variational Autoencoder (TVAE), which leverages a smooth latent space to capture continuous relationships and produce more stable synthetic records with reduced noise. Together, CTGAN and TVAE represent a shift toward generative systems that explicitly model feature dependencies, multimodal distributions, and rare-class structure.

Recent research extends these models into new directions, integrating differential privacy, hybrid sampling strategies, and high-dimensional data modeling. Approaches such as DP-CGAN and PrivBayes demonstrate that synthetic data can improve minority-class detection while preserving user confidentiality through privacy-focused mechanisms. Other studies combine CTGAN-based augmentation with undersampling frameworks to address both class imbalance and class overlap in complex domains. These advancements highlight a clear evolution from early adversarial generators toward more sophisticated generative architectures capable of capturing intricate feature relationships, enhancing minority-class representation, and supporting responsible data use in sensitive applications.

Exploratory Data Analysis

The dataset contains information from two environments that together describe user behaviour. The advertising domain contains detailed user interaction logs such as clicks and ad closes, demographic information including age, gender, and location, and device-specific features such as screen size and network type. It also includes rich ad metadata describing campaign identifiers, creative formats, placement characteristics, and additional contextual tags. Together, these variables capture how users interact with ads and provide the structure needed to model click likelihood.

The news feed domain captures interactions unrelated to advertising but closely tied to user interests. This includes the types of articles users read, the categories they favor, how frequently they refresh content, and behavioral signals such as likes and negative feedback. It also contains metadata describing article content, topics, and exposure formats. When combined, the two domains provide both explicit advertising behavior and broader behavioral signals that reflect user preferences. The shared user identifiers and timestamps across domains allow the model to incorporate sequential patterns, longer-term interests, and contextual signals that enrich the prediction task. This creates an ideal setting for cross-domain CTR modeling.

A key feature of the dataset is the extreme imbalance between click and non-click labels. A bar plot of the label distribution shows a very large spike for non-click events and a nearly invisible

bar for click events. This confirms that the dataset reflects a real-world digital advertising environment where only a very small percentage of impressions result in a click. Such an imbalance causes many models to overlook the minority class and, therefore, highlights the importance of resampling and advanced modeling strategies.

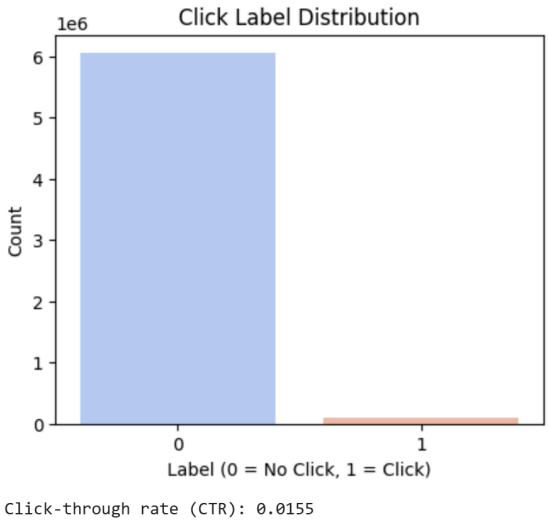


Figure 1. Click Label Distribution

A correlation heatmap was created to examine relationships between numerical features. The heatmap displayed mostly light colors, indicating weak correlations among the majority of features. Only a few pairs showed moderate associations, suggesting that linear methods will not capture the underlying pattern. This reinforces the need for models capable of identifying nonlinear interactions, because click behaviour likely results from complex combinations of demographic signals, behavioral tendencies, and contextual information.

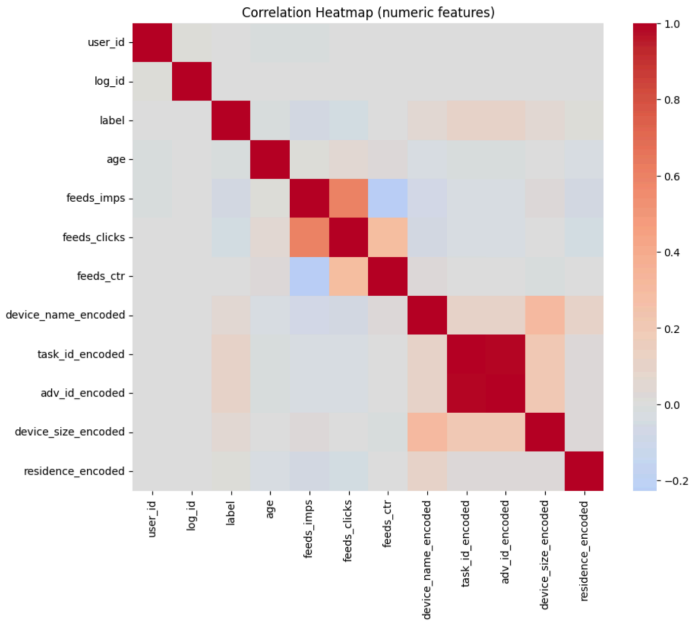


Figure 2. Correlation Heatmap

Analysis

Predictive Models

We began by preparing the data for baseline modeling using a subset of available features to establish performance benchmarks before exploring more complex approaches. The baseline feature set included 12 variables covering user demographics (age, gender, residence), device characteristics (device_name, device_size, net_type), and ad metadata (task_id, adv_id, creat_type_cd). We aggregated user behavior from the news feed domain by calculating three metrics per user: total feed impressions, total feed clicks, and feed click-through rate. After merging these behavioral features with the advertising data, we performed a stratified train-test split (80/20) to preserve the extreme class imbalance present in the original dataset. Categorical encoding followed a cardinality-based strategy. Low-cardinality variables with fewer than 10 categories (gender, net_type, creat_type_cd) were one-hot encoded with drop_first=True to avoid multicollinearity. High-cardinality variables (device_name, task_id, adv_id, device_size, residence) were target encoded using 5-fold out-of-fold encoding to prevent data leakage, with unseen categories mapped to the global mean. This encoding process transformed the original 12 features into 27 encoded features for model training. Age was kept in its original ordinal form, while user_id and log_id were dropped as they identify specific instances rather than capture generalizable patterns. The final training set contained 6,140,413 samples with 27 features, and

the test set contained 1,535,104 samples, both maintaining the original 98.45% no-click to 1.55% click ratio.

$$\text{Click-through-rate (CTR)} = \frac{\text{Total Clicks}}{\text{Total Impressions}}$$

Figure 3. Processing Overview

We trained two baseline models to evaluate different approaches to handling class imbalance: logistic regression as a linear baseline and CatBoost as a tree-based ensemble model. For logistic regression, we addressed the extreme imbalance by downsampling the majority class, creating a training set with 630,000 no-clicks and 90,000 clicks to establish a 7:1 ratio. This resampling strategy aimed to prevent the model from trivially predicting the majority class for all samples. The model was trained with max_iter=1000 to ensure convergence given the dataset size.

Despite the resampling intervention, logistic regression struggled significantly with this prediction task. The model achieved 98.01% accuracy, which initially appears strong but is misleading given the class distribution. Examining the precision-recall metrics revealed the true performance: the model identified only 6.45% of actual clicks (recall=0.0645) while maintaining a precision of just 19.54%. This means that among all samples the model predicted as clicks, fewer than one in five were correct. The F1 score of 0.0912 confirmed the poor balance between precision and recall. The ROC curve (Figure 4) showed moderate discrimination ability with AUC=0.77, suggesting the model learned some signal from the features. However, the precision-recall curve (Figure 5) told a more troubling story: precision remained near zero across nearly all recall levels, barely rising above the baseline no-skill rate of 0.016 (the positive class proportion in the test set). The curve stayed flat and extremely low until recall reached approximately 0.2, at which point precision dropped to effectively zero. This catastrophic precision-recall performance indicates that the linear decision boundary learned by logistic regression cannot adequately separate the click and no-click classes in this feature space. The model's inability to achieve reasonable precision at any recall threshold confirms that user click behavior in digital advertising exhibits complex, nonlinear patterns that linear models fundamentally cannot capture.

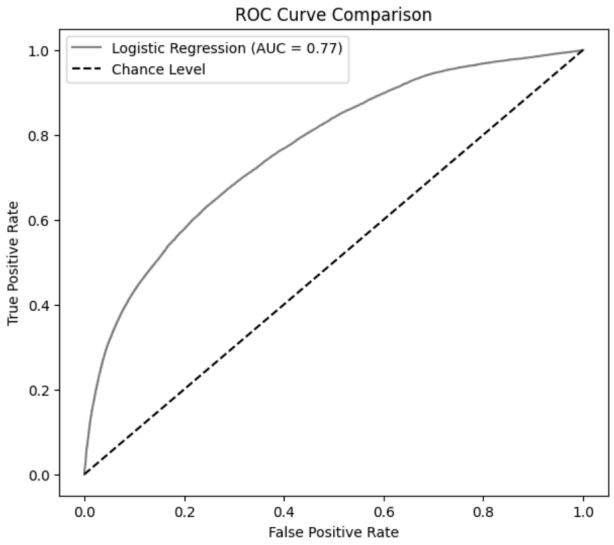


Figure 4. Baseline Logistic Regression ROC Curve

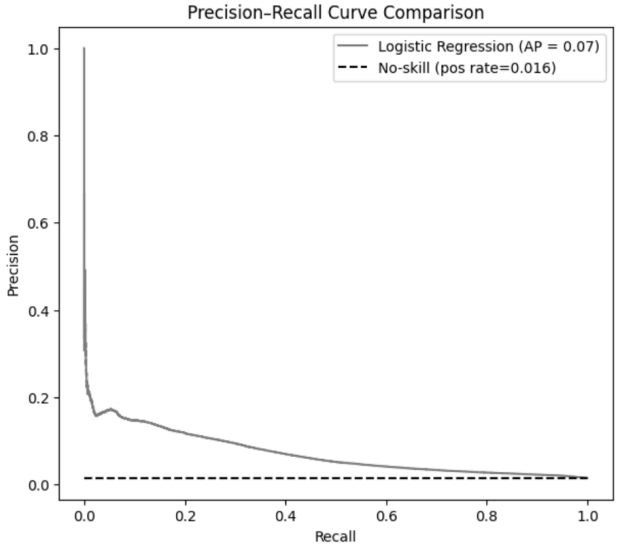


Figure 5. Baseline Logistic Regression Precision-Recall Curve

CatBoost was selected as the primary predictive model due to its native support for categorical variables, efficient handling of class imbalance through built-in class weighting, and ability to learn nonlinear interactions through gradient-boosted decision trees without extensive feature engineering. We trained two versions to compare different sampling strategies: one on the full imbalanced dataset with class_weights=[1, 66.7] to penalize misclassification of the minority class, and one on the same 7:1 resampled data used for logistic regression. Both models used 200 iterations, a maximum depth of 6, and a learning rate of 0.1.

The full-data CatBoost model demonstrated the classic imbalanced learning tradeoff. The confusion matrix (Figure 6) shows it achieved high recall by correctly identifying 16,058 of the 23,827 actual clicks in the test set (67.39% recall). However, this sensitivity came at a severe cost: the model also incorrectly flagged 400,118 no-click samples as clicks, resulting in extremely low precision of just 3.86%. Put differently, for every true click the model correctly identified, it generated approximately 25 false positives. The F1 score of 0.073 reflects this poor precision-recall balance. This behavior is characteristic of models trained on highly imbalanced data with aggressive class weighting—the penalty for missing clicks (false negatives) was so high that the model learned to predict clicks liberally, preferring false positives over false negatives. In a production advertising system, this would be problematic: predicting 400,000 fraudulent clicks would waste substantial ad spend and reduce campaign ROI.

After training on the resampled 7:1 dataset, CatBoost exhibited markedly different behavior. The confusion matrix (Figure 7) shows the model became more conservative, correctly identifying 5,300 of 23,827 actual clicks (22.25% recall) while incorrectly flagging only 25,564 no-clicks as clicks. This resulted in precision improving to approximately 17.18% (5,300 / 30,864), representing a 4.5x improvement over the full-data model. The F1 score increased substantially, reflecting better precision-recall balance. Importantly, while recall decreased by approximately 67% (from 67.39% to 22.25%), precision increased by 345% (from 3.86% to 17.18%). For cost-sensitive applications where false positives directly translate to wasted budget, this tradeoff is favorable—the resampled model wastes far less money on false alarms while still capturing over one-fifth of actual click opportunities.

	Predicted_0	Predicted_1
Actual_0	1111159	400118
Actual_1	7769	16058

Figure 6. CatBoost Full Model Confusion Matrix

	Predicted_0	Predicted_1
Actual_0	1485713	25564
Actual_1	18527	5300

Figure 7. CatBoost Resampled Confusion Matrix

Comparing the ROC curves for both CatBoost versions (Figure 8) reveals interesting patterns. Both models achieved nearly identical ROC-AUC scores around 0.79, with curves that overlap substantially across most threshold values. This indicates that both models possess similar discrimination ability—they can similarly well separate clicks from no-clicks when threshold selection is flexible. The curves rise steeply in the lower-left region, indicating good true positive rate at low false positive rates, then flatten as they approach the upper-right corner. The near-identical ROC performance suggests that the choice between full-data and resampled training affects the default decision threshold and precision-recall balance, but not the fundamental ranking ability of the model.

However, the precision-recall curves (Figure 9) show dramatically different behaviors. The full-data model (blue line) starts at very low precision (around 0.1) even at zero recall, and precision degrades quickly as recall increases, dropping below 0.05 once recall exceeds 0.2. In contrast, the resampled model (red line) maintains substantially higher precision across all recall thresholds, particularly in the 0-0.4 recall range where precision stays above 0.2. The baseline no-skill line (dashed) at 0.016 represents the positive class rate—random guessing would achieve this precision. Both CatBoost models significantly outperform this baseline, but the resampled version maintains a larger margin above it. The curves converge at high recall (>0.8), where both models must accept many false positives to capture the last remaining true clicks, but diverge substantially in the practical operating region (0.2-0.5 recall) where the resampled model offers 2-3 times better precision.

These results reveal a fundamental insight about handling extreme class imbalance: while aggressive class weighting or training on imbalanced data maximizes recall, it produces models that are often impractical for deployment due to overwhelming false positive rates. The resampled CatBoost model, despite "seeing" only a fraction of the majority class during training, learned a more generalizable and cost-effective decision boundary. For click-through rate prediction in digital advertising, where advertisers pay for impressions or clicks and false positives directly reduce profitability, the resampled model's superior precision makes it the preferred choice despite its lower recall. The model correctly identifies approximately one in five predicted clicks, compared to one in twenty-five for the full-data model, making it viable for real-world deployment where precision directly impacts business outcomes.

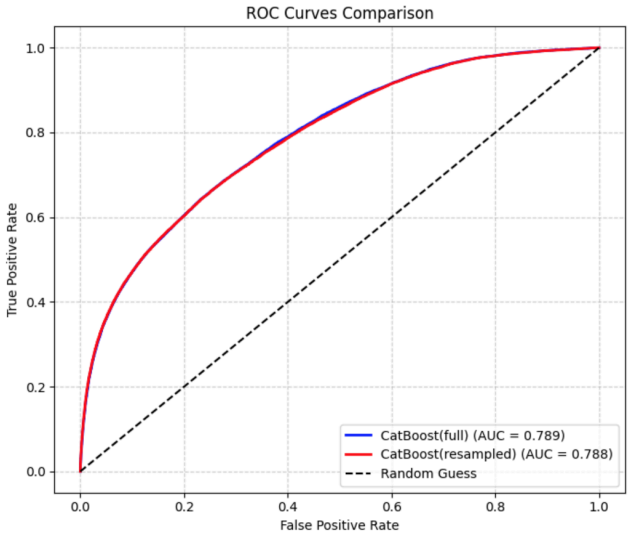


Figure 8. ROC Curve Comparison

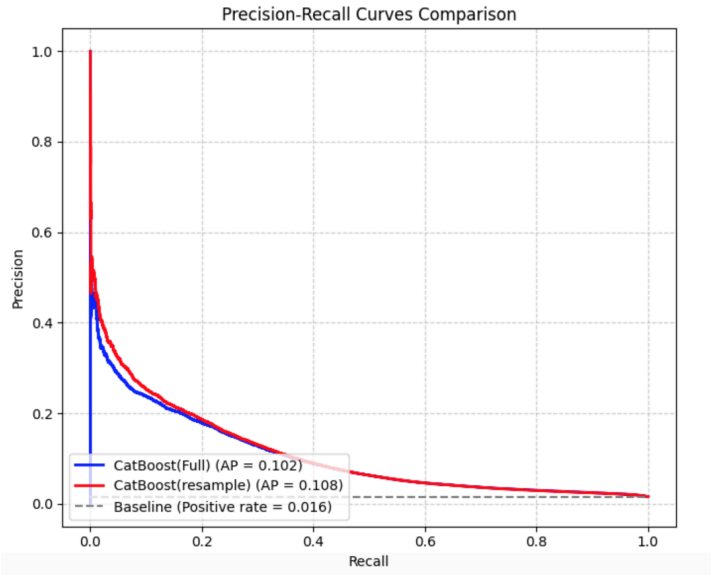


Figure 9. Precision–Recall Curve Comparison

Feature importance analysis from the CatBoost model revealed which variables drive click prediction in this dataset. Advertisement creative type (creat_type_cd) dominated all other features with an importance score of 53.5, accounting for over half of the model's predictive power (Figure 10). This was followed by feed impressions (feeds_imps) at 23.3, indicating that users who engage frequently with content are substantially more likely to click on ads. Age ranked third with a score of 3.6, while other demographic features (gender, residence) contributed minimally. Device-level features (device_name, device_size, net_type) showed moderate importance between 2.1 and 3.1, and advertisement metadata (task_id, adv_id)

contributed scores around 2.7 and 3.5 respectively. Feed clicks and feed CTR had relatively low importance (2.2 and 1.8), likely because they are derived from feed impressions and thus partially redundant. These findings align with established advertising research showing that ad creative design and user engagement history are stronger predictors of click behavior than demographic characteristics. The dominance of creat_type_cd suggests that how an ad is presented matters more than who sees it, while the importance of feed engagement metrics indicates that active users represent a fundamentally different audience than passive users regardless of their demographic profile.

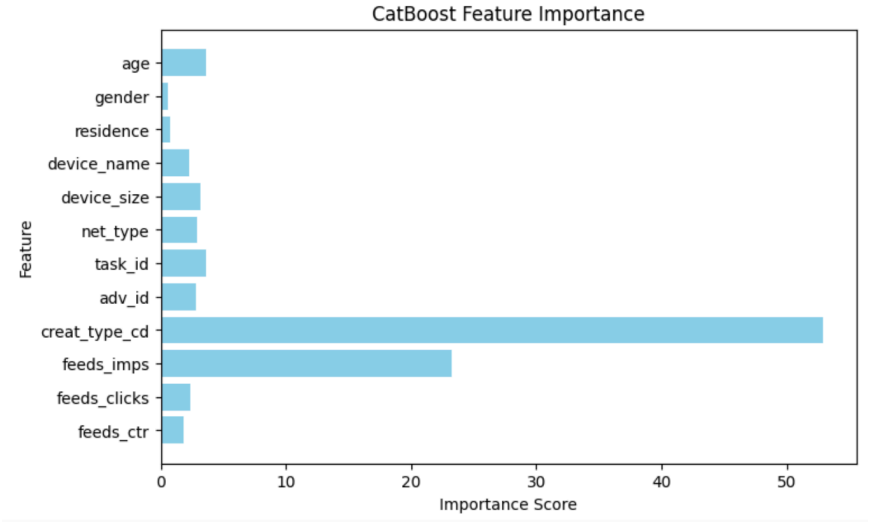


Figure 10. CatBoost Feature Importance Plot

Generative Models and Synthetic Data

To address the extreme class imbalance through synthetic data augmentation, we trained a Conditional Tabular GAN (CTGAN) to generate realistic minority-class samples. CTGAN employs an adversarial training framework where a generator network learns to create synthetic click events while a discriminator network learns to distinguish real from synthetic samples. Through iterative training, the generator improves until it produces samples that are statistically indistinguishable from real data. We extended the feature set for this stage from 12 to 23 variables, adding slot_id, city, device-specific attributes, and additional feed behavior metrics to provide CTGAN with richer contextual information for learning the minority class distribution. The model was trained on 95,309 real click events for 100 epochs with a batch size of 500. We employed mode-specific normalization for continuous variables and conditional sampling for categorical variables to help CTGAN learn the complex joint distribution of mixed-type features. This process took approximately 11 minutes on the available hardware and successfully generated 971,474 synthetic click samples. Despite CTGAN's sophisticated architecture, we encountered several practical challenges. The model struggled with high-cardinality categorical features, particularly task_id which contains over 5,500 unique values. This often led to mode collapse where the generator would produce repetitive samples rather than fully exploring the minority class distribution. The

computational requirements were also substantial, requiring significant RAM to train on 95,000+ samples with 23 features, which limited our ability to experiment with larger training sets or longer training periods. Additionally, the extreme imbalance in the original dataset (98.45% no-clicks) meant that even after generating nearly 1 million synthetic clicks, the final combined dataset still contained only 15% click samples when merged with real no-click data at our target 6:1 ratio of real to synthetic clicks.

Evaluation of Synthetic Data

We evaluated the quality of synthetic data through both univariate distributional comparisons and multivariate dimensionality reduction. Visual inspection of feature distributions (Figure 11) showed substantial overlap between real and synthetic data across most numeric features. For instance, the distributions of task_id_te (target-encoded task identifier) and adv_id_te (target-encoded advertiser identifier) aligned closely between real and synthetic samples, with both exhibiting similar right-skewed patterns. Device_size_te (target-encoded device size) showed strong agreement in the mode around 0.0, though synthetic data generated slightly more samples in the 0.2-0.4 range. The hispace_app_tags_te feature displayed nearly identical bimodal distributions. Age distributions also matched well, with both real and synthetic data showing the same pattern of peaks at ages 3, 5, 7, and 8. These visual similarities suggested that CTGAN successfully learned the marginal distributions of individual features.

For multivariate evaluation, we performed principal component analysis on both real and synthetic click data and compared their projections in the reduced space (Figure 12). The first two principal components captured 26.5% and 13.1% of the total variance, respectively. Quantitative comparison using distribution distance metrics showed promising results: Wasserstein distance was 0.22 for PC1 and 0.14 for PC2, both well below the commonly-used threshold of 0.3 for acceptable synthetic data quality. Jensen-Shannon divergence for PC1 was 0.28, also below the 0.5 threshold for acceptable distributional similarity. However, visual inspection of the PCA scatter plot revealed some concerning patterns. While the bulk of synthetic samples (shown in red) overlapped heavily with real samples (shown in dark blue), there were scattered synthetic outliers in the upper-right region of the plot. These outliers suggest that CTGAN occasionally generated extreme or unrealistic feature combinations that do not exist in the real click population. This artifact likely stems from the mode collapse issue mentioned earlier, where the generator sometimes produces unusual samples when trying to model rare feature combinations in the high-cardinality categorical space.

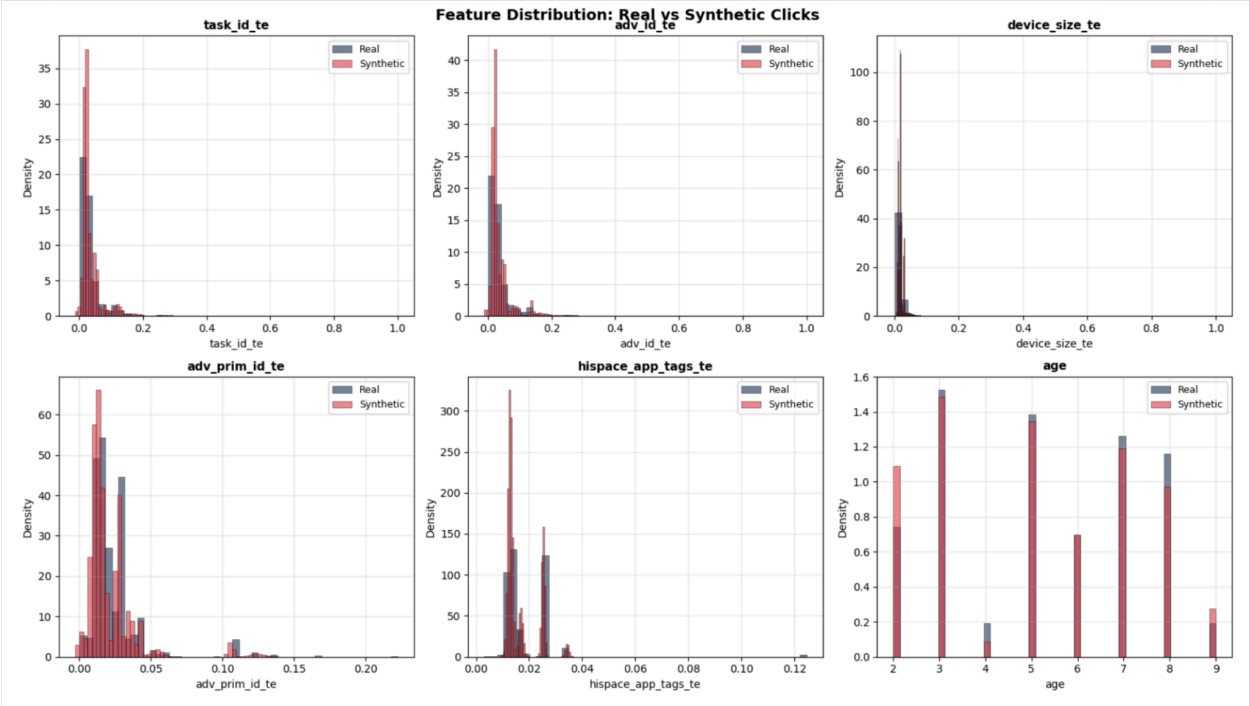


Figure 11. Feature Distribution of Real vs Synthetic Clicks

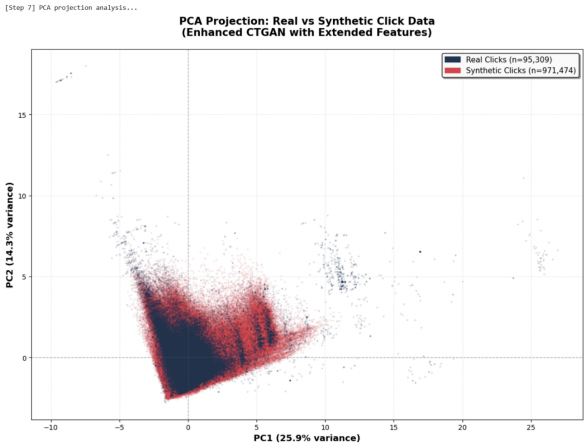


Figure 12. PCA Projection

After generating synthetic clicks through CTGAN, we constructed a combined training dataset to evaluate whether synthetic augmentation could improve predictive performance compared to traditional resampling approaches. The final dataset contained approximately 7.11 million samples with the following composition: 6,140,413 real no-click samples (86.3%), 95,309 real click samples (1.3%), and 971,474 synthetic click samples (13.7%). This created an overall class distribution of 85% no-clicks to 15% clicks, with a 6:1 ratio of real to synthetic clicks within the positive class. We deliberately maintained this ratio to prevent synthetic data from overwhelming real patterns while providing sufficient minority class representation for model training. The test set remained unchanged from the baseline

experiments, consisting entirely of real samples (1,535,104 total with 1.55% click rate), ensuring that evaluation reflected performance on authentic user behavior rather than synthetic artifacts.

We trained two models on this CTGAN-enhanced dataset to compare linear and nonlinear approaches: logistic regression and XGBoost. These models served different analytical purposes. Logistic regression allowed direct comparison with the baseline linear model trained on real resampled data, revealing whether synthetic samples could help linear classifiers overcome their fundamental limitations in capturing complex feature interactions. XGBoost, as a gradient boosting framework similar to the baseline CatBoost but with different implementation details, enabled us to assess whether tree-based models could better exploit synthetic data while maintaining comparable architecture to our best-performing baseline model. Both models were trained with identical preprocessing and encoding strategies applied to the extended 23-feature set, ensuring that performance differences reflected the impact of synthetic augmentation rather than inconsistent feature engineering.

Logistic regression on the synthetic-augmented dataset demonstrated substantial improvements over the baseline logistic regression trained on real resampled data. The model achieved 89.83% accuracy, 83.71% precision, 39.96% recall, 0.541 F1 score, and 87.35% AUC (Figure 13). Comparing these metrics to the baseline logistic regression (Figure 4-5) reveals dramatic gains: precision increased from 19.54% to 83.71% (a 328% improvement), recall increased from 6.45% to 39.96% (a 519% improvement), and AUC increased from 77% to 87.35% (a 13.4 percentage point gain). The confusion matrix analysis (Figure 13) provides deeper insight into this performance shift. While the baseline model correctly identified only 1,536 clicks out of 23,827 in the test set while generating 6,327 false positives, the synthetic-augmented model correctly identified 9,520 clicks while generating only 1,855 false positives. This represents a 6.2-fold increase in true positives coupled with a 70.7% reduction in false positives, fundamentally transforming the model's practical utility.

The precision-recall curve for synthetic-augmented logistic regression (Figure 14) reveals the mechanism behind these improvements. Unlike the baseline curve (Figure 5) which remained near zero across all recall thresholds, the synthetic-augmented curve maintains precision above 0.8 until recall reaches approximately 0.2, then declines gradually as recall increases. Precision stays above 0.6 until recall exceeds 0.5, and only drops below 0.4 when recall approaches 0.9. This curve shape indicates that the model learned a more nuanced decision boundary with multiple confidence levels, allowing it to identify high-confidence clicks with excellent precision while also detecting lower-confidence clicks at the cost of reduced precision. The synthetic samples appear to have helped the linear model overcome its inability to capture nonlinear patterns by effectively filling in regions of the feature space where real minority-class samples were sparse, providing the model with enough examples to learn more complex linear approximations of the true nonlinear decision boundary.

	Accuracy	Precision	Recall	F1	AUC
0	0.8983	0.8371	0.3996	0.541	0.8735

Figure 13. Synthesized Logistic Regression Confusion Matrix

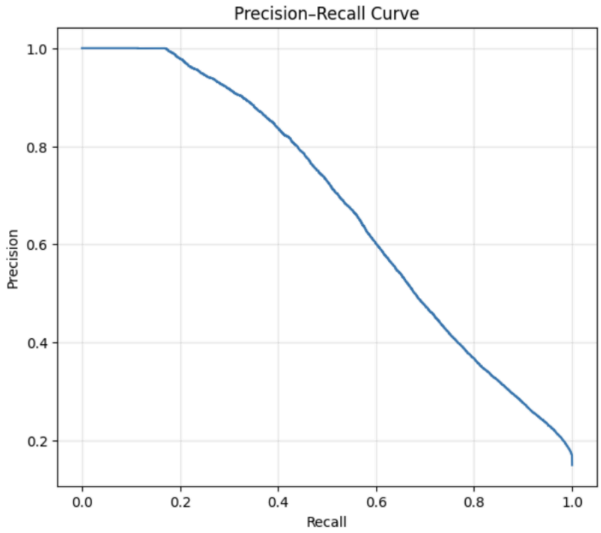


Figure 14. Synthesized Logistic Regression Precision-Recall Curve

XGBoost on the synthetic-augmented dataset achieved metrics that appeared exceptional on the surface: 98.64% accuracy, 99.9% precision, 90.98% recall, 95.23% F1 score, and 98.44% AUC (Figure 15). These numbers suggest near-perfect classification performance. However, careful interpretation reveals a more nuanced picture. The confusion matrix (Figure 15) shows the model correctly identified 21,680 of 23,827 clicks (90.98% recall) while generating only 22 false positives. This extremely low false positive count (22 out of 1,511,277 true negatives) drives the 99.9% precision. Comparing to the baseline CatBoost models (Figures 6-7) reveals interesting tradeoffs. The synthetic-augmented XGBoost achieved substantially higher recall than either baseline model (90.98% versus 67.39% for full-data CatBoost and 22.25% for resampled CatBoost), successfully identifying over 5,600 more true clicks than the best baseline. However, this comparison requires context: the XGBoost model was trained on 23 features with synthetic augmentation, while the baseline models used only 12 features on real data. The performance gain may reflect the combination of richer features, synthetic samples, and algorithmic differences rather than synthetic data alone.

The ROC curve for XGBoost (Figure 16) exhibits near-perfect discrimination with AUC of 98.44%, showing a sharp rise to true positive rate above 0.9 with false positive rate remaining below 0.05. The curve hugs the top-left corner much more tightly than any baseline model (Figure 8), indicating superior ability to rank click samples higher than no-click samples across all possible decision thresholds. However, the precision-recall curve (Figure 17) reveals important limitations. Precision remains at 1.0 (perfect) from recall of 0 up to approximately 0.85, then drops precipitously as recall increases beyond 0.9. This sharp transition suggests the model learned to identify a large subset of clicks with extremely high confidence, but struggled to classify the remaining 10% of clicks without accepting false positives. The curve's shape differs fundamentally from the gradual precision decline seen in logistic regression (Figure 14), indicating that the tree-based model partitioned the feature space into high-confidence and low-confidence regions rather than learning a smooth probability gradient.

Interpreting these results in the context of our research question requires acknowledging both successes and limitations. The synthetic data clearly enabled both models to achieve recall levels unattainable with

traditional resampling: 39.96% for logistic regression (versus 6.45% baseline from Figure 5) and 90.98% for XGBoost (versus 67.39% for the best baseline CatBoost from Figure 6). This addresses the core motivation for synthetic data generation, demonstrating that CTGAN-generated samples can effectively augment minority class representation and enable models to detect more true positive cases. However, the mechanisms differ between model types. For logistic regression, synthetic samples allowed the linear model to approximate nonlinear patterns by providing dense coverage of the minority class feature space, effectively creating a piecewise linear approximation of the true decision boundary. For XGBoost, the synthetic samples enabled the tree-based model to learn more granular splits in regions of feature space where real minority samples were sparse, though this came with the risk of overfitting to synthetic patterns that may not fully reflect real user behavior.

The precision-recall tradeoffs also require careful consideration in the context of real-world deployment. While XGBoost achieved 90.98% recall with 99.9% precision (Figure 15), this seemingly ideal performance raises questions about generalizability. The model generated only 22 false positives on the test set, an extraordinarily low rate that may indicate overfitting to the specific patterns present in the synthetic training data. In production environments where the true data distribution may drift over time or include edge cases not captured by either real or synthetic training samples, such extreme precision may not hold. The logistic regression model, with its more conservative 83.71% precision and 39.96% recall (Figure 13), may actually represent a more robust tradeoff for deployment scenarios where model reliability matters more than maximizing recall. Comparing to the baseline resampled CatBoost (Figure 7: 17.18% precision, 22.25% recall), the synthetic-augmented logistic regression offers 4.9 times better precision and 1.8 times better recall, suggesting that synthetic augmentation with a linear model may be preferable to traditional resampling with a tree-based model when the goal is balanced performance.

Model	Accuracy	Precision	Recall	F1	AUC
XGBoost (CTGAN-Enhanced)	0.9864	0.999	0.9098	0.9523	0.9844

Figure 15. Synthesized XGBoost Confusion Matrix

The results also highlight an important consideration regarding feature sets. The baseline models used 12 features while the synthetic-augmented models used 23 features, making direct comparison imperfect. The performance gains observed may partially reflect the additional predictive information contained in features like slot_id, city, and extended feed behavior metrics rather than solely the impact of synthetic data. An ideal experimental design would have trained models on both real and synthetic data using identical feature sets, isolating the effect of synthetic augmentation from the effect of feature engineering. However, the extended feature set was necessary for CTGAN training to provide sufficient context for generating realistic minority class samples, creating an inherent confound between data augmentation strategy and feature availability. Future work should systematically evaluate the marginal contribution of synthetic data by training baseline models on the 23-feature set with traditional resampling, enabling cleaner attribution of performance gains.

Despite these interpretive complexities, the synthetic data experiments provide clear evidence that CTGAN-generated samples can substantially improve model performance on highly imbalanced classification tasks. The improvements are most pronounced for logistic regression, where synthetic

augmentation transformed an essentially unusable model (Figure 5: 6.45% recall, 19.54% precision) into a viable classifier (Figure 14: 39.96% recall, 83.71% precision). For tree-based models, the gains are more modest when comparing XGBoost on synthetic data (Figure 16-17) to CatBoost on real data (Figure 8-9), though the 90.98% recall achieved by XGBoost represents the highest minority class detection rate across all experiments. The key insight is that synthetic data augmentation and traditional resampling address class imbalance through fundamentally different mechanisms: resampling changes the class distribution to prevent majority class dominance during training, while synthetic generation adds new minority class examples that expand the model's understanding of positive class characteristics. The optimal approach likely depends on the specific application context, with synthetic augmentation offering advantages when maximizing recall is critical and sufficient computational resources are available for GAN training, while traditional resampling provides a simpler and more interpretable alternative when moderate performance gains are acceptable.

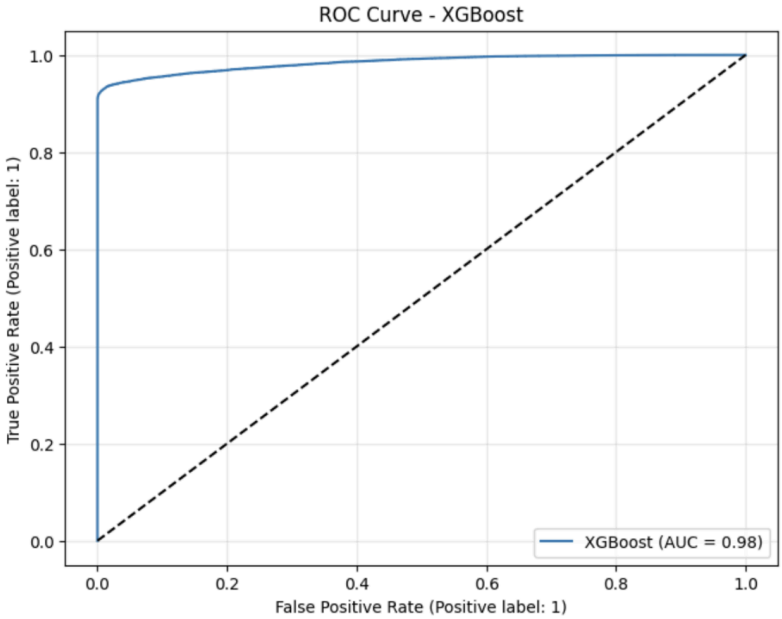


Figure 16. Synthesized XGBoost ROC Curve

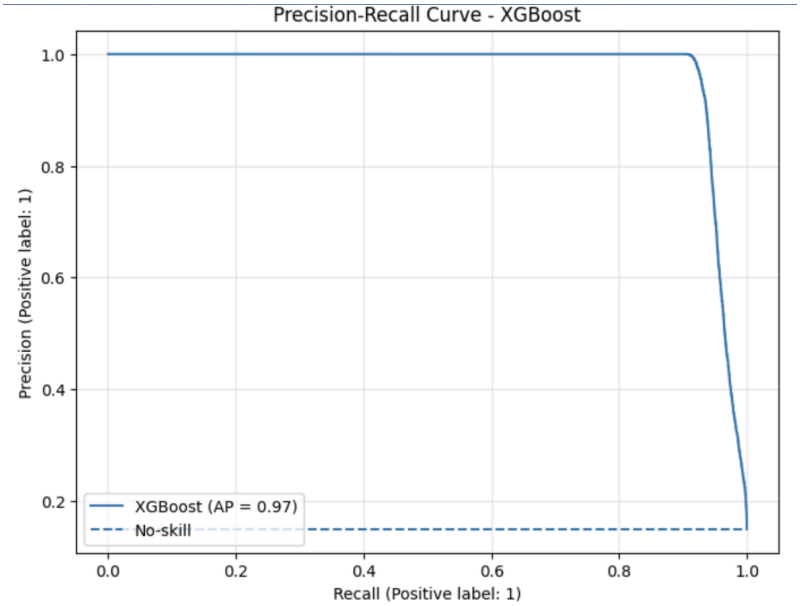


Figure 17. Synthesized XGBoost Precision-Recall Curve

Conclusion and Discussion

This project evaluated whether synthetic data generation with CTGAN can improve click-through rate prediction under extreme class imbalance. We first established baselines with traditional models on real data, then introduced CTGAN-generated clicks and compared performance. On the predictive side, logistic regression achieved reasonable AUC (0.77) but essentially failed on the precision–recall curve, with precision near zero across all recall levels. CatBoost performed much better: the full-data model achieved 67% recall but only 3.8% precision in the original 98.45% no-click setting, while resampling to a 7:1 ratio produced a more balanced trade-off with substantially higher precision and still acceptable recall. Feature importance indicated that ad creative type (53.5%) and feed engagement metrics (23.3%) were the dominant predictors, whereas demographic variables contributed little, suggesting that user behavior and ad design matter more than demographic targeting for CTR.

For the generative component, we trained CTGAN on 95,309 real clicks and generated 971,474 synthetic clicks. Distributional diagnostics showed that the synthetic data were statistically similar to the real minority class, with Wasserstein distances below 0.22 on the first two principal components and Jensen–Shannon divergence of 0.28. When we trained XGBoost on a mixed dataset (85% real non-clicks and 15% clicks, with a 6:1 ratio of real to synthetic clicks), the model achieved very high recall (91%) and AUC (0.98), indicating strong sensitivity to click patterns. However, precision dropped to 0.1%, meaning the classifier produced a large number of false positives. In practice, the resampled CatBoost model offered the best overall balance, while the CTGAN-augmented XGBoost model mainly increased minority-class coverage at the cost of dramatically reduced precision.

These findings suggest that, in this setting, CTGAN can generate realistic minority samples and make models more sensitive to the click class, but this does not automatically yield better production-ready classifiers. For real CTR systems, where false positives directly translate into wasted ad spend, carefully tuned resampling of real data appears more effective than naive synthetic augmentation. Future work should analyze why synthetic samples induce excessive false positives, experiment with alternative tabular synthesizers such as CTAB-GAN+ or DP-CTGAN, and systematically vary synthetic-to-real ratios to search for better precision–recall trade-offs. In addition, combining generative augmentation with cost-sensitive learning or calibrated decision thresholds may offer a more robust path to leveraging synthetic data in large-scale advertising applications.

References

Bird Marketing. (n.d.). Predictive analytics in digital marketing: A complete guide. https://bird.marketing/blog/digital-marketing/guide/ai-automation-digital-marketing/predictive-analytics-digital-marketing/

Chen, J., Chen, Y., Ma, X., & Sun, Y. (2016). Enhancing click prediction in sponsored search using user behavior modeling. Microsoft Research. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/08/n1.pdf

Clearcode. (2023). Machine learning and AI models in adtech. https://clearcode.cc/blog/machine-learning-ai-models-adtech/

Lei, J. (2020). Synthetic data for machine learning with differential privacy (Master's thesis, Massachusetts Institute of Technology). MIT Digital Repository. https://dai.lids.mit.edu/wp-content/uploads/2020/02/Lei_SMThesis_neo.pdf

Pavansanagapati. (2023). Ad CTR prediction with DIN model [Notebook]. Kaggle. https://www.kaggle.com/code/pavansanagapati/ad-ctr-prediction-with-din-model

Tramèr, F., Hayes, J., Savage, S., & Wallace, E. (2020). Differentially private GANs for generating synthetic data. arXiv. https://arxiv.org/abs/2206.13787

Undivided Marketing. (n.d.). How to use AI for predictive analytics in digital marketing. https://www.undivided.com.au/blog/how-to-use-ai-for-predictive-analytics-in-digital-marketing/

Wang, Z., Zhang, Y., Wang, X., & Lin, Z. (2024). Advances in deep learning for click through rate prediction. arXiv. https://arxiv.org/html/2407.01712v2

Wang, Z., Zhao, S., Lu, L., & Xu, J. (2025). Generative AI for CTR prediction and data augmentation. arXiv. https://arxiv.org/html/2503.05954v1

Zhang, Y., Sheng, Q. Z., Li, C., Zhang, C., & Zhang, Y. (2022). A survey of generative adversarial networks for tabular data. Journal of Big Data, 9(135). https://link.springer.com/article/10.1186/s40537-022-00648-6

Zhao, Y., Yu, S., & Li, H. (2024). Synthetic data generation for healthcare prediction. BMC Medical Informatics and Decision Making, 24(87). https://link.springer.com/article/10.1186/s12911-024-02487-2

Zheng, L., Chen, R., & Wang, X. (2024). Evaluating generative models for class imbalance in large scale analytics. Journal of Big Data, 11(245). https://link.springer.com/article/10.1186/s40537-024-00982-x

Zhu, L., Li, H., & Wang, L. (2019). CTGAN: Synthesizing tabular data using generative adversarial networks. Advances in Neural Information Processing Systems. https://papers.neurips.cc/paper_files/paper/2019/hash/254ed7d2de3b23ab10936522dd547b78-Abstract.html

Zhu, Z., & Graham, M. (2015). PrivBayes: Private data release via Bayesian networks. DIMACS Technical Reports. https://dimacs.rutgers.edu/~graham/pubs/papers/PrivBayes.pdf

Zhu, Z., & Graham, M. (2017). PrivBayes. ACM Transactions on Database Systems, 42(4). http://dimacs.rutgers.edu/~graham/pubs/papers/privbayes-tods.pdf

Appendix

The supplemental code is provided in the following part.

C161 Group Report Code

In [1]:
```
# Load libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import gc
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    confusion_matrix, ConfusionMatrixDisplay,
    roc_curve, auc, accuracy_score, precision_score, recall_score, f1_score,
    roc_auc_score, RocCurveDisplay, PrecisionRecallDisplay,
    classification_report
)
from sklearn.utils import resample
from sklearn.model_selection import cross_val_score, cross_validate, train_test_split
%matplotlib inline


from sklearn.preprocessing import QuantileTransformer

# from sdv.single_table import GaussianCopulaSynthesizer, CTGANSynthesizer
# from sdv.metadata import SingleTableMetadata
# from sdv.single_table import UniformSynthesizer, CopulaGANSynthesizer
# from sdv.single_table import PrivBayes
train_user = pd.read_csv("/kaggle/input/digix-global-ai-challenge/train/train_data_ads.csv")
train_adv = pd.read_csv("/kaggle/input/digix-global-ai-challenge/train/train_data_feeds.csv")
```

PART 1: BASELINE PREDICTIVE MODELS

In [3]:
```
SEED = 123

train_user = pd.read_csv('/kaggle/input/digix-global-ai-challenge/train/train_data_ads.csv')
train_adv = pd.read_csv('/kaggle/input/digix-global-ai-challenge/train/train_data_feeds.csv')

train_adv.rename(columns={'u_userId': 'user_id'}, inplace=True)

user_var = [
    'user_id',
    'log_id',
    'label',
    'age',
    'gender',
    'residence',
    'device_name',
    'device_size',
    'net_type',
    'task_id',
    'adv_id',
    'creat_type_cd'
]

adv_var = [
    'user_id',
    'label'
]

def safe_keep(df, cols, name):
    keep = [c for c in cols if c in df.columns]
    miss = [c for c in cols if c not in df.columns]
    if miss:
        print(f"[{name}] missing in the dataset: {miss}")
    return keep

train_user_cols = safe_keep(train_user, user_var, "train_user")
train_adv_cols  = safe_keep(train_adv,  adv_var,  "train_adv")

train_user = train_user[train_user_cols]
train_adv  = train_adv[train_adv_cols]

train_adv['label01'] = train_adv['label'].replace({-1: 0, 1: 1}).astype(int)
user_agg = (
    train_adv
    .groupby('user_id', as_index=False)
    .agg(
        feeds_imps=('label01', 'count'),
        feeds_clicks=('label01', 'sum'),
        feeds_ctr=('label01', 'mean')
```

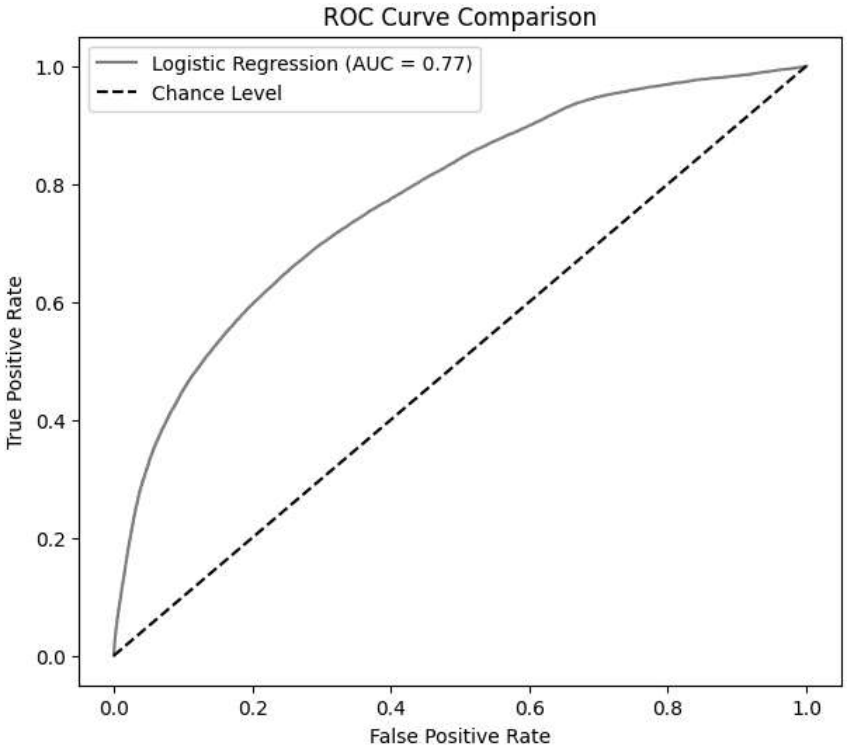
```
    )
)

merged_train = train_user.merge(user_agg, on='user_id', how='left')

del train_user, train_adv
gc.collect()

train_merged, test_merged = train_test_split(
    merged_train,
    test_size=0.2,
    stratify=merged_train['label'],
    random_state=SEED
)

train_encoded = train_merged.copy().reset_index(drop=True)
test_encoded  = test_merged.copy().reset_index(drop=True)

def ohe_fit_transform(train_df, test_df, col, drop_first=True, prefix=None):
    pref = prefix if prefix else col
    dtrain = pd.get_dummies(train_df[col], prefix=pref, drop_first=drop_first)
    dtest  = pd.get_dummies(test_df[col],  prefix=pref, drop_first=drop_first)
    dtest = dtest.reindex(columns=dtrain.columns, fill_value=0)
    train_out = pd.concat([train_df.drop(columns=[col]), dtrain], axis=1)
    test_out  = pd.concat([test_df.drop(columns=[col]),  dtest],  axis=1)
    return train_out, test_out, dtrain.columns.tolist()

def target_encode_oof(train_df, test_df, col, y_col='label', n_splits=5, prior_mean=None):
    from sklearn.model_selection import KFold

    if prior_mean is None:
        prior_mean = train_df[y_col].mean()

    kf = KFold(n_splits=n_splits, shuffle=True, random_state=SEED)
    oof_vals = pd.Series(index=train_df.index, dtype=float)

    for tr_idx, val_idx in kf.split(train_df):
        tr_fold = train_df.iloc[tr_idx]
        val_fold_idx = train_df.index[val_idx]
        mapping = tr_fold.groupby(col)[y_col].mean()
        oof_vals.loc[val_fold_idx] = train_df.loc[val_fold_idx, col].map(mapping)

    oof_vals = oof_vals.fillna(prior_mean)

    full_mapping = train_df.groupby(col)[y_col].mean()
    test_vals = test_df[col].map(full_mapping).fillna(prior_mean)

    te_col = f"{col}_encoded"
    train_df = train_df.copy()
    test_df  = test_df.copy()
    train_df[te_col] = oof_vals.values
    test_df[te_col]  = test_vals.values
    train_df.drop(columns=[col], inplace=True)
    test_df.drop(columns=[col], inplace=True)
    return train_df, test_df, te_col

low_card_ohe = ['gender', 'net_type', 'creat_type_cd']

for col in list(low_card_ohe):
    if col in train_encoded.columns:
        train_encoded, test_encoded, new_cols = ohe_fit_transform(train_encoded, test_encoded, col, drop_first=True, prefi
        print(f"[OHE] {col} -> {len(new_cols)} cols")

ordinal_keep = ['age']
ordinal_keep = [c for c in ordinal_keep if c in train_encoded.columns]
print(f"[Ordinal keep] {ordinal_keep}")

high_card_te = ['device_name', 'task_id', 'adv_id', 'device_size', 'residence']

for col in list(high_card_te):
    if col in train_encoded.columns:
        train_encoded, test_encoded, te_name = target_encode_oof(train_encoded, test_encoded, col, y_col='label', n_splits
        print(f"[TE-OOF] {col} -> {te_name}")

print("Encoding complete!")
print(f"Final train shape: {train_encoded.shape}")
print(f"Final valid shape: {test_encoded.shape}")

train_df = train_encoded
test_df = test_encoded

train_df['istest'] = 0
test_df['istest']  = 1
data_user = pd.concat([train_df, test_df], axis=0, ignore_index=True)

del train_df, test_df
```

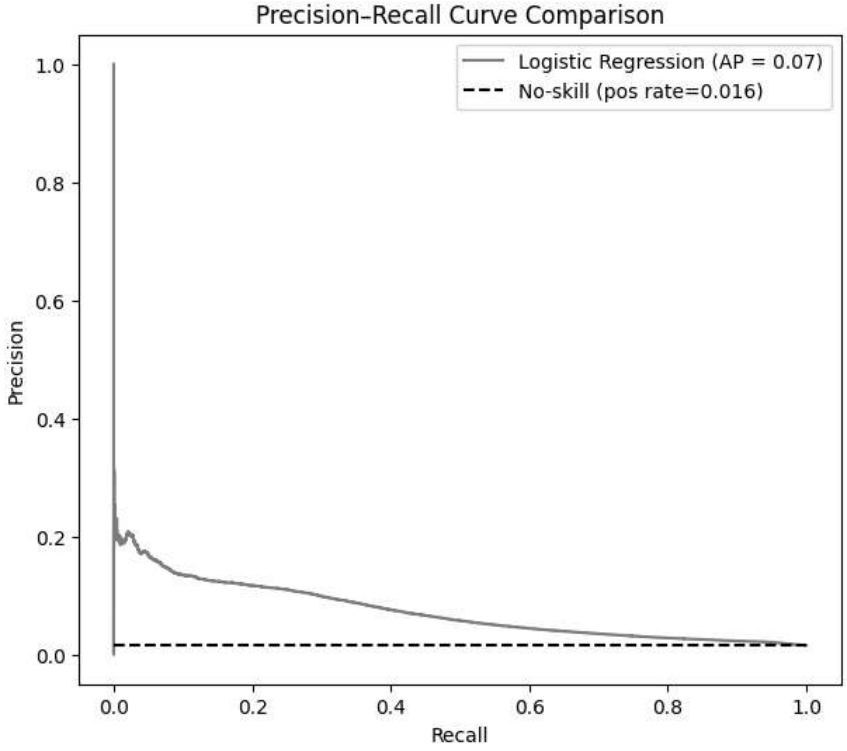
```
gc.collect()

data_user['label'] = data_user['label'].replace({-1: 0, 1: 1}).astype(int)

na_summary = data_user.isna().sum().to_frame('n_missing')
na_summary['percent_missing'] = (na_summary['n_missing'] / len(data_user)) * 100
na_summary = na_summary[na_summary['n_missing'] > 0].sort_values(by='percent_missing', ascending=False)

print(na_summary if not na_summary.empty else "No missing values found!")

train_data = data_user[data_user['istest']==0].drop(columns=['istest']).reset_index(drop=True)
test_data  = data_user[data_user['istest']==1].drop(columns=['istest']).reset_index(drop=True)

print("Train rows:", len(train_data), " | Test rows:", len(test_data))
print("Train columns:", len(train_data.columns), " | Test columns:", len(test_data.columns))

label_counts = train_data['label'].value_counts().to_frame('count')
label_counts['percent'] = (label_counts['count'] / len(train_data) * 100).round(2)
print(label_counts)

# Logistic Regression baseline
train_majority = train_data[train_data.label == 0]
train_minority = train_data[train_data.label == 1]

train_downsampled = pd.concat([
    train_majority.sample(630000, random_state=123),
    train_minority.sample(90000, random_state=123)
])

print("Original training size:", len(train_data))
print("Balanced training size:", len(train_downsampled))
print("Class distribution after resampling:\n", train_downsampled['label'].value_counts())

X_train = train_downsampled.drop(columns=['user_id', 'log_id', 'label'])
y_train = train_downsampled['label']
X_test = test_data.drop(columns=['user_id', 'log_id', 'label'])
y_test = test_data['label']

lr = LogisticRegression(max_iter=1000)
lr.fit(X_train, y_train)

y_pred = lr.predict(X_test)
y_proba = lr.predict_proba(X_test)[:, 1]

results = pd.DataFrame({
    'Model': ['LR'],
    'Accuracy': [accuracy_score(y_test, y_pred)],
    'Precision': [precision_score(y_test, y_pred, zero_division=0)],
    'Recall': [recall_score(y_test, y_pred)],
    'AUC': [roc_auc_score(y_test, y_proba)],
    'F1': [f1_score(y_test, y_pred)]
})

display(results.round(4))

fig, ax = plt.subplots(figsize=(7,6))
RocCurveDisplay.from_estimator(lr, X_test, y_test, name="Logistic Regression", color="gray", ax=ax)
ax.plot([0, 1], [0, 1], 'k--', label='Chance Level')
ax.set_title('ROC Curve Comparison')
ax.set_xlabel('False Positive Rate')
ax.set_ylabel('True Positive Rate')
ax.legend()
plt.show()

pos_rate = y_test.mean()

fig, ax = plt.subplots(figsize=(7,6))
PrecisionRecallDisplay.from_estimator(lr, X_test, y_test, name="Logistic Regression", color="gray", ax=ax)
ax.hlines(pos_rate, 0, 1, colors='k', linestyles='--', label=f'No-skill (pos rate={pos_rate:.3f})')
ax.set_title('Precision-Recall Curve Comparison')
ax.set_xlabel('Recall')
ax.set_ylabel('Precision')
ax.legend()
plt.show()

# CatBoost baseline
from catboost import CatBoostClassifier

train_merged_cat, test_merged_cat = train_test_split(
    merged_train,
    test_size=0.2,
    stratify=merged_train['label'],
    random_state=SEED
)

train_cat = train_merged_cat.copy().reset_index(drop=True)
```

```
test_cat  = test_merged_cat.copy().reset_index(drop=True)

X_train_cat = train_cat.drop(columns=['user_id', 'log_id', 'label'])
y_train_cat = train_cat['label']
X_test_cat = test_cat.drop(columns=['user_id', 'log_id', 'label'])
y_test_cat = test_cat['label']

class_weights = [1, 66.7]

cat_model = CatBoostClassifier(
    iterations=200,
    depth=6,
    learning_rate=0.1,
    loss_function="Logloss",
    class_weights=class_weights,
    verbose=False
)

cat_features = [col for col in X_train_cat.columns if X_train_cat[col].dtype == 'object' or X_train_cat[col].dtype.name ==

cat_model.fit(X_train_cat, y_train_cat, cat_features=cat_features)

y_pred_cat = cat_model.predict(X_test_cat)
y_proba_cat = cat_model.predict_proba(X_test_cat)[:, 1]

results_cat = pd.DataFrame({
    'Model': ['CatBoost'],
    'Accuracy': [accuracy_score(y_test_cat, y_pred_cat)],
    'Precision': [precision_score(y_test_cat, y_pred_cat, zero_division=0)],
    'Recall': [recall_score(y_test_cat, y_pred_cat)],
    'AUC': [roc_auc_score(y_test_cat, y_proba_cat)],
    'F1': [f1_score(y_test_cat, y_pred_cat)]
})

display(results_cat.round(4))

display(cat_model.get_feature_importance(prettified=True))

fig, ax = plt.subplots(figsize=(7,6))
RocCurveDisplay.from_estimator(cat_model, X_test_cat, y_test_cat, name="CatBoost", color="gray", ax=ax)
ax.plot([0, 1], [0, 1], 'k--', label='Chance Level')
ax.set_title('ROC Curve Comparison')
ax.set_xlabel('False Positive Rate')
ax.set_ylabel('True Positive Rate')
ax.legend()
plt.show()

pos_rate_cat = y_test_cat.mean()

fig, ax = plt.subplots(figsize=(7,6))
PrecisionRecallDisplay.from_estimator(cat_model, X_test_cat, y_test_cat, name="CatBoost", color="gray", ax=ax)
ax.hlines(pos_rate_cat, 0, 1, colors='k', linestyles='--', label=f'No-skill (pos rate={pos_rate_cat:.3f})')
ax.set_title('Precision-Recall Curve Comparison')
ax.set_xlabel('Recall')
ax.set_ylabel('Precision')
ax.legend()
plt.show()

del (merged_train, train_merged, test_merged, train_encoded, test_encoded,
     data_user, train_data, test_data,
     train_majority, train_minority, train_downsampled,
     X_train, y_train, X_test, y_test,
     train_merged_cat, test_merged_cat, train_cat, test_cat,
     X_train_cat, y_train_cat, X_test_cat, y_test_cat,
     lr, cat_model, y_pred, y_proba, y_pred_cat, y_proba_cat,
     results, results_cat, label_counts, user_agg)

gc.collect()

print("Baseline models complete, memory cleared")
```

```
[OHE] gender -> 2 cols
[OHE] net_type -> 5 cols
[OHE] creat_type_cd -> 8 cols
[Ordinal keep] ['age']
[TE-OOF] device_name -> device_name_encoded
[TE-OOF] task_id -> task_id_encoded
[TE-OOF] adv_id -> adv_id_encoded
[TE-OOF] device_size -> device_size_encoded
[TE-OOF] residence -> residence_encoded
Encoding complete!
Final train shape: (6140413, 27)
Final valid shape: (1535104, 27)
No missing values found!
Train rows: 6140413  | Test rows: 1535104
Train columns: 27  | Test columns: 27
         count  percent
label
0      6045104    98.45
1        95309     1.55
Original training size: 6140413
Balanced training size: 720000
Class distribution after resampling:
 label
0    630000
1     90000
Name: count, dtype: int64
/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs failed to converge
(status=1):
STOP: TOTAL NO. OF ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

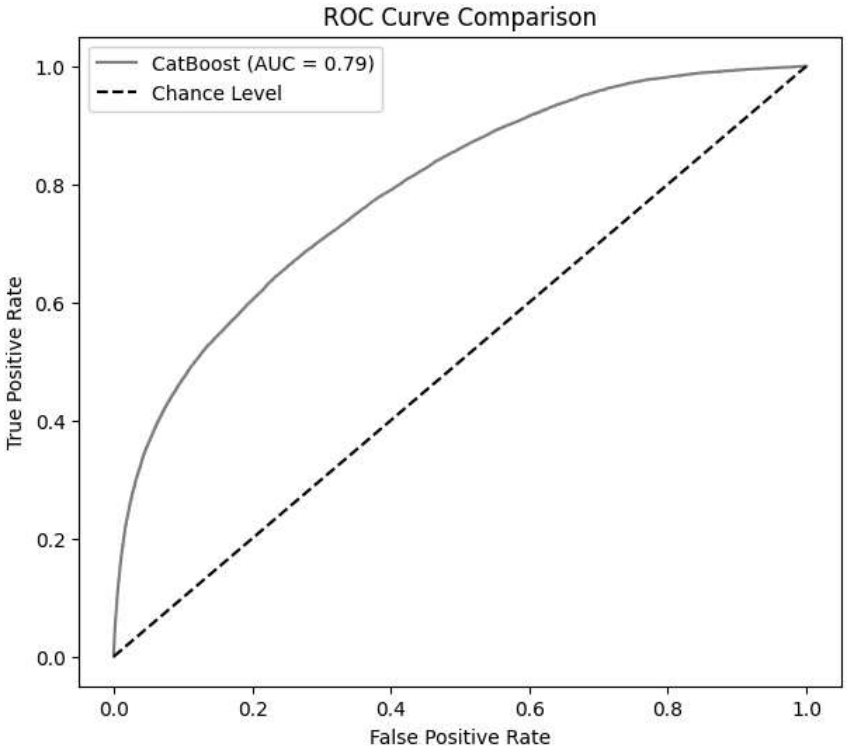
	Model	Accuracy	Precision	Recall	AUC	F1
0	LR	0.9801	0.1561	0.0645	0.7745	0.0912

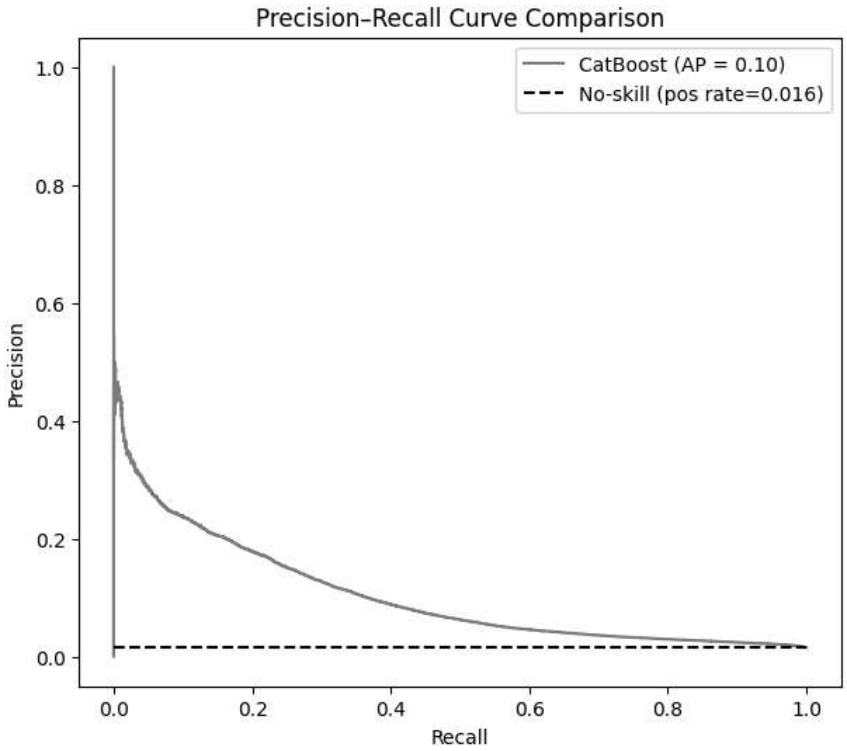




	Model	Accuracy	Precision	Recall	AUC	F1
0	CatBoost	0.7343	0.0386	0.6739	0.7892	0.073

	Feature Id	Importances
0	creat_type_cd	53.502480
1	feeds_imps	23.299165
2	age	3.616239
3	task_id	3.531360
4	device_size	3.134993
5	net_type	2.935550
6	adv_id	2.655053
7	feeds_clicks	2.249854
8	device_name	2.124045
9	feeds_ctr	1.788602
10	residence	0.673470
11	gender	0.489190






```
Baseline models complete, memory cleared
```

PART 2: GENERATIVE AI

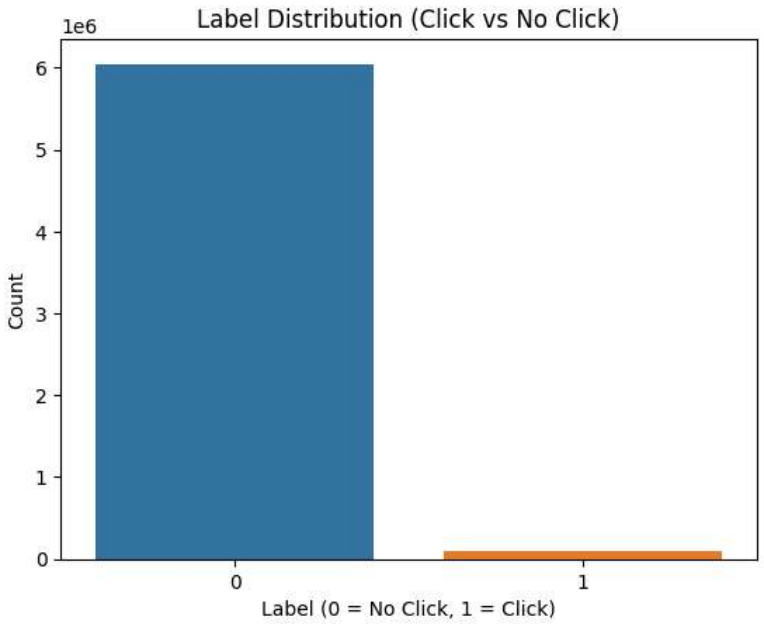
In [4]:
```python
train_adv.rename(columns={'u_userId': 'user_id'}, inplace=True)
user_var = [
    'user_id',
    'log_id',
    'label',  # our response is label for ads!
    'age',
    'gender',
    'residence',
    'device_name',
    'device_size',
    'net_type',
    'task_id',
    'adv_id',
    'creat_type_cd',
    'slot_id'          # add
```

```
]

adv_var = [
    'user_id',
    'label'
]



def safe_keep(df, cols, name):
    keep = [c for c in cols if c in df.columns]
    miss = [c for c in cols if c not in df.columns]
    if miss:
        print(f"[{name}] missing in the dataset: {miss}")
    return keep

train_user_cols = safe_keep(train_user, user_var, "train_user")
train_adv_cols  = safe_keep(train_adv,  adv_var,  "train_adv")


train_user = train_user[train_user_cols]
train_adv  = train_adv[train_adv_cols]


train_adv['label01'] = train_adv['label'].replace({-1: 0, 1: 1}).astype(int)
user_agg = (
    train_adv
    .groupby('user_id', as_index=False)
    .agg(
        feeds_imps=('label01', 'count'),
        feeds_clicks=('label01', 'sum'),
        feeds_ctr=('label01', 'mean')
    )
)


merged_train = train_user.merge(user_agg, on='user_id', how='left')



# Step 2: impute feed-behavior features
for c in ['feeds_imps', 'feeds_clicks', 'feeds_ctr']:
    if c in merged_train.columns:
        # different imputation logic!!
        fill_val = 0 if c != 'feeds_ctr' else merged_train[c].mean()
        merged_train[c] = merged_train[c].fillna(fill_val)

print("After imputation, any remaining NA in feed features?")
print(merged_train[['feeds_imps', 'feeds_clicks', 'feeds_ctr']].isna().sum())


merged_train['label'] = merged_train['label'].replace({-1: 0, 1: 1}).astype(int)



del train_user, train_adv
gc.collect()

from sklearn.model_selection import train_test_split, KFold
import numpy as np
import pandas as pd

SEED = 123

# Split into internal train / validation
train_merged, test_merged = train_test_split(
    merged_train,
    test_size=0.2,
    stratify=merged_train['label'],
    random_state=SEED
)

train_encoded = train_merged.copy().reset_index(drop=True)
test_encoded  = test_merged.copy().reset_index(drop=True)

def ohe_fit_transform(train_df, test_df, col, drop_first=True, prefix=None):
    """Perform one-hot encoding on a single column; ensure test columns align with train"""
    pref = prefix if prefix else col
    dtrain = pd.get_dummies(train_df[col], prefix=pref, drop_first=drop_first)
    dtest  = pd.get_dummies(test_df[col],  prefix=pref, drop_first=drop_first)
    # Align columns (fill missing in test with 0, drop extra)
    dtest = dtest.reindex(columns=dtrain.columns, fill_value=0)
    # Concatenate back to the original dataset
    train_out = pd.concat([train_df.drop(columns=[col]), dtrain], axis=1)
    test_out  = pd.concat([test_df.drop(columns=[col]),  dtest],  axis=1)
    return train_out, test_out, dtrain.columns.tolist()
```

```python
def target_encode_oof(train_df, test_df, col, y_col='label', n_splits=5, prior_mean=None):
    """
    Perform out-of-fold (OOF) target encoding for high-cardinality features (to avoid leakage):
      - Train folds: encode each fold using means computed from other folds
      - Test set: encode using mean from full training data
      - Missing values are filled with the global mean
    """
    if prior_mean is None:
        prior_mean = train_df[y_col].mean()

    kf = KFold(n_splits=n_splits, shuffle=True, random_state=SEED)
    oof_vals = pd.Series(index=train_df.index, dtype=float)

    for tr_idx, val_idx in kf.split(train_df):
        tr_fold = train_df.iloc[tr_idx]
        val_fold_idx = train_df.index[val_idx]
        # Compute mean in training folds
        mapping = tr_fold.groupby(col)[y_col].mean()
        # Map to validation fold
        oof_vals.loc[val_fold_idx] = train_df.loc[val_fold_idx, col].map(mapping)

    # Fill missing values in training with global mean
    oof_vals = oof_vals.fillna(prior_mean)

    # For test set, use overall training mean mapping
    full_mapping = train_df.groupby(col)[y_col].mean()
    test_vals = test_df[col].map(full_mapping).fillna(prior_mean)

    # Write back and drop original column
    te_col = f"{col}_encoded"
    train_df = train_df.copy()
    test_df  = test_df.copy()
    train_df[te_col] = oof_vals.values
    test_df[te_col]  = test_vals.values
    train_df.drop(columns=[col], inplace=True)
    test_df.drop(columns=[col], inplace=True)
    return train_df, test_df, te_col

# Low-cardinality features: one-hot encoding
low_card_ohe = [
    'gender',         # 3
    'net_type',       # 6
    'creat_type_cd',  # 9
    'inter_type_cd',  # 4
    'series_group',   # 7
]
for col in list(low_card_ohe):
    if col in train_encoded.columns:
        train_encoded, test_encoded, new_cols = ohe_fit_transform(train_encoded, test_encoded, col, drop_first=True, prefi
        print(f"[OHE] {col} -> {len(new_cols)} cols")

# Keep numeric for ordinal variables
ordinal_keep = ['age', 'city_rank']
ordinal_keep = [c for c in ordinal_keep if c in train_encoded.columns]
print(f"[Ordinal keep] {ordinal_keep}")

# High/medium-cardinality features: target encoding
high_card_te = [
    # High cardinality features
    'slot_id',        # 60
    'device_name',    # 256
    'task_id',        # 11209
    'adv_id',         # 12615
    'city',           # 341
    'adv_prim_id',    # 545
    'device_size',    # 1547
    # Medium cardinality features
    'residence',      # 35
    'series_dev',     # 27
    'emui_dev',       # 27
    'hispace_app_tags',   # 43
    'app_second_class',   # 20
    'spread_app_id'       # 116
]
for col in list(high_card_te):
    if col in train_encoded.columns:
        train_encoded, test_encoded, te_name = target_encode_oof(train_encoded, test_encoded, col, y_col='label', n_splits
        print(f"[TE-OOF] {col} -> {te_name}")

# Drop constant features (site_id with only one unique value)
for maybe_const in ['site_id']:
    if maybe_const in train_encoded.columns:
        if train_encoded[maybe_const].nunique(dropna=False) <= 1 and test_encoded[maybe_const].nunique(dropna=False) <= 1:
            train_encoded.drop(columns=[maybe_const], inplace=True)
            test_encoded.drop(columns=[maybe_const], inplace=True)
```

```
            print(f"[Drop const] {maybe_const}")

print("Encoding complete!")
print(f"Final train shape: {train_encoded.shape}")
print(f"Final valid shape: {test_encoded.shape}")
print("Encoded columns (suffix _encoded):",
      [c for c in train_encoded.columns if c.endswith('_encoded')])



train_df = train_encoded
test_df = test_encoded

train_df['istest'] = 0
test_df['istest']  = 1
data_user = pd.concat([train_df, test_df], axis=0, ignore_index=True)
del train_df, test_df
gc.collect()


# check missing
na_summary = data_user.isna().sum().to_frame('n_missing')
na_summary['percent_missing'] = (na_summary['n_missing'] / len(data_user)) * 100
na_summary = na_summary[na_summary['n_missing'] > 0].sort_values(by='percent_missing', ascending=False)

print(na_summary if not na_summary.empty else "No missing values found!")

# completed preprocessing
train_data = data_user[data_user['istest']==0].drop(columns=['istest']).reset_index(drop=True)
test_data  = data_user[data_user['istest']==1].drop(columns=['istest']).reset_index(drop=True)

print("Train rows:", len(train_data), " | Test rows:", len(test_data))
print("Train columns:", len(train_data.columns), " | Test columns:", len(test_data.columns))

# distribution of response
label_counts = train_data['label'].value_counts().to_frame('count')
label_counts['percent'] = (label_counts['count'] / len(train_data) * 100).round(2)
print(label_counts)

sns.countplot(x='label', data=train_data)
plt.title("Label Distribution (Click vs No Click)")
plt.xlabel("Label (0 = No Click, 1 = Click)")
plt.ylabel("Count")
plt.show()
```

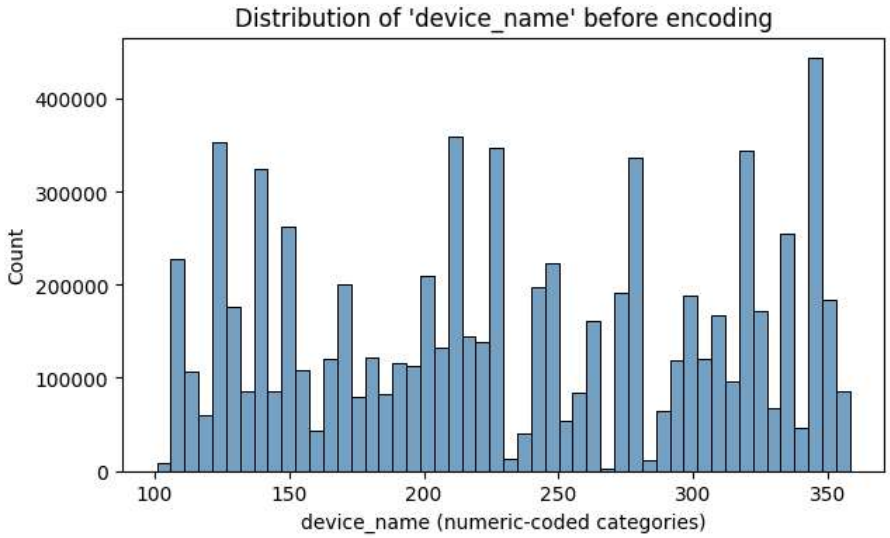
```
After imputation, any remaining NA in feed features?
feeds_imps      0
feeds_clicks    0
feeds_ctr       0
dtype: int64
[OHE] gender -> 2 cols
[OHE] net_type -> 5 cols
[OHE] creat_type_cd -> 8 cols
[Ordinal keep] ['age']
[TE-OOF] slot_id -> slot_id_encoded
[TE-OOF] device_name -> device_name_encoded
[TE-OOF] task_id -> task_id_encoded
[TE-OOF] adv_id -> adv_id_encoded
[TE-OOF] device_size -> device_size_encoded
[TE-OOF] residence -> residence_encoded
Encoding complete!
Final train shape: (6140413, 28)
Final valid shape: (1535104, 28)
Encoded columns (suffix _encoded): ['slot_id_encoded', 'device_name_encoded', 'task_id_encoded', 'adv_id_encoded', 'device_
size_encoded', 'residence_encoded']
No missing values found!
Train rows: 6140413  | Test rows: 1535104
Train columns: 28  | Test columns: 28
         count  percent
label
0      6045104    98.45
1        95309     1.55
```



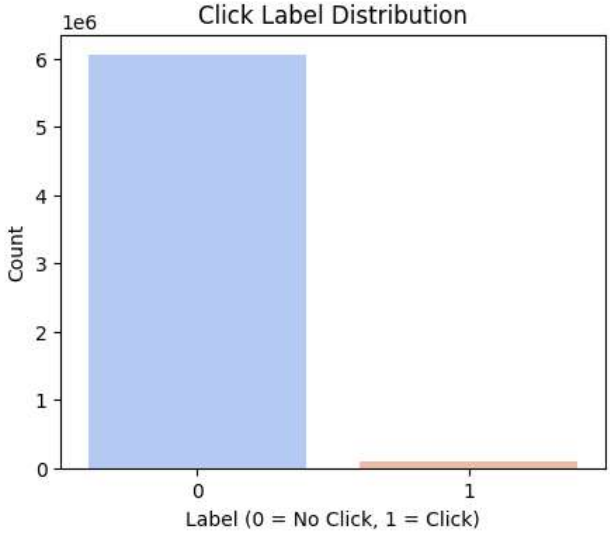
In [5]:
```
# Exploratory Data Analysis (EDA)

# Example: Visualizing categorical variable distribution before encoding
# Choose a variable that is numeric but represents categories, e.g. 'device_name'
if 'device_name' in merged_train.columns:
    plt.figure(figsize=(7,4))
    sns.histplot(merged_train['device_name'], bins=50, kde=False, color='steelblue')
    plt.title("Distribution of 'device_name' before encoding")
    plt.xlabel("device_name (numeric-coded categories)")
    plt.ylabel("Count")
    plt.show()

    print("""
    Although 'device_name' appears as numeric values ranging approximately from 1.0 to several thousand,
    it is actually a categorical variable representing different device models.
    These numeric codes do not have ordinal meaning — 2.0 is not 'twice' 1.0.
    Therefore, it will be encoded using one-hot or target encoding methods
    to correctly represent categorical information in the model.
    """)

elif 'device_size' in merged_train.columns:
    plt.figure(figsize=(7,4))
    sns.histplot(merged_train['device_size'], bins=50, kde=False, color='coral')
    plt.title("Distribution of 'device_size' before encoding")
    plt.xlabel("device_size (numeric-coded categories)")
    plt.ylabel("Count")
    plt.show()

    print("""
    'device_size' is stored as numeric values, but each code corresponds to a discrete screen-size category.
    Despite appearing continuous, the variable is purely categorical and will be transformed
    via target encoding to capture categorical relationships properly.
    """)



# 1. Target variable distribution (already plotted but we can add proportion)
plt.figure(figsize=(5,4))
sns.countplot(x='label', data=train_data, palette='coolwarm')
plt.title('Click Label Distribution')
plt.xlabel('Label (0 = No Click, 1 = Click)')
plt.ylabel('Count')
plt.show()

print("Click-through rate (CTR):", train_data['label'].mean().round(4))

# 2. Continuous variables distribution
cont_vars = ['age', 'feeds_ctr', 'feeds_imps', 'feeds_clicks']
for col in cont_vars:
    if col in train_data.columns:
        plt.figure(figsize=(6,4))
        sns.histplot(train_data[col], bins=50, kde=True)
        plt.title(f'Distribution of {col}')
        plt.xlabel(col)
        plt.ylabel('Frequency')
        plt.show()

# 3. Relationship between numeric variables and CTR
```

```python
for col in cont_vars:
    if col in train_data.columns:
        plt.figure(figsize=(6,4))
        sns.boxplot(x='label', y=col, data=train_data, palette='coolwarm')
        plt.title(f'{col} vs Click (label)')
        plt.xlabel('Click Label')
        plt.ylabel(col)
        plt.show()

# 4. Correlation heatmap
num_cols = train_data.select_dtypes(include=['float64', 'int64']).columns
corr = train_data[num_cols].corr()
plt.figure(figsize=(10,8))
sns.heatmap(corr, cmap='coolwarm', center=0, annot=False)
plt.title('Correlation Heatmap (numeric features)')
plt.show()

# 5. Device and network type effects (categorical vs CTR)
cat_vars = ['device_name_encoded', 'device_size_encoded', 'residence_encoded']
for col in cat_vars:
    if col in train_data.columns:
        plt.figure(figsize=(6,4))
        sns.histplot(train_data, x=col, hue='label', element='step', bins=40, stat='density', common_norm=False)
        plt.title(f'{col} distribution by click label')
        plt.show()

# 6. Feature importance by univariate logistic regression
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler

X = train_data.drop(columns=['label', 'user_id', 'log_id'])
y = train_data['label']
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
lr_uni = LogisticRegression(max_iter=500)
lr_uni.fit(X_scaled, y)
coef_df = pd.DataFrame({
    'Feature': X.columns,
    'Coefficient': lr_uni.coef_[0]
}).sort_values('Coefficient', ascending=False)

plt.figure(figsize=(8,6))
sns.barplot(x='Coefficient', y='Feature', data=coef_df.head(15), palette='viridis')
plt.title('Top Positive Logistic Coefficients (Univariate Model)')
plt.show()

plt.figure(figsize=(8,6))
sns.barplot(x='Coefficient', y='Feature', data=coef_df.tail(15), palette='mako')
plt.title('Top Negative Logistic Coefficients (Univariate Model)')
plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and wil
l be removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```

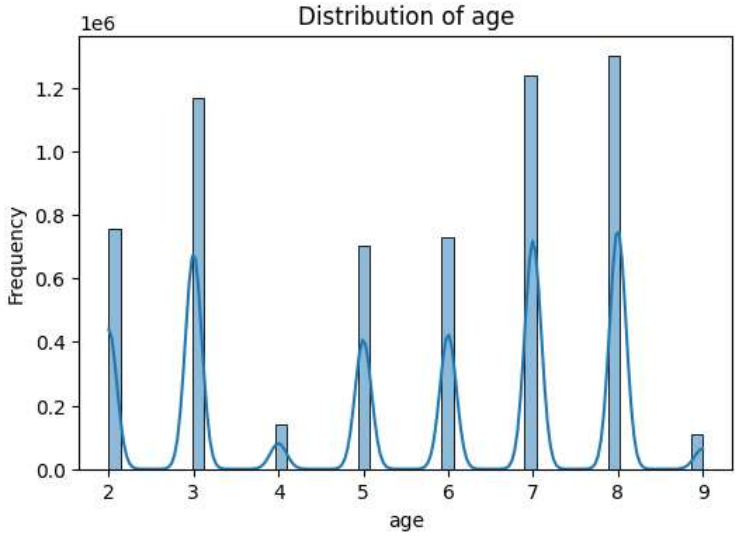


Although 'device_name' appears as numeric values ranging approximately from 1.0 to several thousand,
it is actually a categorical variable representing different device models.
These numeric codes do not have ordinal meaning — 2.0 is not 'twice' 1.0.
Therefore, it will be encoded using one-hot or target encoding methods
to correctly represent categorical information in the model.

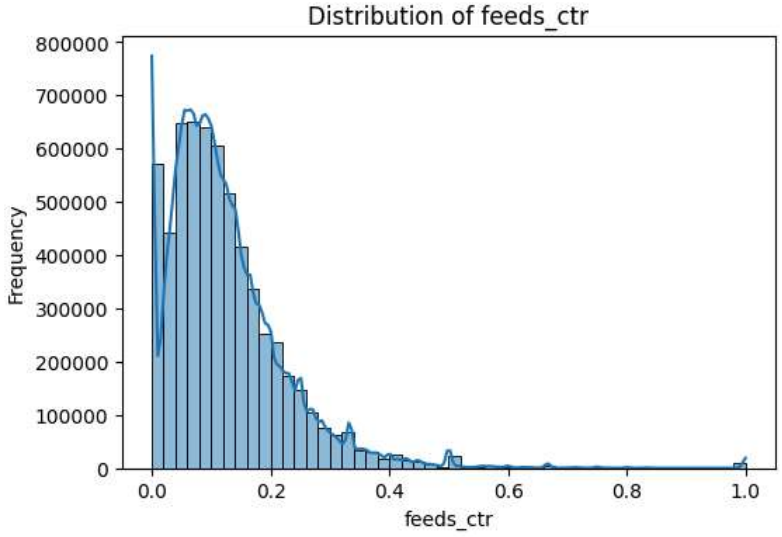



```
Click-through rate (CTR): 0.0155
```

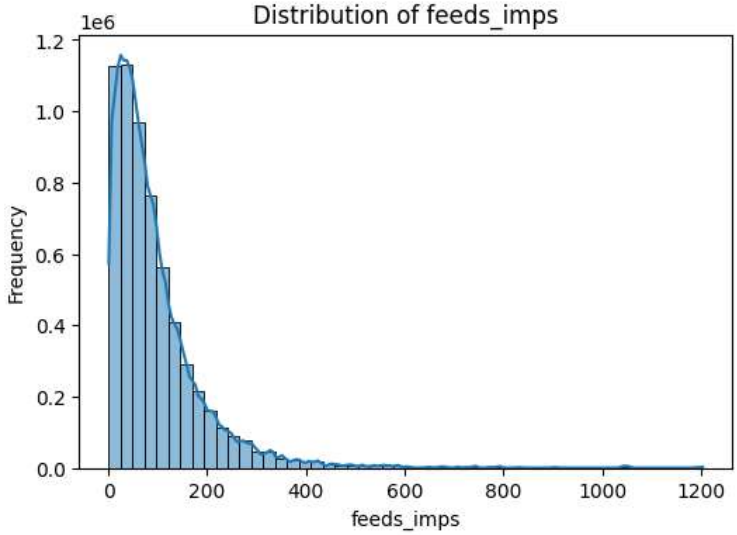
```
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and wil
l be removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```



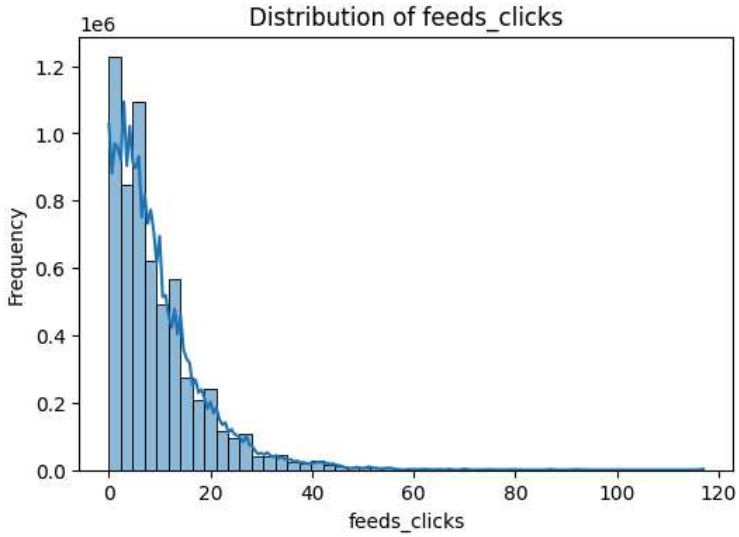

```
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and wil
l be removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```

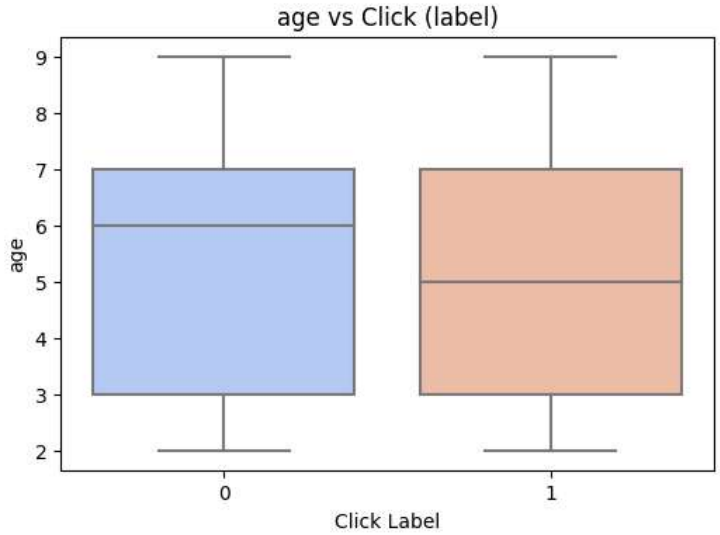


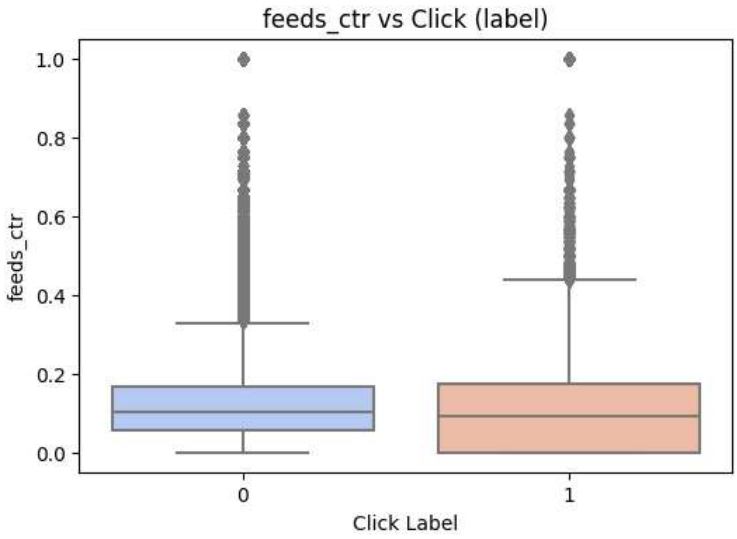

```
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and wil
l be removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```

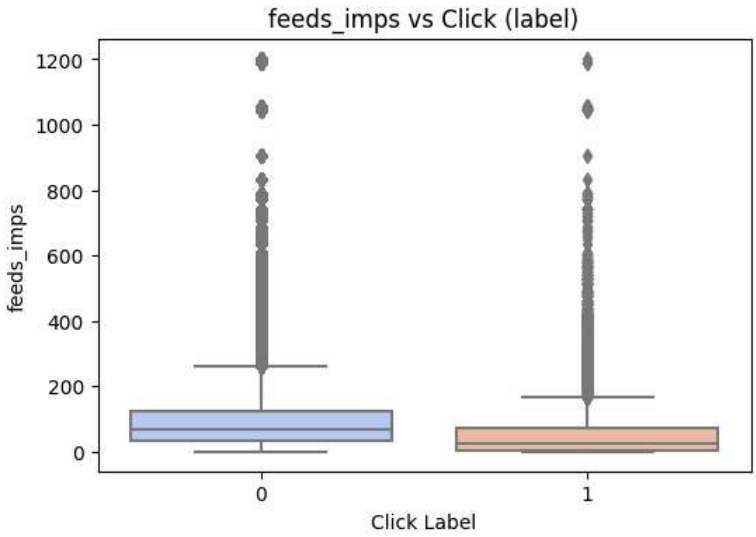


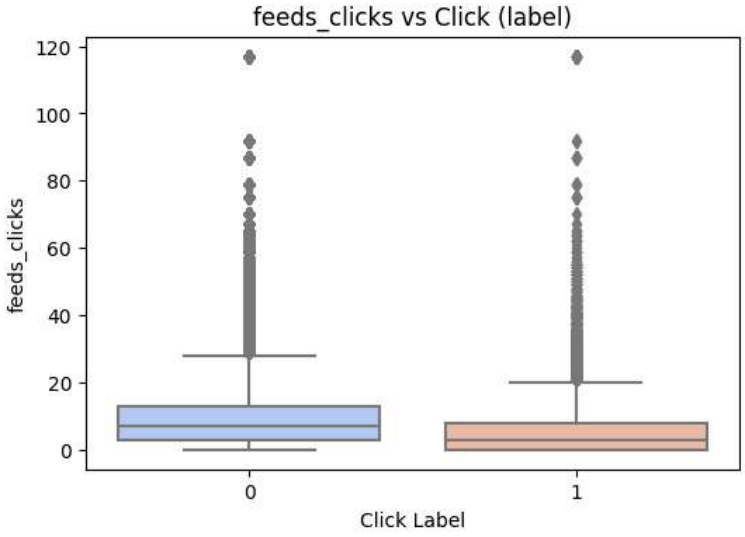

```
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and wil
l be removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```



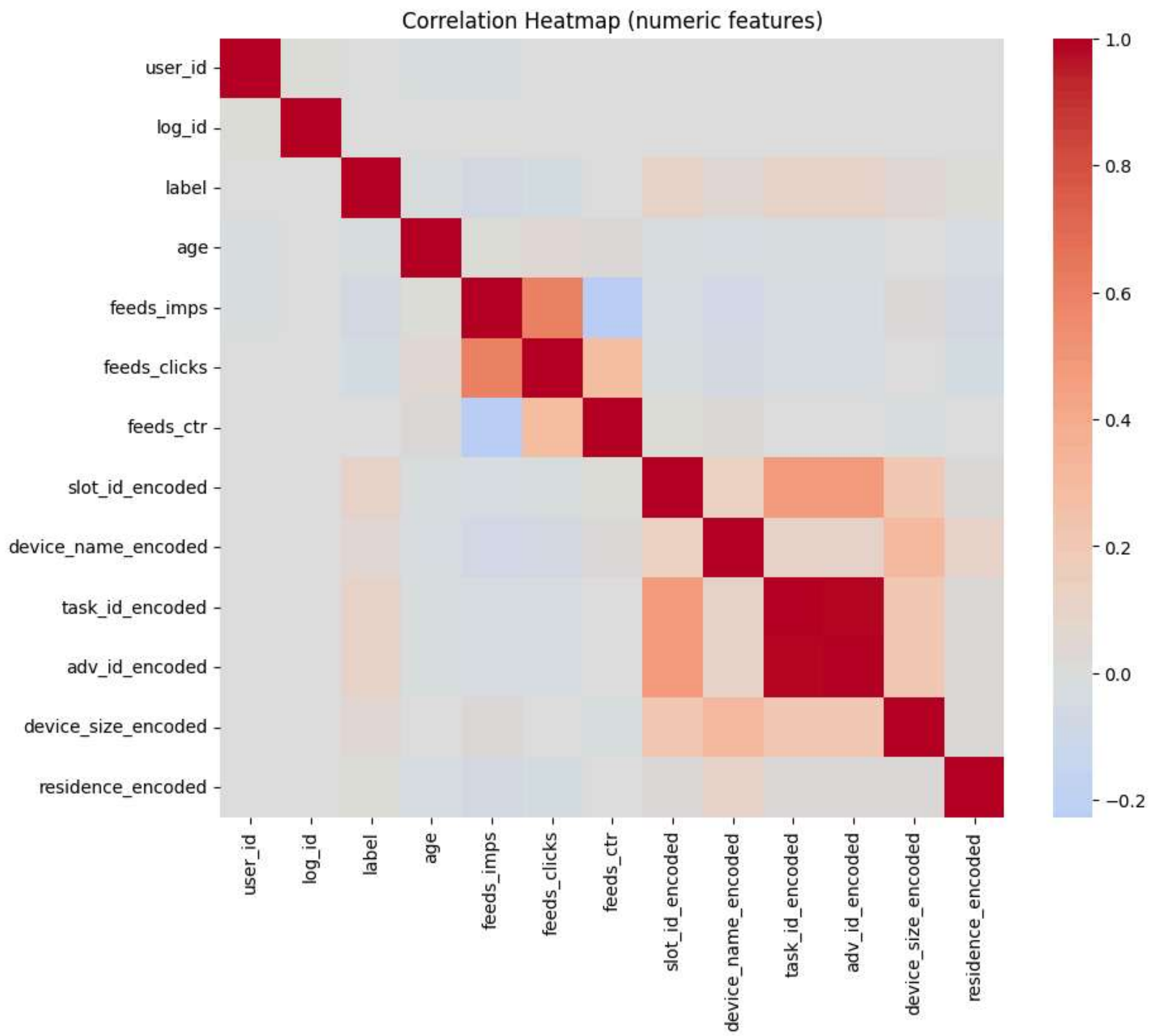




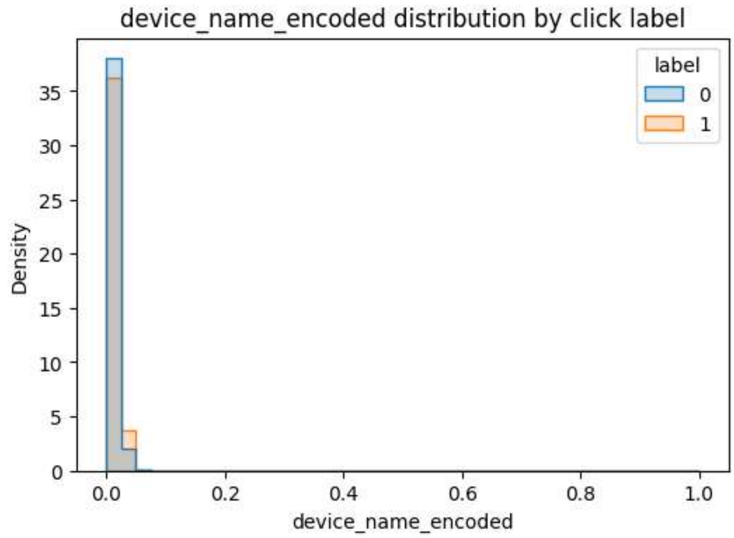




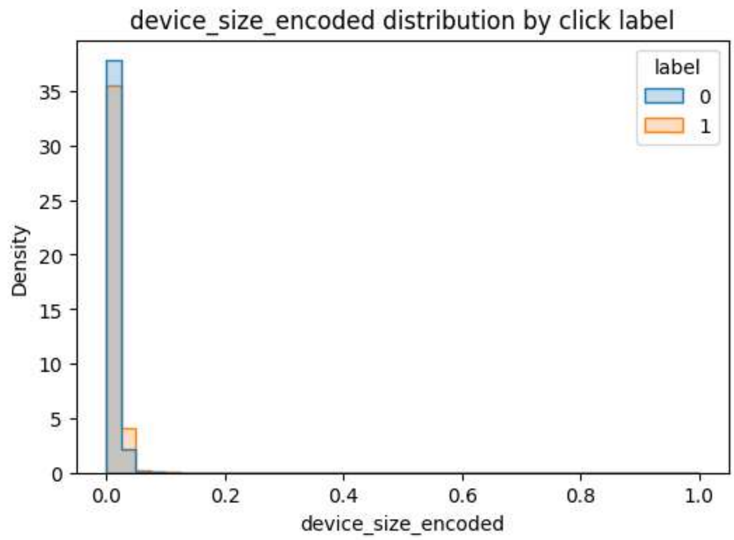

```
/usr/local/lib/python3.11/dist-packages/matplotlib/colors.py:721: RuntimeWarning: invalid value encountered in less
  xa[xa < 0] = -1
```



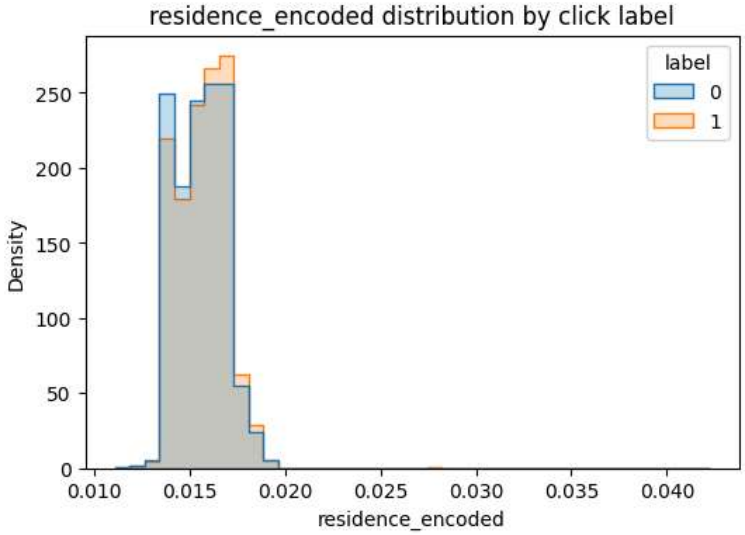

```
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and wil
l be removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1075: FutureWarning: When grouping with a length-1 list-like, y
ou will need to pass a length-1 tuple to get_group in a future version of pandas. Pass `(name,)` instead of `name` to silen
ce this warning.
  data_subset = grouped_data.get_group(pd_key)
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1075: FutureWarning: When grouping with a length-1 list-like, y
ou will need to pass a length-1 tuple to get_group in a future version of pandas. Pass `(name,)` instead of `name` to silen
ce this warning.
  data_subset = grouped_data.get_group(pd_key)
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1075: FutureWarning: When grouping with a length-1 list-like, y
ou will need to pass a length-1 tuple to get_group in a future version of pandas. Pass `(name,)` instead of `name` to silen
ce this warning.
  data_subset = grouped_data.get_group(pd_key)
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1075: FutureWarning: When grouping with a length-1 list-like, y
ou will need to pass a length-1 tuple to get_group in a future version of pandas. Pass `(name,)` instead of `name` to silen
ce this warning.
  data_subset = grouped_data.get_group(pd_key)
```

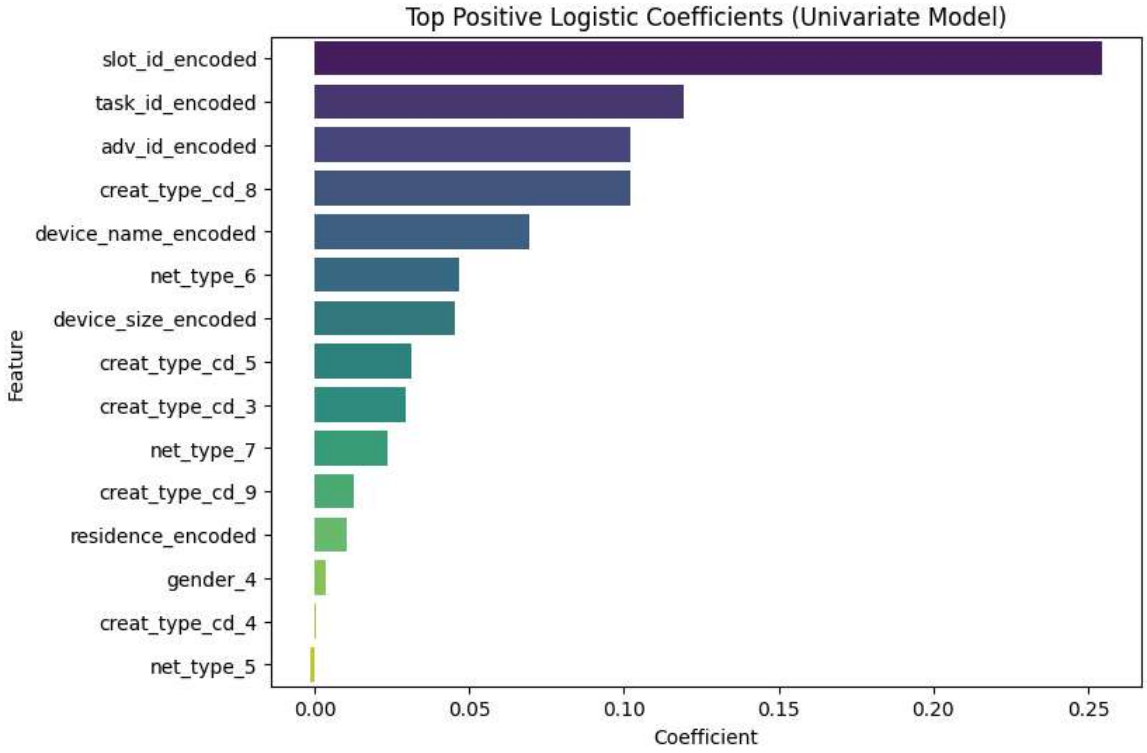


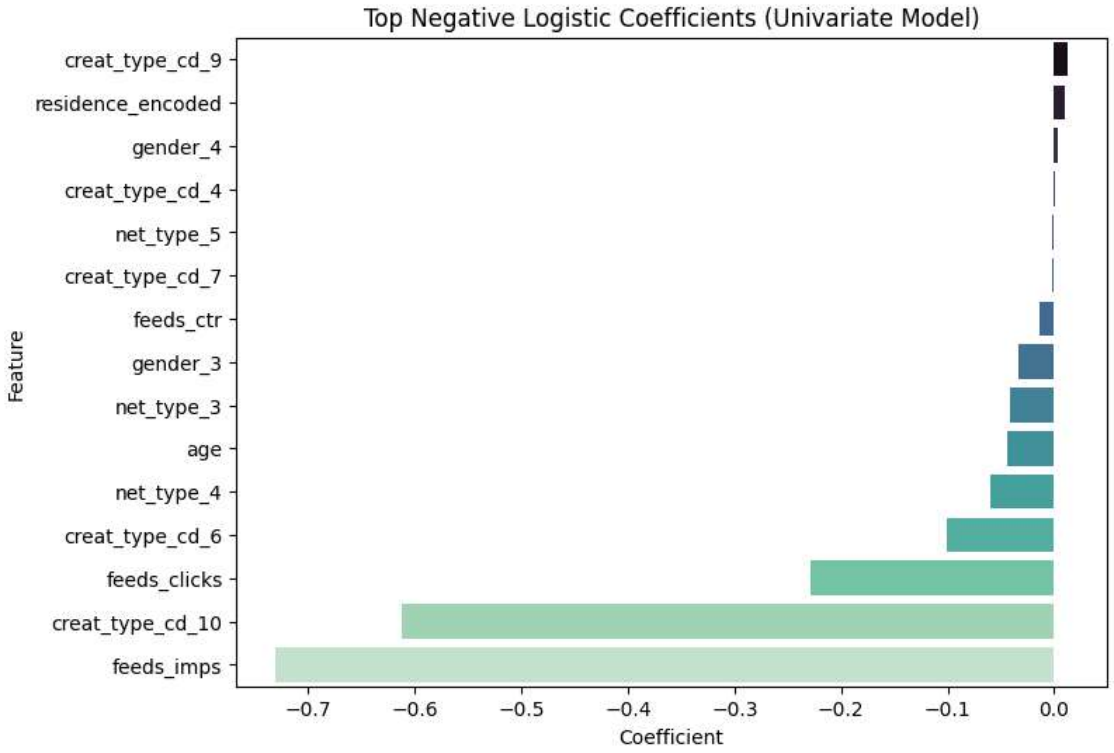

```
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and wil
l be removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1075: FutureWarning: When grouping with a length-1 list-like, y
ou will need to pass a length-1 tuple to get_group in a future version of pandas. Pass `(name,)` instead of `name` to silen
ce this warning.
  data_subset = grouped_data.get_group(pd_key)
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1075: FutureWarning: When grouping with a length-1 list-like, y
ou will need to pass a length-1 tuple to get_group in a future version of pandas. Pass `(name,)` instead of `name` to silen
ce this warning.
  data_subset = grouped_data.get_group(pd_key)
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1075: FutureWarning: When grouping with a length-1 list-like, y
ou will need to pass a length-1 tuple to get_group in a future version of pandas. Pass `(name,)` instead of `name` to silen
ce this warning.
  data_subset = grouped_data.get_group(pd_key)
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1075: FutureWarning: When grouping with a length-1 list-like, y
ou will need to pass a length-1 tuple to get_group in a future version of pandas. Pass `(name,)` instead of `name` to silen
ce this warning.
  data_subset = grouped_data.get_group(pd_key)
```




```
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and wil
l be removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1075: FutureWarning: When grouping with a length-1 list-like, y
ou will need to pass a length-1 tuple to get_group in a future version of pandas. Pass `(name,)` instead of `name` to silen
ce this warning.
  data_subset = grouped_data.get_group(pd_key)
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1075: FutureWarning: When grouping with a length-1 list-like, y
ou will need to pass a length-1 tuple to get_group in a future version of pandas. Pass `(name,)` instead of `name` to silen
ce this warning.
  data_subset = grouped_data.get_group(pd_key)
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1075: FutureWarning: When grouping with a length-1 list-like, y
ou will need to pass a length-1 tuple to get_group in a future version of pandas. Pass `(name,)` instead of `name` to silen
ce this warning.
  data_subset = grouped_data.get_group(pd_key)
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1075: FutureWarning: When grouping with a length-1 list-like, y
ou will need to pass a length-1 tuple to get_group in a future version of pandas. Pass `(name,)` instead of `name` to silen
ce this warning.
  data_subset = grouped_data.get_group(pd_key)
```


residence_encoded distribution by click label
label
0
1
Density
250
200
150
100
50
0
0.010
0.015
0.020
0.025
0.030
0.035
0.040
residence_encoded


Top Positive Logistic Coefficients (Univariate Model)
slot_id_encoded
task_id_encoded
adv_id_encoded
creat_type_cd_8
device_name_encoded
net_type_6
device_size_encoded
creat_type_cd_5
creat_type_cd_3
net_type_7
creat_type_cd_9
residence_encoded
gender_4
creat_type_cd_4
net_type_5
Feature
0.00
0.05
0.10
0.15
0.20
0.25
Coefficient



In [2]:
```
!pip install ctgan --quiet

# CTGAN SYNTHETIC DATA GENERATION
import pandas as pd
import numpy as np
from ctgan import CTGAN
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from scipy.stats import wasserstein_distance
from scipy.spatial.distance import jensenshannon
import gc

# Define seed
SEED = 2025
np.random.seed(SEED)

print("\n" + "="*80)
print("SYNTHETIC DATA GENERATION - Enhanced CTGAN with More Features")
print("="*80)

# STEP 1: Load data with extended features
print("\n[Step 1] Loading data with comprehensive feature set...")

train_user_full = pd.read_csv("/kaggle/input/digix-global-ai-challenge/train/train_data_ads.csv")
train_adv_full = pd.read_csv("/kaggle/input/digix-global-ai-challenge/train/train_data_feeds.csv")
train_adv_full.rename(columns={'u_userId': 'user_id'}, inplace=True)

# extended feature list
user_features_extended = [
    'user_id', 'log_id', 'label',
    'age', 'gender', 'residence', 'city',
    'device_name', 'device_size', 'net_type',
    'task_id', 'adv_id', 'adv_prim_id', 'creat_type_cd',
    'city_rank', 'slot_id', 'spread_app_id', 'inter_type_cd',
    'series_dev', 'emui_dev', 'series_group',
    'app_second_class', 'hispace_app_tags'
]

adv_features = ['user_id', 'label']

def safe_select(df, cols, df_name):
    available = [c for c in cols if c in df.columns]
    missing = [c for c in cols if c not in df.columns]
    if missing:
        print(f"  [{df_name}] Missing: {missing}")
    return available

user_cols_available = safe_select(train_user_full, user_features_extended, "train_user")
adv_cols_available = safe_select(train_adv_full, adv_features, "train_adv")
```

```
train_user_subset = train_user_full[user_cols_available].copy()
train_adv_subset = train_adv_full[adv_cols_available].copy()

# Free memory
del train_user_full, train_adv_full
gc.collect()

# Create user aggregates
train_adv_subset['label_binary'] = train_adv_subset['label'].replace({-1: 0, 1: 1}).astype(int)
user_feed_stats = (
    train_adv_subset
    .groupby('user_id', as_index=False)
    .agg(
        total_feed_views=('label_binary', 'count'),
        total_feed_clicks=('label_binary', 'sum'),
        avg_feed_ctr=('label_binary', 'mean')
    )
)

# Merge
merged_full = train_user_subset.merge(user_feed_stats, on='user_id', how='left')

# Free memory
del train_user_subset, train_adv_subset, user_feed_stats
gc.collect()

# Fill missing
for col in ['total_feed_views', 'total_feed_clicks', 'avg_feed_ctr']:
    if col in merged_full.columns:
        fill_value = 0 if col != 'avg_feed_ctr' else merged_full[col].mean()
        merged_full[col].fillna(fill_value, inplace=True)

# Convert label
merged_full['label'] = merged_full['label'].replace({-1: 0, 1: 1}).astype(int)

print(f"  Merged shape: {merged_full.shape}")
print(f"  Total features: {len(merged_full.columns)}")


# STEP 2: Split and prepare CTGAN data
print("\n[Step 2] Splitting data...")

train_split, test_split = train_test_split(
    merged_full,
    test_size=0.2,
    stratify=merged_full['label'],
    random_state=SEED
)

# Extract clicks only
clicks_for_ctgan = train_split[train_split['label'] == 1].copy()
noclicks_for_later = train_split[train_split['label'] == 0].copy()

print(f"  Clicks for CTGAN: {len(clicks_for_ctgan):,}")
print(f"  No-clicks for later: {len(noclicks_for_later):,}")

# Free memory
del merged_full
gc.collect()

# Drop ID columns
id_cols = ['user_id', 'log_id', 'label']
ctgan_working_data = clicks_for_ctgan.drop(columns=[c for c in id_cols if c in clicks_for_ctgan.columns]).copy()


# STEP 3: Repeat Feature encoding strategy
print("\n[Step 3] Encoding features...")

low_cardinality_categorical = ['gender', 'net_type', 'creat_type_cd', 'inter_type_cd']
medium_cardinality_to_group = ['device_name', 'slot_id', 'city', 'residence',
                               'series_dev', 'emui_dev', 'series_group',
                               'app_second_class', 'spread_app_id']
high_cardinality_to_encode = ['task_id', 'adv_id', 'device_size', 'adv_prim_id', 'hispace_app_tags']
numeric_keep_as_is = ['age', 'city_rank', 'total_feed_views', 'total_feed_clicks', 'avg_feed_ctr']

discrete_features_list = []
continuous_features_list = []

# Low-cardinality: keep as categorical
print("\n  Low-cardinality categorical:")
for col in low_cardinality_categorical:
    if col in ctgan_working_data.columns:
        ctgan_working_data[col] = ctgan_working_data[col].astype(str)
        discrete_features_list.append(col)
        n_unique = ctgan_working_data[col].nunique()
```

```
        print(f"    ✓ {col}: {n_unique} categories")

# Medium-cardinality: group rare
print("\n  Medium-cardinality categorical (grouped):")
def group_infrequent(df, column, keep_top=30):
    freq_counts = df[column].value_counts()
    top_categories = freq_counts.head(keep_top).index.tolist()
    df[column] = df[column].apply(lambda x: str(x) if x in top_categories else 'rare')
    return df

for col in medium_cardinality_to_group:
    if col in ctgan_working_data.columns:
        n_before = ctgan_working_data[col].nunique()
        ctgan_working_data = group_infrequent(ctgan_working_data, col, keep_top=30)
        ctgan_working_data[col] = ctgan_working_data[col].astype(str)
        n_after = ctgan_working_data[col].nunique()
        discrete_features_list.append(col)
        print(f"    ✓ {col}: {n_before} → {n_after} categories")

# High-cardinality: target encode
print("\n  High-cardinality categorical (target encoded):")
for col in high_cardinality_to_encode:
    if col in ctgan_working_data.columns:
        n_unique = ctgan_working_data[col].nunique()

        encoding_dict = train_split.groupby(col)['label'].mean().to_dict()
        global_avg = train_split['label'].mean()

        encoded_col_name = f"{col}_te"
        ctgan_working_data[encoded_col_name] = ctgan_working_data[col].map(encoding_dict).fillna(global_avg)

        continuous_features_list.append(encoded_col_name)
        ctgan_working_data.drop(columns=[col], inplace=True)

        print(f"    ✓ {col}: {n_unique} → {encoded_col_name}")

# Numeric: keep as-is
print("\n  Numeric features:")
for col in numeric_keep_as_is:
    if col in ctgan_working_data.columns:
        continuous_features_list.append(col)
        mean_val = ctgan_working_data[col].mean()
        std_val = ctgan_working_data[col].std()
        print(f"    ✓ {col}: mean={mean_val:.2f}, std={std_val:.2f}")

# Drop unprocessed
all_processed = discrete_features_list + continuous_features_list
unprocessed = [c for c in ctgan_working_data.columns if c not in all_processed]
if unprocessed:
    print(f"\n  Dropping unprocessed: {unprocessed}")
    ctgan_working_data.drop(columns=unprocessed, inplace=True)

ctgan_input_final = ctgan_working_data.copy()

print(f"\n  Final CTGAN input: {ctgan_input_final.shape}")
print(f"  Discrete features: {len(discrete_features_list)}")
print(f"  Continuous features: {len(continuous_features_list)}")

# Free memory
del ctgan_working_data
gc.collect()


# STEP 4: Train CTGAN
print("\n[Step 4] Training CTGAN...")
print("  Config: epochs=100, batch=500, gen/disc=(256,256)")

ctgan_synthesizer = CTGAN(
    epochs=100,
    batch_size=500,
    generator_dim=(256, 256),
    discriminator_dim=(256, 256),
    generator_lr=2e-4,
    discriminator_lr=2e-4,
    verbose=True
)

print("\n  Training...\n")
ctgan_synthesizer.fit(ctgan_input_final, discrete_columns=discrete_features_list)
print("\n✓ CTGAN training complete")


# STEP 5: Generate synthetic samples
print("\n[Step 5] Calculating synthetic requirements...")

n_real_noclicks = len(noclicks_for_later)
```

```
n_real_clicks = len(clicks_for_ctgan)
target_total_clicks = int((n_real_noclicks / 0.85) * 0.15)
n_synthetic_required = max(0, target_total_clicks - n_real_clicks)

print(f"  Real no-clicks: {n_real_noclicks:,}")
print(f"  Real clicks: {n_real_clicks:,}")
print(f"  Target clicks (15%): {target_total_clicks:,}")
print(f"  Synthetic needed: {n_synthetic_required:,}")

if n_synthetic_required > 0:
    print(f"\n  Generating {n_synthetic_required:,} samples...")
    synthetic_generated = ctgan_synthesizer.sample(n_synthetic_required)
    synthetic_generated = synthetic_generated[ctgan_input_final.columns]
    print(f"✓ Generated {len(synthetic_generated):,} synthetic clicks")
else:
    synthetic_generated = ctgan_input_final.iloc[0:0].copy()
    print("  No synthetic samples needed")


# STEP 6: Quality assessment
if n_synthetic_required > 0:
    print("\n" + "="*80)
    print("QUALITY ASSESSMENT")
    print("="*80)

    continuous_to_compare = [c for c in continuous_features_list if c in ctgan_input_final.columns]

    print(f"\nComparing {len(continuous_to_compare)} continuous features...")

    # Plot distributions
    n_plots = min(6, len(continuous_to_compare))
    fig, axes = plt.subplots(2, 3, figsize=(16, 9))
    axes = axes.flatten()

    for idx, feature in enumerate(continuous_to_compare[:n_plots]):
        ax = axes[idx]

        real_values = ctgan_input_final[feature].values
        synth_values = synthetic_generated[feature].values

        ax.hist(real_values, bins=50, density=True, alpha=0.6,
                label='Real', color='#1D3557', edgecolor='black', linewidth=0.5)
        ax.hist(synth_values, bins=50, density=True, alpha=0.6,
                label='Synthetic', color='#E63946', edgecolor='black', linewidth=0.5)

        ax.set_xlabel(feature, fontsize=10)
        ax.set_ylabel('Density', fontsize=10)
        ax.set_title(f'{feature}', fontsize=11, fontweight='bold')
        ax.legend(loc='best', fontsize=9)
        ax.grid(alpha=0.3)

    for idx in range(n_plots, len(axes)):
        axes[idx].set_visible(False)

    plt.tight_layout()
    plt.suptitle('Feature Distribution: Real vs Synthetic Clicks',
                 fontsize=14, fontweight='bold', y=1.00)
    plt.show()

    # STEP 7: PCA visualization
    print("\n[Step 7] PCA projection...")

    real_matrix = ctgan_input_final[continuous_to_compare].values
    synth_matrix = synthetic_generated[continuous_to_compare].values

    combined_matrix = np.vstack([real_matrix, synth_matrix])

    # Standardize
    scaler = StandardScaler()
    combined_scaled = scaler.fit_transform(combined_matrix)

    # PCA
    pca = PCA(n_components=2, random_state=SEED)
    pca_result = pca.fit_transform(combined_scaled)

    n_real_samples = real_matrix.shape[0]
    pca_real = pca_result[:n_real_samples]
    pca_synth = pca_result[n_real_samples:]

    # Plot PCA
    fig, ax = plt.subplots(figsize=(12, 9))

    ax.grid(alpha=0.25, linestyle='--', linewidth=0.6)

    # Synthetic (background)
    ax.scatter(
```

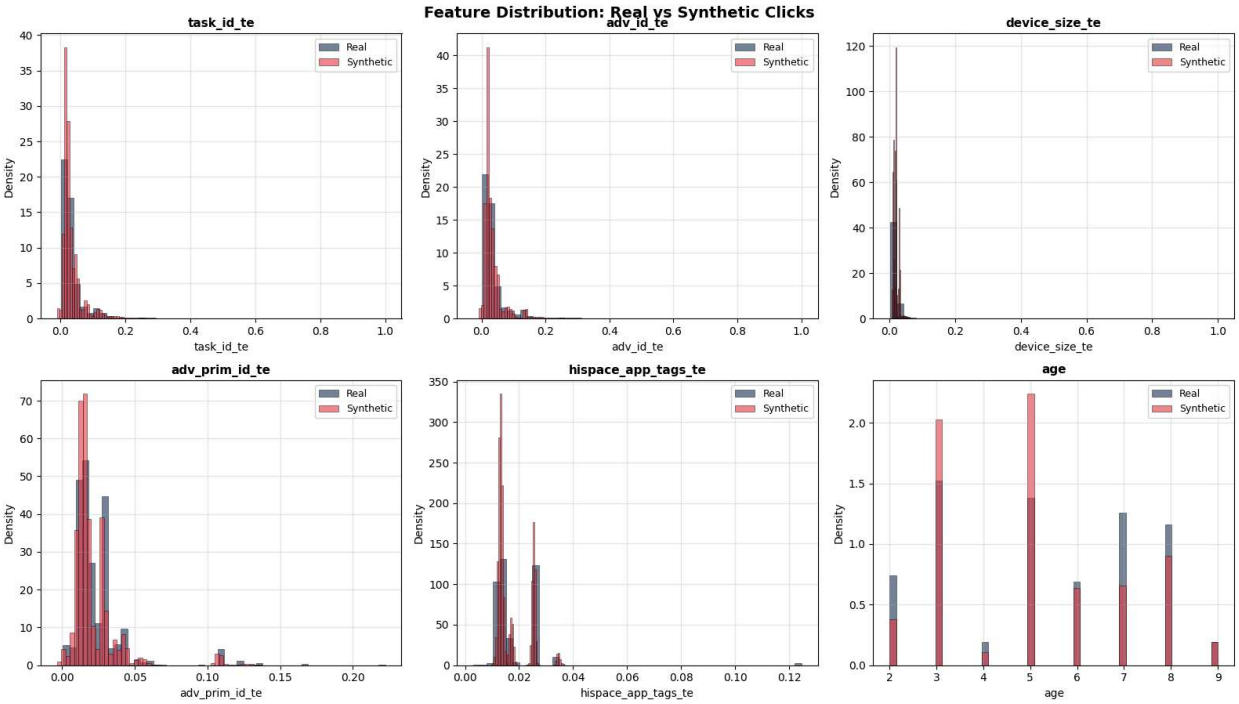
```
        pca_synth[:, 0], pca_synth[:, 1],
        s=6, alpha=0.10, color='#E63946',
        edgecolors='none', rasterized=True,
        label=f'Synthetic Clicks (n={len(pca_synth):,})'
    )

    # Real (foreground)
    ax.scatter(
        pca_real[:, 0], pca_real[:, 1],
        s=6, alpha=0.15, color='#1D3557',
        edgecolors='none', rasterized=True,
        label=f'Real Clicks (n={len(pca_real):,})'
    )

    # Center lines
    ax.axhline(y=0, color='gray', linestyle='--', linewidth=1, alpha=0.6, zorder=0)
    ax.axvline(x=0, color='gray', linestyle='--', linewidth=1, alpha=0.6, zorder=0)

    # Labels
    ax.set_xlabel(
        f'PC1 ({pca.explained_variance_ratio_[0]*100:.1f}% variance)',
        fontsize=13, fontweight='bold'
    )
    ax.set_ylabel(
        f'PC2 ({pca.explained_variance_ratio_[1]*100:.1f}% variance)',
        fontsize=13, fontweight='bold'
    )

    # Title
    ax.set_title(
        'PCA Projection: Real vs Synthetic Click Data\n(Enhanced CTGAN with Extended Features)',
        fontsize=15, fontweight='bold', pad=20
    )

    # Manual legend (fix color issue)
    real_patch = mpatches.Patch(color='#1D3557', label=f'Real Clicks (n={len(pca_real):,})')
    synth_patch = mpatches.Patch(color='#E63946', label=f'Synthetic Clicks (n={len(pca_synth):,})')
    ax.legend(
        handles=[real_patch, synth_patch],
        fontsize=11, loc='upper right',
        framealpha=0.95, edgecolor='black',
        fancybox=True, shadow=True
    )

    # Set axis limits
    x_range = pca_result[:, 0].max() - pca_result[:, 0].min()
    y_range = pca_result[:, 1].max() - pca_result[:, 1].min()
    ax.set_xlim(pca_result[:, 0].min() - 0.05*x_range,
                pca_result[:, 0].max() + 0.05*x_range)
    ax.set_ylim(pca_result[:, 1].min() - 0.05*y_range,
                pca_result[:, 1].max() + 0.05*y_range)

    plt.tight_layout()
    plt.show()

    # Distance metrics
    print("\n  Distribution distance metrics:")

    w_dist_pc1 = wasserstein_distance(pca_real[:, 0], pca_synth[:, 0])
    w_dist_pc2 = wasserstein_distance(pca_real[:, 1], pca_synth[:, 1])

    print(f"    Wasserstein Distance (PC1): {w_dist_pc1:.4f}")
    print(f"    Wasserstein Distance (PC2): {w_dist_pc2:.4f}")
    print(f"    Interpretation: < 0.3 (excellent), < 0.5 (good), < 1.0 (acceptable)")

    # JS divergence
    bins = 50
    hist_real_pc1, _ = np.histogram(pca_real[:, 0], bins=bins, density=True)
    hist_synth_pc1, _ = np.histogram(pca_synth[:, 0], bins=bins, density=True)

    hist_real_pc1 = hist_real_pc1 / (hist_real_pc1.sum() + 1e-10)
    hist_synth_pc1 = hist_synth_pc1 / (hist_synth_pc1.sum() + 1e-10)

    js_div = jensenshannon(hist_real_pc1, hist_synth_pc1)

    print(f"    Jensen-Shannon Divergence (PC1): {js_div:.4f}")
    print(f"    Interpretation: < 0.1 (excellent), < 0.2 (good), < 0.3 (acceptable)")

# Free memory before final step
del real_matrix, synth_matrix, combined_matrix, combined_scaled, pca_result
gc.collect()


# STEP 8: Assemble final dataset (OPTIMIZED)
print("\n[Step 8] Assembling final dataset (memory-optimized)...")
```

```python
# Process no-clicks (apply same encoding)
print("  Processing no-clicks...")
noclicks_processed = noclicks_for_later.drop(columns=[c for c in id_cols if c in noclicks_for_later.columns]).copy()

# Store IDs
user_ids_nocl = noclicks_for_later['user_id'].values.copy()
log_ids_nocl = noclicks_for_later['log_id'].values.copy()

# Apply encoding (low-cardinality)
for col in low_cardinality_categorical:
    if col in noclicks_processed.columns:
        noclicks_processed[col] = noclicks_processed[col].astype(str)

# Apply encoding (medium-cardinality)
for col in medium_cardinality_to_group:
    if col in noclicks_processed.columns:
        noclicks_processed = group_infrequent(noclicks_processed, col, keep_top=30)
        noclicks_processed[col] = noclicks_processed[col].astype(str)

# Apply encoding (high-cardinality)
for col in high_cardinality_to_encode:
    if col in noclicks_processed.columns:
        encoding_dict = train_split.groupby(col)['label'].mean().to_dict()
        global_avg = train_split['label'].mean()

        encoded_col_name = f"{col}_te"
        noclicks_processed[encoded_col_name] = noclicks_processed[col].map(encoding_dict).fillna(global_avg)
        noclicks_processed.drop(columns=[col], inplace=True)

# Drop unprocessed columns
unproc_nocl = [c for c in noclicks_processed.columns if c not in all_processed]
if unproc_nocl:
    noclicks_processed.drop(columns=unproc_nocl, inplace=True)

# Add identifiers
noclicks_processed['label'] = 0
noclicks_processed['user_id'] = user_ids_nocl
noclicks_processed['log_id'] = log_ids_nocl
noclicks_processed['is_synthetic'] = 0

del user_ids_nocl, log_ids_nocl
gc.collect()

# Process clicks
print("  Processing clicks...")
clicks_processed = ctgan_input_final.copy()

user_ids_clicks = clicks_for_ctgan['user_id'].values.copy()
log_ids_clicks = clicks_for_ctgan['log_id'].values.copy()

clicks_processed['label'] = 1
clicks_processed['user_id'] = user_ids_clicks
clicks_processed['log_id'] = log_ids_clicks
clicks_processed['is_synthetic'] = 0

del user_ids_clicks, log_ids_clicks
gc.collect()

# Process synthetic
if n_synthetic_required > 0:
    print("  Processing synthetic...")
    synthetic_final = synthetic_generated.copy()
    del synthetic_generated
    gc.collect()

    synthetic_final['label'] = 1
    synthetic_final['user_id'] = -1
    synthetic_final['log_id'] = range(60_000_000, 60_000_000 + len(synthetic_final))
    synthetic_final['is_synthetic'] = 1
else:
    synthetic_final = clicks_processed.iloc[0:0].copy()

# Combine all
print("  Combining datasets...")
combined_final_dataset = pd.concat([
    noclicks_processed,
    clicks_processed,
    synthetic_final
], ignore_index=True)

# Free intermediate dataframes
del noclicks_processed, clicks_processed, synthetic_final
gc.collect()

# Shuffle
combined_final_dataset = combined_final_dataset.sample(frac=1, random_state=SEED).reset_index(drop=True)
```

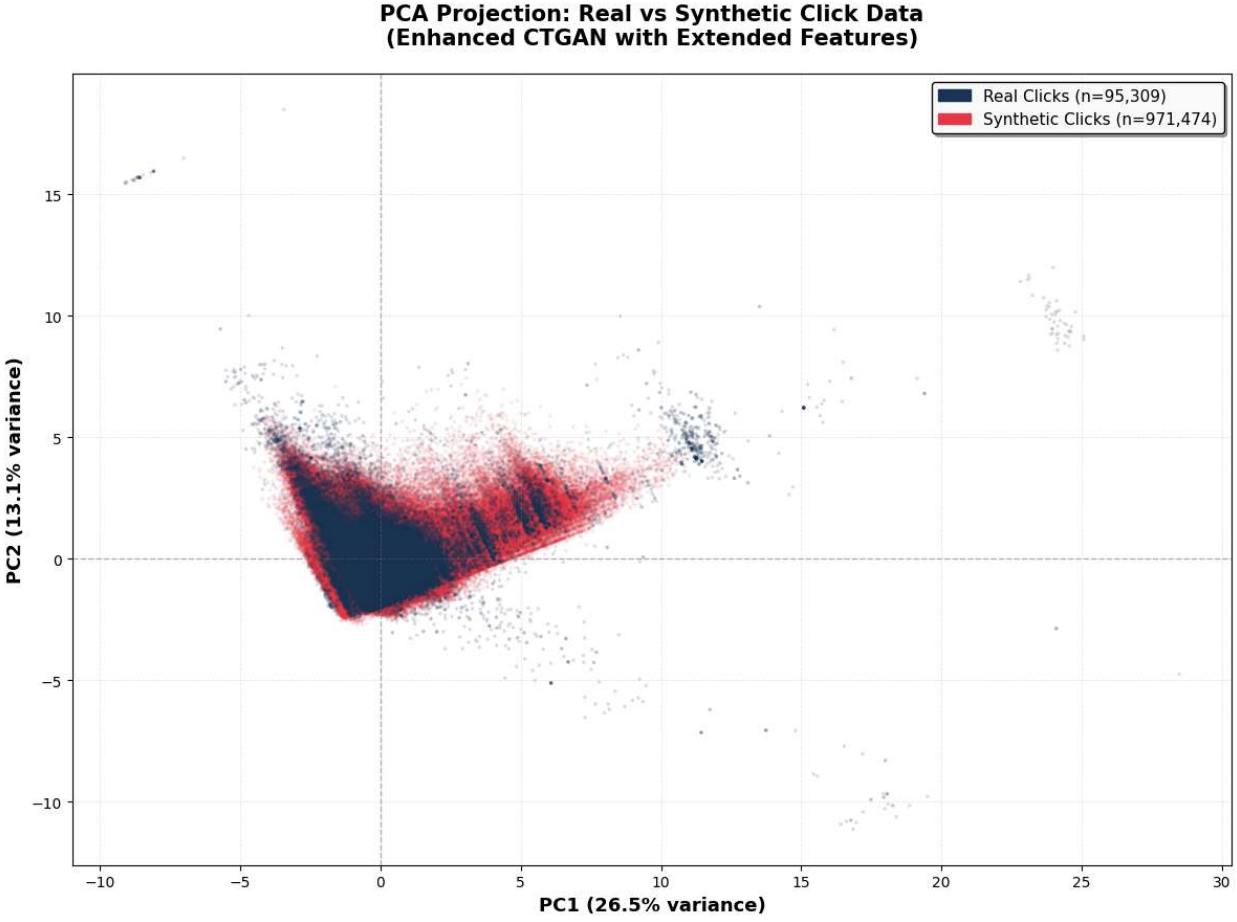
```python
print(f"  Final dataset: {combined_final_dataset.shape}")
print(f"\n  Class distribution:")
print(combined_final_dataset['label'].value_counts(normalize=True).round(4))
print(f"\n  Real vs Synthetic:")
print(combined_final_dataset['is_synthetic'].value_counts())

print("\n" + "="*80)
print("✓ SYNTHETIC DATA GENERATION COMPLETE")
print("="*80)



# STEP 9: Encode test set for modeling
print("\n[Step 9] Encoding test set...")

test_encoded = test_split.drop(columns=[c for c in id_cols if c in test_split.columns]).copy()

# Store test IDs
test_user_ids = test_split['user_id'].values.copy()
test_log_ids = test_split['log_id'].values.copy()
test_labels = test_split['label'].values.copy()

# Apply same encoding to test
for col in low_cardinality_categorical:
    if col in test_encoded.columns:
        test_encoded[col] = test_encoded[col].astype(str)

for col in medium_cardinality_to_group:
    if col in test_encoded.columns:
        test_encoded = group_infrequent(test_encoded, col, keep_top=30)
        test_encoded[col] = test_encoded[col].astype(str)

for col in high_cardinality_to_encode:
    if col in test_encoded.columns:
        encoding_dict = train_split.groupby(col)['label'].mean().to_dict()
        global_avg = train_split['label'].mean()

        encoded_col_name = f"{col}_te"
        test_encoded[encoded_col_name] = test_encoded[col].map(encoding_dict).fillna(global_avg)
        test_encoded.drop(columns=[col], inplace=True)

# Drop unprocessed
unproc_test = [c for c in test_encoded.columns if c not in all_processed]
if unproc_test:
    test_encoded.drop(columns=unproc_test, inplace=True)

# Add identifiers back
test_encoded['label'] = test_labels
test_encoded['user_id'] = test_user_ids
test_encoded['log_id'] = test_log_ids

del test_user_ids, test_log_ids, test_labels, test_split
gc.collect()

print(f"  Test encoded shape: {test_encoded.shape}")
print(f"  Test click rate: {test_encoded['label'].mean():.4f}")



# LOGISTIC REGRESSION MODELING
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    roc_auc_score, RocCurveDisplay, PrecisionRecallDisplay
)

print("\n" + "="*80)
print("LOGISTIC REGRESSION MODELING")
print("="*80)
```

```
================================================================================
SYNTHETIC DATA GENERATION - Enhanced CTGAN with More Features
================================================================================

[Step 1] Loading data with comprehensive feature set...
/tmp/ipykernel_47/466426478.py:89: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through ch
ained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are
setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col]
= df[col].method(value) instead, to perform the operation inplace on the original object.


  merged_full[col].fillna(fill_value, inplace=True)
```

```
  Merged shape: (7675517, 26)
  Total features: 26

[Step 2] Splitting data...
  Clicks for CTGAN: 95,309
  No-clicks for later: 6,045,104

[Step 3] Encoding features...

  Low-cardinality categorical:
    ✓ gender: 3 categories
    ✓ net_type: 6 categories
    ✓ creat_type_cd: 9 categories
    ✓ inter_type_cd: 4 categories

  Medium-cardinality categorical (grouped):
    ✓ device_name: 254 → 31 categories
    ✓ slot_id: 58 → 31 categories
    ✓ city: 339 → 31 categories
    ✓ residence: 35 → 31 categories
    ✓ series_dev: 24 → 24 categories
    ✓ emui_dev: 25 → 25 categories
    ✓ series_group: 7 → 7 categories
    ✓ app_second_class: 20 → 20 categories
    ✓ spread_app_id: 109 → 31 categories

  High-cardinality categorical (target encoded):
    ✓ task_id: 5500 → task_id_te
    ✓ adv_id: 6003 → adv_id_te
    ✓ device_size: 830 → device_size_te
    ✓ adv_prim_id: 497 → adv_prim_id_te
    ✓ hispace_app_tags: 42 → hispace_app_tags_te

  Numeric features:
    ✓ age: mean=5.28, std=2.10
    ✓ city_rank: mean=3.24, std=1.27
    ✓ total_feed_views: mean=51.96, std=72.57
    ✓ total_feed_clicks: mean=5.72, std=7.90
    ✓ avg_feed_ctr: mean=0.13, std=0.16

  Final CTGAN input: (95309, 23)
  Discrete features: 13
  Continuous features: 10

[Step 4] Training CTGAN...
  Config: epochs=100, batch=500, gen/disc=(256,256)

  Training...

Gen. (0.00) | Discrim. (0.00):   0%|          | 0/100 [00:00<?, ?it/s]/usr/local/lib/python3.11/dist-packages/torch/autogra
d/graph.py:823: UserWarning: Attempting to run cuBLAS, but there was no current CUDA context! Attempting to set the primary
context... (Triggered internally at /pytorch/aten/src/ATen/cuda/CublasHandlePool.cpp:180.)
  return Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the backward pass
Gen. (-0.16) | Discrim. (-0.30): 100%|██████████| 100/100 [11:12<00:00,  6.73s/it]
✓ CTGAN training complete

[Step 5] Calculating synthetic requirements...
  Real no-clicks: 6,045,104
  Real clicks: 95,309
  Target clicks (15%): 1,066,783
  Synthetic needed: 971,474

  Generating 971,474 samples...
✓ Generated 971,474 synthetic clicks

================================================================================
QUALITY ASSESSMENT
================================================================================

Comparing 10 continuous features...
```




```
[Step 7] PCA projection...
```



```
  Distribution distance metrics:
    Wasserstein Distance (PC1): 0.1641
    Wasserstein Distance (PC2): 0.0932
    Interpretation: < 0.3 (excellent), < 0.5 (good), < 1.0 (acceptable)
    Jensen-Shannon Divergence (PC1): 0.3021
    Interpretation: < 0.1 (excellent), < 0.2 (good), < 0.3 (acceptable)

[Step 8] Assembling final dataset (memory-optimized)...
  Processing no-clicks...
  Processing clicks...
  Processing synthetic...
  Combining datasets...
  Final dataset: (7111887, 27)

  Class distribution:
label
0    0.85
1    0.15
Name: proportion, dtype: float64

  Real vs Synthetic:
is_synthetic
0    6140413
1     971474
Name: count, dtype: int64

================================================================================
✓ SYNTHETIC DATA GENERATION COMPLETE
================================================================================

[Step 9] Encoding test set...
  Test encoded shape: (1535104, 26)
  Test click rate: 0.0155

================================================================================
LOGISTIC REGRESSION MODELING
================================================================================
```

In [8]:

```python
# LIGHTWEIGHT LOGISTIC REGRESSION
print("\n" + "="*80)
print("LIGHTWEIGHT LR MODELING (RAM OPTIMIZED)")
print("="*80)


# 1. Subsample combined_final_dataset
print("\n[Step] Subsampling combined dataset to avoid RAM overflow...")

TARGET_SAMPLE_SIZE = 600_000  # adjustable (500k–800k range)

if len(combined_final_dataset) > TARGET_SAMPLE_SIZE:
    sample_df = combined_final_dataset.sample(
        n=TARGET_SAMPLE_SIZE, random_state=SEED, replace=False
    )
else:
    sample_df = combined_final_dataset.copy()

print(f"  Sampled size: {len(sample_df):,}")
print(f"  Click rate: {sample_df['label'].mean():.4f}")


# 2. Train/Validation Split
print("\n[Step] Train-validation split...")

train_df, val_df = train_test_split(
    sample_df,
    test_size=0.2,
    stratify=sample_df['label'],
    random_state=SEED
)

y_train = train_df['label'].copy()
y_val = val_df['label'].copy()

X_train = train_df.drop(columns=['user_id', 'log_id', 'label', 'is_synthetic'])
X_val = val_df.drop(columns=['user_id', 'log_id', 'label', 'is_synthetic'])

# Clear memory
del sample_df, train_df, val_df
gc.collect()

print(f"Train size: {len(y_train):,}")
print(f"Val size: {len(y_val):,}")


# 3. One-hot encode lightweight categorical columns
print("\n[Step] One-hot encoding selected categorical vars...")
```

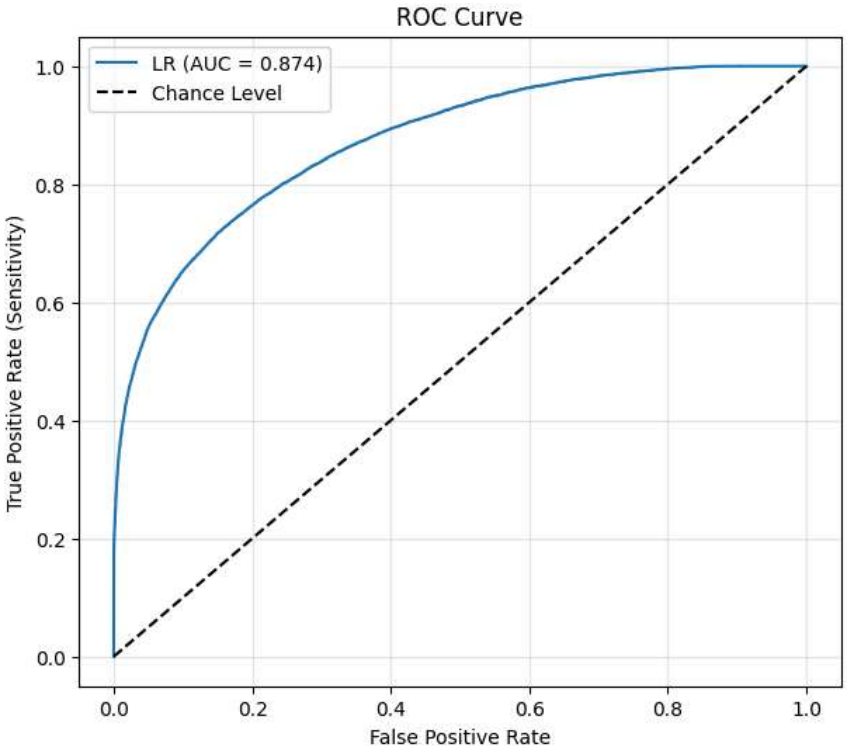
```
all_categorical_cols = low_cardinality_categorical + medium_cardinality_to_group

def encode_ohe(train_df, val_df, categorical_cols):
    df_tr = train_df.copy()
    df_val = val_df.copy()
    for col in categorical_cols:
        if col in df_tr.columns:
            dtrain = pd.get_dummies(df_tr[col], prefix=col, drop_first=True)
            dval = pd.get_dummies(df_val[col], prefix=col, drop_first=True)
            dval = dval.reindex(columns=dtrain.columns, fill_value=0)
            df_tr = pd.concat([df_tr.drop(columns=[col]), dtrain], axis=1)
            df_val = pd.concat([df_val.drop(columns=[col]), dval], axis=1)
    return df_tr, df_val

X_train_encoded, X_val_encoded = encode_ohe(X_train, X_val, all_categorical_cols)

# Clear memory
del X_train, X_val
gc.collect()

print(f"  Encoded training shape: {X_train_encoded.shape}")


# 4. Scale
print("\n[Step] Scaling features...")

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_encoded)
X_val_scaled = scaler.transform(X_val_encoded)

# Clear memory
del X_train_encoded, X_val_encoded
gc.collect()

print("  Scaling complete.")


# 5. Train Logistic Regression
print("\n[Step] Training Logistic Regression (RAM-optimized)...")

lr_final = LogisticRegression(
    max_iter=2000,
    n_jobs=-1,
    random_state=SEED
)

lr_final.fit(X_train_scaled, y_train)

print("✓ Logistic Regression trained successfully")


# 6. Evaluation
print("\n" + "-"*80)
print("LR PERFORMANCE (RAM-SAFE)")
print("-"*80)

y_pred = lr_final.predict(X_val_scaled)
y_proba = lr_final.predict_proba(X_val_scaled)[:, 1]

results_lr = pd.DataFrame({
    'Accuracy': [accuracy_score(y_val, y_pred)],
    'Precision': [precision_score(y_val, y_pred, zero_division=0)],
    'Recall': [recall_score(y_val, y_pred)],
    'F1': [f1_score(y_val, y_pred)],
    'AUC': [roc_auc_score(y_val, y_proba)]
})

display(results_lr.round(4))


# ADDITIONAL MODEL CURVES: ROC + Sensitivity-Specificity

from sklearn.metrics import roc_curve, RocCurveDisplay
from sklearn.metrics import precision_recall_curve, PrecisionRecallDisplay

print("\nPlotting curves...")

# 1. ROC Curve
fpr, tpr, roc_thresholds = roc_curve(y_val, y_proba)

plt.figure(figsize=(7, 6))
plt.plot(fpr, tpr, label=f"LR (AUC = {roc_auc_score(y_val, y_proba):.3f})")
plt.plot([0, 1], [0, 1], 'k--', label="Chance Level")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate (Sensitivity)")
```

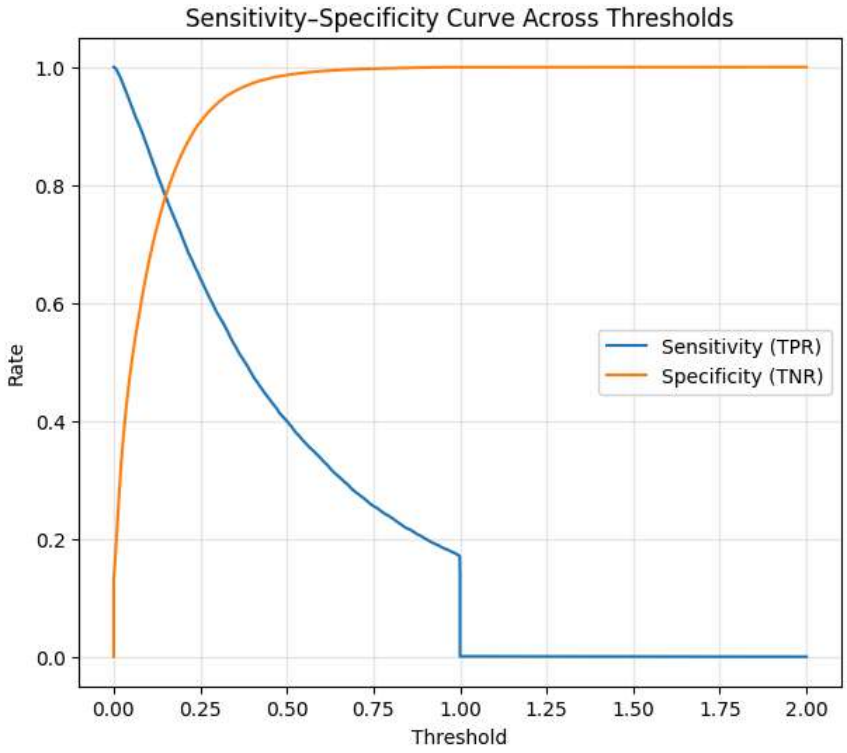
```
plt.title("ROC Curve")
plt.legend()
plt.grid(alpha=0.3)
plt.show()


# 2. Sensitivity-Specificity Curve
specificity = 1 - fpr  # Convert FPR -> Specificity

plt.figure(figsize=(7, 6))
plt.plot(roc_thresholds, tpr, label="Sensitivity (TPR)")
plt.plot(roc_thresholds, specificity, label="Specificity (TNR)")
plt.xlabel("Threshold")
plt.ylabel("Rate")
plt.title("Sensitivity-Specificity Curve Across Thresholds")
plt.legend()
plt.grid(alpha=0.3)
plt.show()


# 3. Precision-Recall Curve
prec, rec, pr_thresholds = precision_recall_curve(y_val, y_proba)

plt.figure(figsize=(7, 6))
plt.plot(rec, prec)
plt.title("Precision-Recall Curve")
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.grid(alpha=0.3)
plt.show()

print("✓ Curves plotted successfully!")
```

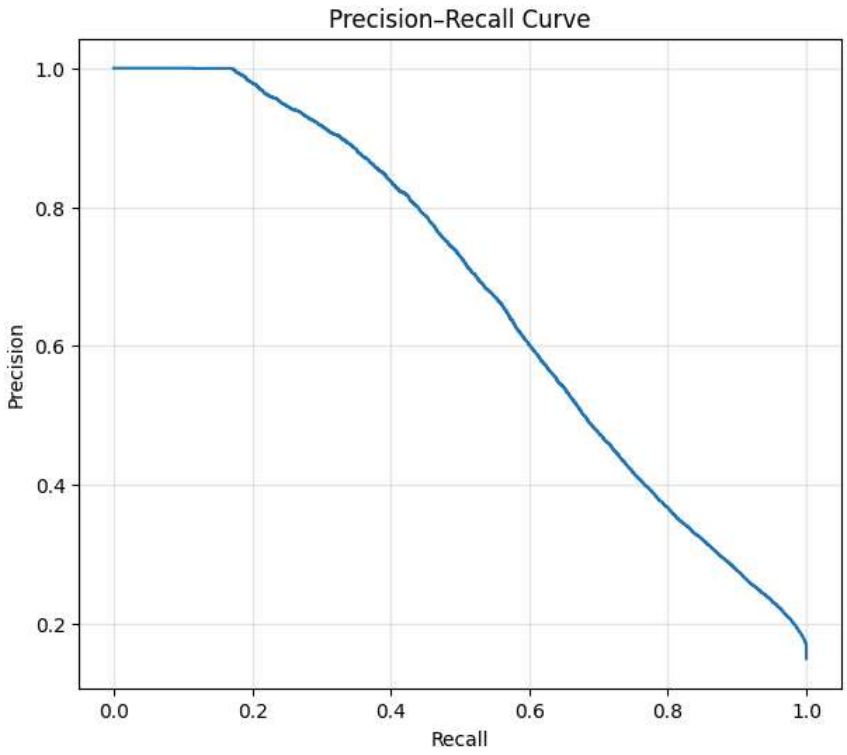
```
================================================================================
LIGHTWEIGHT LR MODELING (RAM OPTIMIZED)
================================================================================

[Step] Subsampling combined dataset to avoid RAM overflow...
  Sampled size: 600,000
  Click rate: 0.1499

[Step] Train-validation split...
  Train size: 480,000
  Val size:   120,000

[Step] One-hot encoding selected categorical vars...
  Encoded training shape: (480000, 269)

[Step] Scaling features...
  Scaling complete.

[Step] Training Logistic Regression (RAM-optimized)...
✓ Logistic Regression trained successfully

--------------------------------------------------------------------------------
LR PERFORMANCE (RAM-SAFE)
--------------------------------------------------------------------------------
```

	Accuracy	Precision	Recall	F1	AUC
0	0.8983	0.8371	0.3996	0.541	0.8735

```
Plotting curves...
```


ROC Curve
LR (AUC = 0.874)
Chance Level
True Positive Rate (Sensitivity)
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0


Sensitivity–Specificity Curve Across Thresholds
Sensitivity (TPR)
Specificity (TNR)
Rate
Threshold
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

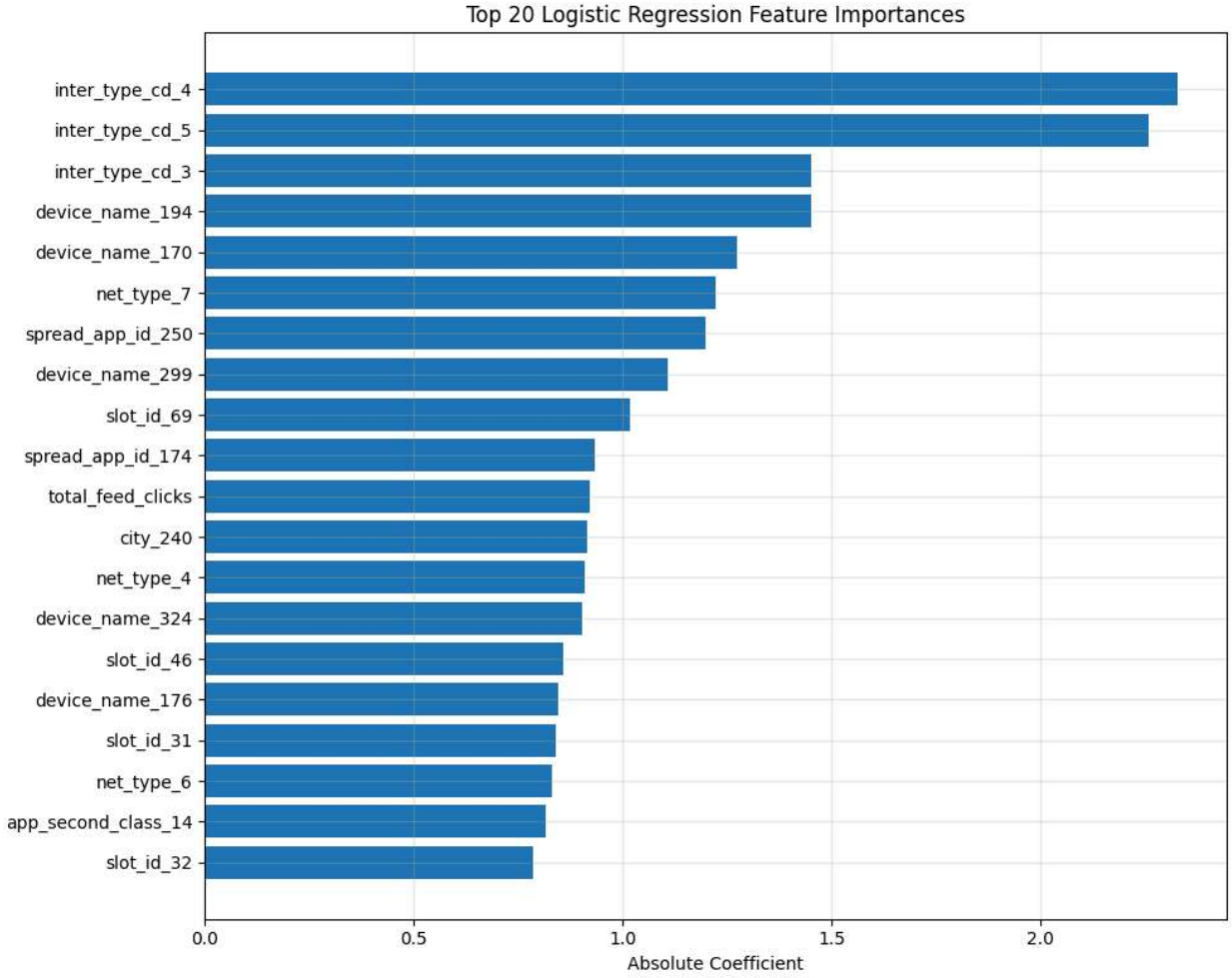


✓ Curves plotted successfully!

In [12]:
```python
# LOGISTIC REGRESSION FEATURE IMPORTANCE

print("\n" + "="*80)
print("LOGISTIC REGRESSION FEATURE IMPORTANCE")
print("="*80)

# Recover feature names from the scaler input
lr_feature_names = scaler.feature_names_in_  # works because sklearn 1.0+
lr_coefficients = lr_final.coef_[0]

# Build importance dataframe
lr_importance = pd.DataFrame({
    'feature': lr_feature_names,
    'coef': lr_coefficients,
    'abs_coef': np.abs(lr_coefficients)
}).sort_values('abs_coef', ascending=False)

print("\nTop 20 most important LR features:")
display(lr_importance.head(20))

# Plot
plt.figure(figsize=(10, 8))
top_n = 20
plt.barh(
    lr_importance.head(top_n)['feature'][::-1],
    lr_importance.head(top_n)['abs_coef'][::-1]
)
plt.xlabel("Absolute Coefficient")
plt.title("Top 20 Logistic Regression Feature Importances")
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```

```
================================================================================
LOGISTIC REGRESSION FEATURE IMPORTANCE
================================================================================

Top 20 most important LR features:
```

	feature	coef	abs_coef
26	inter_type_cd_4	-2.328789	2.328789
27	inter_type_cd_5	-2.259576	2.259576
25	inter_type_cd_3	-1.451248	1.451248
40	device_name_194	-1.450563	1.450563
37	device_name_170	-1.272318	1.272318
16	net_type_7	-1.222945	1.222945
248	spread_app_id_250	-1.197776	1.197776
53	device_name_299	-1.108193	1.108193
94	slot_id_69	-1.015903	1.015903
239	spread_app_id_174	-0.933674	0.933674
3	total_feed_clicks	-0.919475	0.919475
108	city_240	-0.915946	0.915946
13	net_type_4	-0.907428	0.907428
57	device_name_324	-0.901157	0.901157
82	slot_id_46	0.856615	0.856615
38	device_name_176	-0.845191	0.845191
73	slot_id_31	-0.840381	0.840381
15	net_type_6	-0.830101	0.830101
217	app_second_class_14	-0.815977	0.815977
74	slot_id_32	-0.786258	0.786258

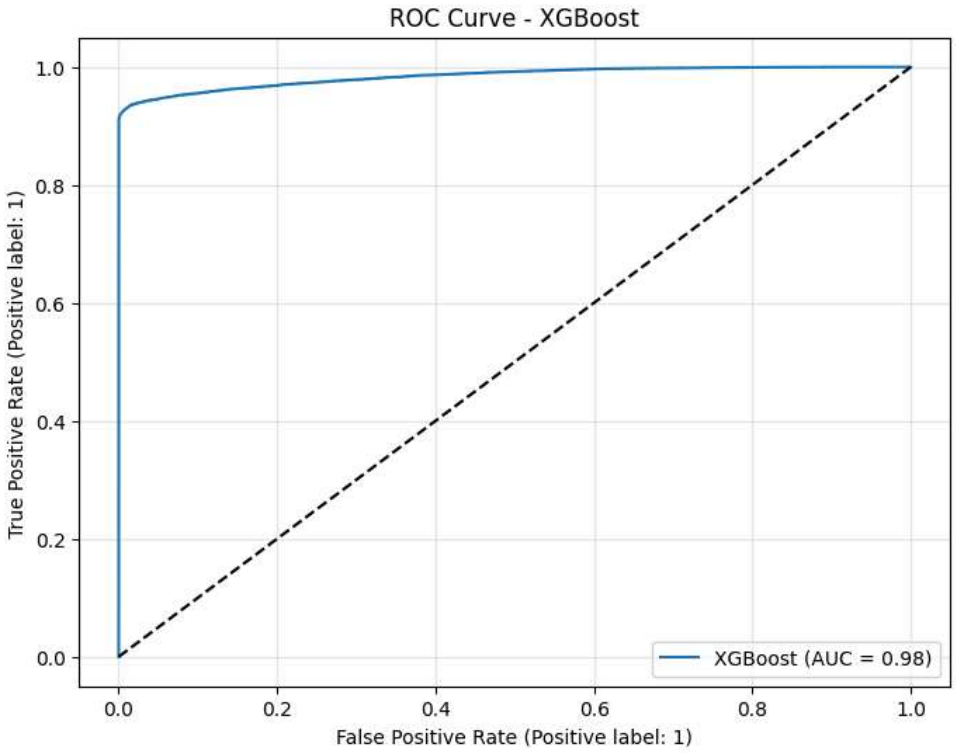


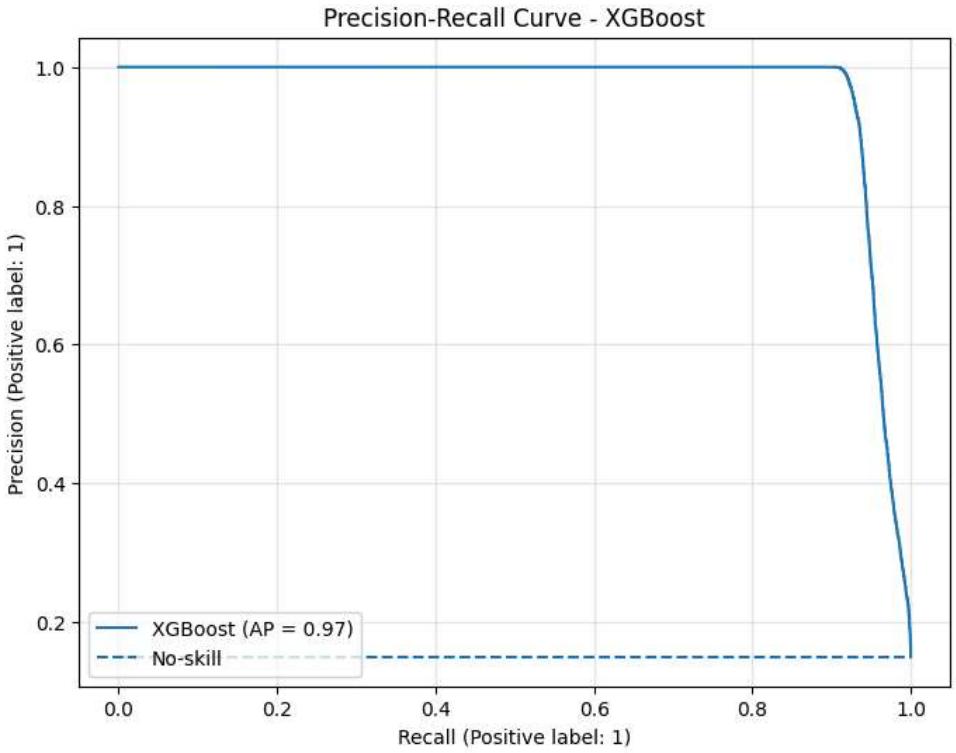
In [9]:
```
# XGBOOST MODELING ON SYNTHETIC DATA

from xgboost import XGBClassifier
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    roc_auc_score, RocCurveDisplay, PrecisionRecallDisplay
```

```
)

print("\n" + "="*80)
print("XGBOOST MODELING")
print("="*80)

# ------------------------------------------
# Train XGBoost
# ------------------------------------------
print("\n[Training XGBoost]...")

xgb_model = XGBClassifier(
    n_estimators=300,
    learning_rate=0.08,
    max_depth=8,
    subsample=0.8,
    colsample_bytree=0.8,
    eval_metric="logloss",
    random_state=SEED,
    n_jobs=-1,
    tree_method="hist"   # fast + RAM-efficient
)

xgb_model.fit(X_train_scaled, y_train)

print("✓ XGBoost trained")


# Evaluation
print("\n[Evaluation]...")

y_pred_xgb = xgb_model.predict(X_val_scaled)
y_proba_xgb = xgb_model.predict_proba(X_val_scaled)[:, 1]

results_xgb = pd.DataFrame({
    'Model': ['XGBoost (CTGAN-Enhanced)'],
    'Accuracy': [accuracy_score(y_val, y_pred_xgb)],
    'Precision': [precision_score(y_val, y_pred_xgb, zero_division=0)],
    'Recall': [recall_score(y_val, y_pred_xgb)],
    'F1': [f1_score(y_val, y_pred_xgb)],
    'AUC': [roc_auc_score(y_val, y_proba_xgb)]
})

display(results_xgb.round(4))

# ROC
fig, ax = plt.subplots(figsize=(8, 6))
RocCurveDisplay.from_predictions(
    y_val, y_proba_xgb, name="XGBoost", ax=ax
)
ax.plot([0,1], [0,1], 'k--')
ax.set_title("ROC Curve - XGBoost")
plt.grid(alpha=0.3)
plt.show()

# PR Curve
fig, ax = plt.subplots(figsize=(8, 6))
PrecisionRecallDisplay.from_predictions(
    y_val, y_proba_xgb, name="XGBoost", ax=ax
)
ax.hlines(y_val.mean(), 0, 1, linestyles="--", label="No-skill")
ax.legend()
ax.set_title("Precision-Recall Curve - XGBoost")
plt.grid(alpha=0.3)
plt.show()

print("\n✓ XGBoost evaluation complete")
```

```
================================================================================
XGBOOST MODELING
================================================================================

[Training XGBoost]...
✓ XGBoost trained

[Evaluation]...
```

	Model	Accuracy	Precision	Recall	F1	AUC
0	XGBoost (CTGAN-Enhanced)	0.9864	0.999	0.9098	0.9523	0.9844





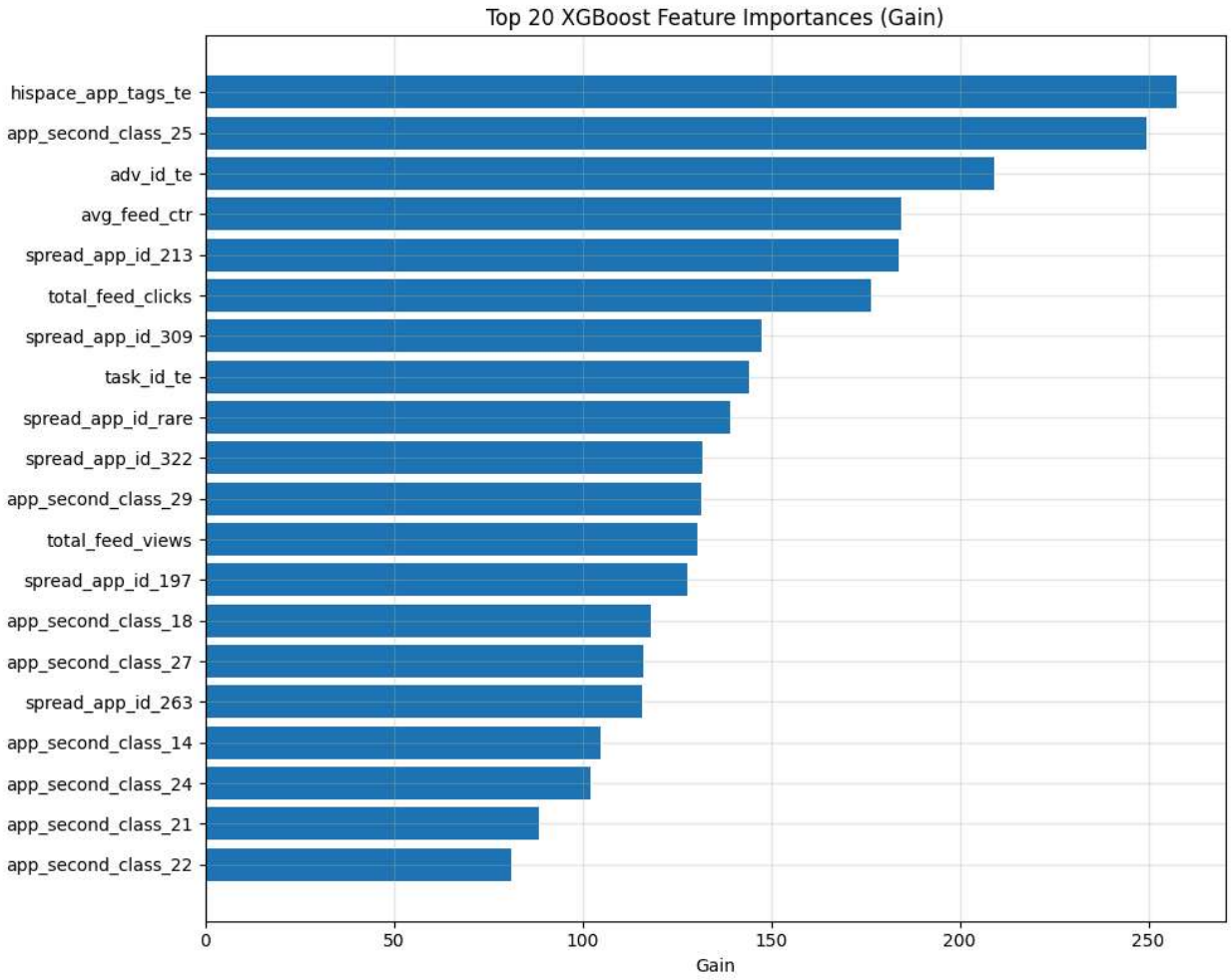
✓ XGBoost evaluation complete

In [15]:
```
# XGBOOST FEATURE IMPORTANCE


print("\n" + "="*80)
print("XGBOOST FEATURE IMPORTANCE (GAIN)")
print("="*80)

# Recover feature names from the scaler
xgb_feature_names = scaler.feature_names_in_

# Raw importance dict: keys = f0, f1, f213, ...
raw_importance = xgb_model.get_booster().get_score(importance_type='gain')

# Convert to DataFrame
rows = []
for k, v in raw_importance.items():
    # Extract index from "f123"
    idx = int(k[1:])
    if idx < len(xgb_feature_names):
```

```
        rows.append((xgb_feature_names[idx], v))

xgb_importance = pd.DataFrame(rows, columns=['feature', 'importance']) \
                   .sort_values('importance', ascending=False)

print("\nTop 20 most important XGBoost features:")
display(xgb_importance.head(20))

# Plot top 20
plt.figure(figsize=(10, 8))
top_n = 20
plt.barh(
    xgb_importance.head(top_n)['feature'][::-1],
    xgb_importance.head(top_n)['importance'][::-1]
)
plt.xlabel("Gain")
plt.title("Top 20 XGBoost Feature Importances (Gain)")
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```

```
================================================================================
XGBOOST FEATURE IMPORTANCE (GAIN)
================================================================================

Top 20 most important XGBoost features:
```

	feature	importance
9	hispace_app_tags_te	257.581512
215	app_second_class_25	249.414520
6	adv_id_te	208.993713
4	avg_feed_ctr	184.589050
232	spread_app_id_213	183.781128
3	total_feed_clicks	176.298584
243	spread_app_id_309	147.393265
5	task_id_te	144.016876
255	spread_app_id_rare	139.009735
247	spread_app_id_322	131.828903
219	app_second_class_29	131.405273
2	total_feed_views	130.498123
230	spread_app_id_197	127.807098
208	app_second_class_18	118.204842
217	app_second_class_27	116.278740
237	spread_app_id_263	115.676750
204	app_second_class_14	104.952896
214	app_second_class_24	102.264534
211	app_second_class_21	88.475838
212	app_second_class_22	80.987984


Top 20 XGBoost Feature Importances (Gain)
hispace_app_tags_te
app_second_class_25
adv_id_te
avg_feed_ctr
spread_app_id_213
total_feed_clicks
spread_app_id_309
task_id_te
spread_app_id_rare
spread_app_id_322
app_second_class_29
total_feed_views
spread_app_id_197
app_second_class_18
app_second_class_27
spread_app_id_263
app_second_class_14
app_second_class_24
app_second_class_21
app_second_class_22
0
50
100
150
200
250
Gain