ChatGPT

Comparative Study of DP Synthetic Data Generation: Model-First vs Stat-First – Four-Week Roadmap

Project Overview: This COMP430 group project will implement and compare two differentially private (DP) synthetic data generation pipelines on the UCI Adult Income dataset: (1) a **Model-First pipeline** using a Conditional Tabular GAN (CTGAN) with DP training (via DP-SGD), and (2) a **Stat-First pipeline** using privatized statistical modeling (e.g., releasing noisy marginals or a PrivBayes-style Bayesian network). Over four weeks, the team will build both systems in Python (Jupyter notebooks, NumPy, pandas, scikit-learn, PyTorch, etc.), and evaluate them using utility and privacy metrics. The roadmap below details weekly goals, tasks, member responsibilities, technical steps, code snippets, and tips for balancing privacy (ε) vs utility.

Week 1: Project Setup, Research, and Planning

Goals: Establish a strong foundation in differential privacy and synthetic data generation. Familiarize the team with CTGAN, DP-SGD, and statistical methods (marginals, MWEM, PrivBayes). Set up the development environment, obtain and preprocess the Adult dataset, and allocate tasks among team members.

1.1 Background Research and Knowledge Sharing

- **Differential Privacy (DP) Basics:** All members review DP fundamentals (ε, δ, noise addition, sensitivity). Understand how DP-SGD (Differentially Private Stochastic Gradient Descent) injects noise into model training [1]. Review how privacy loss is accounted (e.g., moments accountant) and the meaning of an $(ε, δ)$ guarantee.
- **CTGAN and Model-First Approach:** Study the CTGAN architecture for tabular data. CTGAN uses a conditional GAN that samples a column value ("condition") and trains the generator to produce data conditioned on that value [2]. This preserves feature dependencies and handles categorical imbalance via "training-by-sampling". Read the CTGAN paper (Xu et al. 2019) and note key techniques: mode-specific normalization for continuous features, Gumbel-softmax for categorical, and the conditional discriminator.
- **Stat-First Methods:** Research methods that first privatize statistics:
- *Noisy Marginals:* Adding Laplace noise to frequency counts or low-dimensional histograms of data, then generating samples consistent with those noisy distributions.
- *PrivBayes:* A DP Bayesian network approach. Understand how PrivBayes builds a DAG of features and uses noisy conditional distributions to generate data [3]. In PrivBayes, one releases noisy marginal counts to parameterize a Bayesian network of feature dependencies, then samples synthetic data from it.
- *MWEM (Multiplicative Weights Exponential Mechanism):* Iteratively improves a synthetic dataset so that it yields answers to a set of queries (e.g., marginals) close to the real data answers, under DP constraints. (Note: MWEM is complex; the team may use an existing library for MWEM due to time).
- **Evaluation Metrics:** Read about synthetic data evaluation:

- *TSTR (Train on Synthetic, Test on Real):* a model trained on synthetic data is evaluated on real holdout data [4]. If its performance is close to a model trained on real data (TRTR), the synthetic data retained useful statistical patterns [5].
- *Statistical Similarity:* Compare real vs synthetic distributions (mean, variance, correlations) and use measures like Jensen-Shannon Divergence or propensity score MSE (pMSE) — where a classifier tries to distinguish real vs synthetic records, with a lower pMSE meaning synthetic data is statistically closer to real.
- *Basic Privacy Checks:* Understand membership inference attacks (attempts to determine if a given real record was in the training set). While our methods are DP (provably limiting this risk), the team can still perform an empirical test as an extra sanity check.

Team Responsibilities:
Assign preliminary roles (flexible as needed): - *Member 1:* DP-SGD specialist – research Opacus and TensorFlow Privacy, how to track ε during training. - *Member 2:* GAN specialist – review CTGAN codebases (e.g., SDV's CTGAN, or PyTorch implementations) and figure out how to integrate DP. - *Member 3:* Stat model specialist – research PrivBayes code (e.g., DataSynthesizer library) and MWEM (e.g., OpenDP SmartNoise), plan how to implement one of these. - *Member 4:* Evaluation lead – design the evaluation framework (prepare train/holdout splits, decide classification tasks for TSTR, set up code for computing stats and possibly membership inference tests).

The team will cross-collaborate, but having point persons for each area ensures coverage. By end of Week 1, everyone should share key findings (perhaps in a short internal presentation or document) so all understand both pipelines.

1.2 Environment Setup and Data Preparation

- **Environment:** Set up a Python development environment (ensure all members can run Jupyter notebooks). Install needed libraries:
- PyTorch and **Opacus** for DP-SGD [6].
- NumPy, pandas, **scikit-learn** for data handling and evaluation.
- Possibly install **SmartNoise SDK** (`smartnoise-synth`) as a reference for stat-first methods (it provides DP synthesizers like MWEM and DP-CTGAN) [7], and/or **DataSynthesizer** for PrivBayes.
- If using TensorFlow, ensure TensorFlow Privacy is available (though this plan uses PyTorch/Opacus primarily).
- **Acquire the UCI Adult Dataset:** This dataset can be downloaded from the UCI Machine Learning Repository (often provided as `adult.data` and `adult.test` CSV files). Load it into pandas and combine train/test if needed (since we'll do our own splitting). For convenience, the team might treat the provided "test" as additional data or use only the one file and then split.

```
import pandas as pd

# Load Adult dataset (assuming it is in the current directory or provide
path)
columns = ["age", "workclass", "fnlwgt", "education", "education-num",
           "marital-status", "occupation", "relationship", "race", "sex",
           "capital-gain", "capital-loss", "hours-per-week", "native-
country", "income"]
adult_df = pd.read_csv("adult.data", names=columns, na_values="?",
```

```
skipinitialspace=True)
adult_df.head()
```

- **Preprocessing:** Clean the data:
- Handle missing values (Adult uses "?" as missing for some categorical fields — decide to drop or impute; dropping missing entries is common for this dataset).
- Normalize or scale continuous columns if needed. For CTGAN, we will use its internal transformations (CTGAN handles continuous via mode-specific normalization), but we still might bin some continuous features for the stat-first method.
- Encode categorical variables. CTGAN typically expects one-hot encoding for categories and uses an embedding to sample conditions. For stat-first (PrivBayes), categorical features can remain as label-encoded integers representing categories.
- **Label:** The `income` column ("<=50K" or ">50K") is the target for the predictive task. Convert it to binary (0/1) for evaluation ease.

```
# Basic preprocessing
adult_df = adult_df.dropna()  # drop rows with missing values for simplicity
# Convert target to 0/1
adult_df['income'] = adult_df['income'].apply(lambda x: 1 if '>50K' in x else
0)
# Example: Label-encode categoricals for stat-first methods
from sklearn.preprocessing import LabelEncoder
cat_cols = ["workclass","education","marital-status","occupation",
            "relationship","race","sex","native-country"]
for col in cat_cols:
    adult_df[col] = LabelEncoder().fit_transform(adult_df[col])
adult_df.head()
```

- **Data Exploration:** Do a quick exploratory data analysis to understand feature distributions (useful for debugging synthetic data later). For example, check the class balance of `income`, distribution of ages, etc. Ensure these insights are noted:
- E.g., "% of >50K in real data" – our synthetic data should hopefully preserve this roughly.

- Which features are categorical and how many categories each has (e.g., `education` has 16 categories, `native-country` has many, etc.), since CTGAN's conditional sampling will involve these.

- **Plan Privacy Budget:** Decide on initial privacy budget targets (ε, δ) for the project. Often δ is set to 1e-5 or 1e-6 for dataset of this size. For ε, common choices might be 1, 3, 5, or 10 for experimentation. The team might aim to produce synthetic sets at multiple privacy levels to see the trade-off:

- e.g., "high privacy" (ε≈1), "medium" (ε≈5), "low privacy" (ε≈10+ for near-non-DP baseline).
- **Note:** We won't finalize ε now, but keep these in mind as goals to test in Week 4. The privacy budget will be consumed differently by the two pipelines (DP-SGD vs noisy queries).

Week 2: Model-First Pipeline – Differentially Private CTGAN

Goals: Implement the CTGAN-based synthetic data generator with differential privacy. This includes building or adapting a CTGAN model and integrating Opacus for DP-SGD training. By end of Week 2, produce an initial synthetic dataset using the DP-CTGAN pipeline and verify basic functionality.

2.1 Implementing CTGAN Model (Generator & Discriminator)

- **Model Architecture:** Build the neural network components for CTGAN.
- *Generator:* a neural network that takes a noise vector z (and a conditional vector specifying a certain category value for a chosen feature) and outputs a synthetic data row (all features).
- *Discriminator:* a neural network that takes a data row (real or fake, also conditioned on the same feature's value) and outputs a probability of being real. The discriminator is "conditional" because it receives the sampled condition as input as well.
- **Conditional Sampling:** At each training step, CTGAN picks one discrete column as the "condition" and samples a value for it. Then it draws a real data example with that value to feed to the discriminator, and the generator must produce a fake example with that condition [2]. This strategy addresses category imbalance by oversampling rare categories in the condition selection.
- For our implementation, we can simplify by treating all categorical features as potential conditions (one is chosen uniformly or via frequency log sampling – but note: the OpenDP implementation found non-private frequency sampling problematic and defaulted to uniform for DP [8]). We may start with uniform selection for simplicity in DP.

- *Member 2* can start from an existing CTGAN code (e.g., from SDV or other open source) to avoid reinventing every detail. Focus on ensuring we understand the data transforms: one-hot encoding for categoricals, and the special normalization for continuous features (CTGAN uses a variational Gaussian mixture to handle multimodal continuous data).

- **Code – Define a simple Generator and Discriminator (PyTorch):**

```
import torch
import torch.nn as nn

# Example sizes (to be replaced with actual dims):
data_dim = adult_df.shape[1]  # number of features (including possibly one-
hot expanded dims)
noise_dim = 128               # length of random noise vector
cond_dim = 1                  # condition vector length (if one-hot for
categories, use that length)

class Generator(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, output_dim),
```

```
            nn.Sigmoid()  # assuming normalized output in [0,1] or can use
Tanh
        )
    def forward(self, noise, cond):
        # Concatenate noise and conditional vector
        x = torch.cat([noise, cond], dim=1)
        return self.net(x)

class Discriminator(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.LeakyReLU(0.2),
            nn.Linear(128, 64),
            nn.LeakyReLU(0.2),
            nn.Linear(64, 1),
            nn.Sigmoid()
        )
    def forward(self, data, cond):
        # Concatenate data and conditional vector
        x = torch.cat([data, cond], dim=1)
        return self.net(x)
}

G = Generator(noise_dim + cond_dim, output_dim=data_dim)
D = Discriminator(input_dim=data_dim + cond_dim)
```

Note: The above is a **simplified outline**. In practice, if using one-hot encodings, `data_dim` should account for expanded categorical dummy variables. The conditional vector would be a one-hot of the chosen category value. The architecture can be more complex (and CTGAN uses separate components to handle continuous vs categorical differently), but this gives a starting point.

- **Loss Functions:** Use the standard GAN losses (Binary Cross-Entropy loss for discriminator and generator objectives). CTGAN specifically uses training-by-sampling (condition strategy) and some tricks like balancing training on each category. We will not delve into those details here but ensure we at least implement the adversarial training loop.

2.2 Integrating Differential Privacy with Opacus

- **Opacus Privacy Engine:** We will train the GAN using DP-SGD. Opacus makes this easier by providing a `PrivacyEngine` that wraps the optimizer and data loader to clip gradients and add noise [1] [6]. The key steps:
- Determine the `batch_size` and `sample_rate`. For example, if our dataset (after preprocessing) has N records and we choose batch_size B, then sample_rate = B/N.
- Set the noise multiplier `σ` (std of noise) and clipping norm `C` (max per-sample gradient norm). These are hyperparameters impacting the privacy/utility trade-off. We might start with, say, `C = 1.0` and `σ = 1.0` (which is a common starting point).
- Attach the PrivacyEngine to our optimizer. During training, Opacus will clip gradients of each model parameter for each batch and add Gaussian noise.

- Track the cumulative privacy budget (ε) using Opacus' accountant.

- **Code – Initialize DP training with Opacus:**

```
from opacus import PrivacyEngine

# Set training parameters
batch_size = 64
learning_rate = 1e-3
noise_multiplier = 1.0  # initial guess, adjust later
max_grad_norm = 1.0
delta = 1e-5  # target delta

# Data loader for training data
from torch.utils.data import DataLoader, TensorDataset
train_data = TensorDataset(torch.tensor(X_train.values, dtype=torch.float32))
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)

# Initialize models and optimizer
G = Generator(noise_dim + cond_dim, data_dim)
D = Discriminator(input_dim=data_dim + cond_dim)
optimizer_G = torch.optim.Adam(G.parameters(), lr=learning_rate)
optimizer_D = torch.optim.Adam(D.parameters(), lr=learning_rate)

# Attach PrivacyEngine to the optimizer (for discriminator, as we train it on
real data)
privacy_engine = PrivacyEngine()
model_D, optimizer_D, train_loader = privacy_engine.make_private(
    module=D,
    optimizer=optimizer_D,
    data_loader=train_loader,
    noise_multiplier=noise_multiplier,
    max_grad_norm=max_grad_norm,
)
```

In the above, we apply DP to the **discriminator's** training (since that is directly trained on real data labels of real vs fake). We could also consider making the generator's training DP (as gradients flow through D to G), but typically ensuring D is trained with DP-SGD suffices to protect the real data. It's important to understand which parts of the GAN training access real data: the discriminator sees real data samples each iteration, so its gradient updates must be privatized. The generator's updates come from discriminator feedback and don't directly see real data, so making D's training DP should be the primary focus.

- **Training Loop:** Train in alternating steps: update D on real+fake data, then update G. Clip and noise are applied to D's gradients via Opacus. Monitor the training process (losses might be noisy due to DP noise). It's common to need more epochs for DP training because noise slows convergence. We also need to avoid very high learning rates which combined with noise can cause instability.

```
import torch.nn.functional as F

n_epochs = 5
for epoch in range(n_epochs):
    for real_batch in train_loader:
        real_batch = real_batch[0]  # TensorDataset yields tuple
        # Sample a random condition (choose a categorical feature and a
value)
        # For simplicity, assume a fixed condition feature index or cycle
through them
        cond_feature = 1  # e.g., 'education' column index as condition
        # Sample a value for condition from uniform or weighted distribution
        cond_val = torch.randint(0, num_categories[cond_feature],
(batch_size,))
        # Build condition vector (one-hot encode cond_val)
        cond_vec = F.one_hot(cond_val,
num_classes=num_categories[cond_feature]).float()
        # Split real_batch into features and get that feature's one-hot for D
        real_data = real_batch.clone()
        real_data = real_data  # (already floats scaled 0-1 perhaps)
        real_cond = cond_vec  # one-hot condition for discriminator input

        # Generate fake data with G
        noise = torch.randn(batch_size, noise_dim)
        fake_data = G(noise, cond_vec)
        fake_cond = cond_vec  # generator used same cond for fake

        # Train Discriminator: maximize log(D(real)) + log(1-D(fake))
        optimizer_D.zero_grad()
        real_preds = D(real_data, real_cond)
        fake_preds = D(fake_data.detach(), fake_cond)
        loss_D = - (torch.log(real_preds + 1e-8).mean() + torch.log(1 -
fake_preds + 1e-8).mean())
        loss_D.backward()
        optimizer_D.step()  # Opacus will clip and add noise here
automatically

        # Train Generator: maximize log(D(fake)) (or minimize -log(D(fake)))
        optimizer_G.zero_grad()
        fake_preds = D(fake_data, fake_cond)  # fresh prediction
        loss_G = - torch.log(fake_preds + 1e-8).mean()
        loss_G.backward()
        optimizer_G.step()

    # End of epoch - track privacy and losses
    epsilon = privacy_engine.get_epsilon(delta)
    print(f"Epoch {epoch+1}: ε = {epsilon:.2f}, Loss_D = {loss_D.item():.
4f}, Loss_G = {loss_G.item():.4f}")
```

This pseudocode illustrates the training loop with DP. **Important:** In practice we need to ensure that `privacy_engine` is tracking the correct number of samples. Opacus requires either specifying sample_rate or providing the DataLoader length. The code above uses `make_private` on the DataLoader which should internally compute sample_rate = batch_size / len(train_data).

- **Privacy Tracking:** Using `privacy_engine.get_epsilon(delta)` allows us to see how ε grows with each epoch. We should stop training once we hit our target ε. Alternatively, decide on fixed epochs and compute resulting ε. For instance, if `noise_multiplier=1.0`, `sample_rate=0.1` (10% each batch), after a certain number of steps we might get ε around a few units. *(Example: Using Opacus's script, DP-SGD with 1% sample rate, noise 1.0, 3 epochs gives ε≈2.39 for δ=1e-5* [9] *.)* Our setup will differ, but we use such references to guide our parameter choices.

- **Generate Synthetic Data:** After training, use the generator to sample synthetic records:

- Loop to generate desired number of records (e.g., same number as original dataset or a proportion thereof).
- For each record, sample a random noise vector and also randomly pick a conditional feature and value (the generator was trained with conditional inputs; one approach is to randomly pick a condition for each sample, or iterate through conditions evenly).
- Alternatively, to generate unconditionally, one can sample a condition according to its marginal probability and feed it (this ensures synthetic distribution follows real distribution of that feature).
- Use the generator's output and then invert any preprocessing to get data in original format (e.g., map one-hot back to category labels, inverse normalize continuous values).

```
# Synthesize new data
G.eval()
synthetic_data = []
num_samples = len(adult_df)  # generate same number as original
for i in range(num_samples):
    # Choose a random condition feature and value
    cond_feature = 1  # (for simplicity, fix one feature here or randomize)
    cond_val = torch.randint(0, num_categories[cond_feature], (1,))
    cond_vec = F.one_hot(cond_val,
num_classes=num_categories[cond_feature]).float()
    noise = torch.randn(1, noise_dim)
    fake = G(noise, cond_vec)
    synthetic_data.append(fake.detach().numpy()[0])
synthetic_data = np.array(synthetic_data)
# Map synthetic_data back to human-readable form (inverse normalization,
etc.)
```

- **Team Checkpoint:** By mid-to-late Week 2, *Member 2* (with help from others) should have a working DP-CTGAN prototype. *Member 1* assists in tuning DP hyperparameters (if training diverges, try increasing noise gradually from a smaller value or lowering the learning rate). Everyone should review a small sample of the synthetic data to ensure it's not completely nonsensical (e.g., check ranges of numeric fields, ensure categories are valid). Don't worry if quality is initially poor under strict DP; we will refine and compare in Week 4.

2.3 Initial Utility Checks (TSTR Dry-Run)

- If time permits in Week 2, do a quick preliminary TSTR test with the synthetic data from DP-CTGAN:
- Train a simple classifier (e.g., logistic regression or random forest) on the synthetic data (with `income` as label).
- Evaluate on a holdout of real data.
- Compare to training the same classifier on real training data and testing on holdout (this is TRTR for baseline).
- This quick check gives an early sense of utility. For example:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

# Split real data into train/holdout for evaluation
real_train, real_holdout = train_test_split(adult_df, test_size=0.2,
random_state=42)
X_real_train = real_train.drop("income", axis=1); y_real_train =
real_train["income"]
X_holdout = real_holdout.drop("income", axis=1); y_holdout =
real_holdout["income"]

# Train on synthetic (from CTGAN) and test on real holdout
X_synth = pd.DataFrame(synthetic_data, columns=processed_columns)  #
synthetic features
# (Ensure synthetic data is scaled or encoded same as real; convert to
DataFrame and apply same encoding mappings if needed)
clf = RandomForestClassifier()
clf.fit(X_synth, y_real_train.sample(len(X_synth), replace=True))  # need
labels for synthetic; we can use real train labels randomly or a model to
label synth data if unsupervised synth
score_synth = clf.score(X_holdout, y_holdout)

# Train on real and test on real (baseline)
clf2 = RandomForestClassifier()
clf2.fit(X_real_train, y_real_train)
score_real = clf2.score(X_holdout, y_holdout)
print(f"TSTR Accuracy - Model trained on synthetic: {score_synth:.3f}, on
real: {score_real:.3f}")
```

- *Note:* Truly, synthetic data generation as we do is **unsupervised**, it doesn't guarantee the synthetic records come with labels (since CTGAN didn't explicitly use the `income` label in generation). To apply TSTR properly, one approach is to include the target as a feature in generation (so the GAN generates `income` as another column). We could do that, treating `income` like any other column to be synthesized. If we do include `income` in the GAN, then the synthetic data will have labels which we can directly use for TSTR. Alternatively, we could train a classifier on synthetic *features* to predict real labels by pairing each synthetic record with a label from a real record (though that random pairing is not ideal). **Preferred**: include `income` in the data passed to CTGAN so that synthetic data includes an `income` value for each record.

- This quick test can be done once we have some synthetic data, to catch glaring issues (if synthetic data yields near-random accuracy, that might be expected if ε is very low, but if it's too low utility, maybe we adjust some parameters).
- Save the synthetic dataset for formal evaluation in Week 4.

Week 3: Stat-First Pipeline – Privatized Statistical Generation

Goals: Implement the stat-first DP synthetic data pipeline. Choose an approach (noisy marginals, PrivBayes, or MWEM) and carry it out to create synthetic data under a given privacy budget. By end of Week 3, have a second synthetic dataset (or multiple, if trying variants) to compare with the GAN approach.

3.1 Choosing and Designing the Stat-First Method

- Given time constraints, the team decides to implement **PrivBayes (DP Bayesian network)** as the stat-first method, since it strikes a balance between simplicity and ability to capture correlations. (Optionally, if someone deeply studied MWEM and it seems feasible via a library, that could be attempted too. But here we outline PrivBayes.)

PrivBayes Approach Recap:
PrivBayes generates a synthetic dataset by: 1. **Structure Learning with DP:** Determine an ordering of attributes and a Bayesian network structure (each node has at most one parent in the simplified version) using DP. Typically, mutual information between attribute pairs is used to decide edges (this would require DP estimates of pairwise mutual info or correlations). 2. **Noisy Marginal Release:** Release the distribution of the first attribute (root) with noise, and for each subsequent attribute, release the conditional distribution given its parent, all under DP. These are essentially noisy contingency tables. 3. **Sampling:** Draw synthetic data by first sampling the root from its noisy distribution, then iterating through the network ordering and sampling each attribute from the noisy conditional distribution given its parent's sampled value.

We will simplify: - Use a predetermined structure (to avoid complex DP structure learning). For example, choose an ordering of features by domain knowledge or randomly. *Member 3* might identify that certain features strongly influence others (e.g., `education -> occupation`, or `marital-status -> relationship`), and set those as parent-child for demonstration. - Ensure all features are discrete. Continuous ones (like `age`, `hours-per-week`, capital gains) we **bin into ranges** to make them categorical. (Adult data: age could be binned into say 10-year bins; hours per week into low/medium/high, etc., or even treat them as categorical values if moderate distinct values). - Allocate privacy budget across all released distributions. For instance, if total $\varepsilon = 1.0$ for this method: - Use ε_1 for the root distribution, ε_2 for the next feature's conditional distribution, etc., such that sum(ε_i) = 1.0 (or use composition theorems appropriately). A simple equal split or proportional split can be done. δ can be same overall δ as before.

Team Decision: For demonstration, suppose we pick 3 features to model (to keep it small in code example): `education` (root), `occupation` (child of education), and `income` (child of occupation). This is a tiny Bayesian network: `education -> occupation -> income`. In practice, we would include more features, but this will illustrate the process.

3.2 Computing Noisy Marginals (DP Queries)

- *Member 3* writes code to compute the required distributions with Laplace or Gaussian noise:

- E.g., $P(\text{education})$ (distribution of education levels in data) with Laplace noise on each count.
- $P(\text{occupation} \mid \text{education})$ (distribution of occupation given each education level) with noise.
- $P(\text{income} \mid \text{occupation})$ with noise.

The sensitivity of count queries here is 1 (each person contributes to one count). Using Laplace mechanism: add i.i.d. Laplace(scale = 1/ε_i) noise to each count in the histogram. If counts are large, the relative noise might be small; for low counts or many categories, noise might distort more.

- **Code – Noisy marginal example:** (Using our simplified 3-feature example)

```
import numpy as np

# Assuming adult_df has columns 'education', 'occupation', 'income' (encoded
as integers)
epsilon_total = 1.0
# Split epsilon among distributions (just an example)
epsilon1 = 0.4  # for P(education)
epsilon2 = 0.4  # for P(occupation | education)
epsilon3 = 0.2  # for P(income | occupation)

# 1. Noisy distribution for education
edu_counts = adult_df['education'].value_counts().sort_index().to_numpy()
noisy_edu_counts = edu_counts + np.random.laplace(loc=0, scale=1/epsilon1,
size=edu_counts.shape)
noisy_edu_counts = np.clip(noisy_edu_counts, a_min=0, a_max=None)  # no
negative counts
P_education = noisy_edu_counts / noisy_edu_counts.sum()

# 2. Noisy conditional distribution for occupation given education
# Compute occupation counts for each education level
occupation_given_edu =
adult_df.groupby(['education','occupation']).size().unstack(fill_value=0)
noisy_occ_given_edu = occupation_given_edu + np.random.laplace(loc=0,
scale=1/epsilon2, size=occupation_given_edu.shape)
noisy_occ_given_edu = np.clip(noisy_occ_given_edu, 0, None)
# Normalize conditional distributions: each education row sums to 1
P_occ_given_edu = noisy_occ_given_edu.div(noisy_occ_given_edu.sum(axis=1),
axis=0)

# 3. Noisy conditional distribution for income given occupation
income_given_occ =
adult_df.groupby(['occupation','income']).size().unstack(fill_value=0)
noisy_inc_given_occ = income_given_occ + np.random.laplace(loc=0, scale=1/
epsilon3, size=income_given_occ.shape)
noisy_inc_given_occ = np.clip(noisy_inc_given_occ, 0, None)
P_inc_given_occ = noisy_inc_given_occ.div(noisy_inc_given_occ.sum(axis=1),
axis=0)
```

In this snippet: - `P_education` is an array of probabilities for each education category. - `P_occ_given_edu` is a DataFrame where each row (education) gives a probability distribution across occupations. - `P_inc_given_occ` is a DataFrame where each row (occupation) gives distribution of income values (0 or 1).

We used Laplace mechanism for simplicity. We should document that using Gaussian mechanism with δ could be an alternative (especially if we want to leverage better composition via advanced composition or RDP accountant), but Laplace is straightforward for counts.

3.3 Sampling Synthetic Data from the Noisy Model

- With the noisy distributions, generate synthetic records by sampling in topological order (root to leaves):
- Sample an `education` value from `P_education`.
- Sample an `occupation` given that education from `P_occ_given_edu`.
- Sample an `income` given that occupation from `P_inc_given_occ`.
- (If more features were in the network, continue accordingly.)

- Repeat for the desired number of records.

- **Code – Generate synthetic data from PrivBayes model:**

```
synthetic_records = []
num_samples = len(adult_df)
education_values = list(range(len(P_education)))  # assuming categories are
0..n-1
for _ in range(num_samples):
    # 1. Sample education
    edu = np.random.choice(education_values, p=P_education)
    # 2. Sample occupation given education
    occ_dist = P_occ_given_edu.loc[edu].to_numpy()
    occ_categories = list(range(len(occ_dist)))
    occ = np.random.choice(occ_categories, p=occ_dist)
    # 3. Sample income given occupation
    inc_dist = P_inc_given_occ.loc[occ].to_numpy()
    inc_categories = list(range(len(inc_dist)))
    inc = np.random.choice(inc_categories, p=inc_dist)
    synthetic_records.append([edu, occ, inc])

synth_df = pd.DataFrame(synthetic_records,
columns=['education','occupation','income'])
# Inverse transform if needed (here they are still label-encoded integers,
which might be fine for analysis, or map back to original labels if desired)
```

- *Member 3* should expand this approach to all features (or a rich subset) rather than just 3. However, adding too many features in a single network increases complexity exponentially if each node had multiple parents. PrivBayes typically restricts parent count to 1 or 2 to manage this. Our example used at most 1 parent. We could chain more features (making essentially a tree structure). For instance: `education -> occupation -> income -> workclass -> ...` a simple chain. This won't capture all correlations (a tree can't represent all pairwise

dependencies), but it's a reasonable compromise. The team might choose a more meaningful structure (e.g., split into a few sub-networks or a tree based on known relationships).

- **Privacy Accounting:** Ensure the total ε used by summing parts does not exceed the target. In our example, ε_total = 1.0 was split into 0.4+0.4+0.2. By basic composition, that sums to 1.0 (we ignore δ here for Laplace mechanism, or assume δ small if using advanced composition). If using Gaussian mechanism or advanced composition, one could compute total ε for given δ properly. The team should document the allocation of budget clearly in the report.

3.4 (Optional) Utilizing Existing Tools

If implementing PrivBayes from scratch is too time-consuming, consider using or referencing existing libraries: - **DataSynthesizer** (from DataResponsibly project) has a PrivBayes implementation in Python. - **SmartNoise Synthesizers:** The OpenDP SmartNoise library (`snsynth`) can train MWEM or PrivBayes synthesizers with a few lines [7]. For example:

```
from snsynth import Synthesizer
synth = Synthesizer.create('privbayes', epsilon=1.0)
synth.fit(adult_df)  # assuming adult_df is all categorical or discretized
synth_data = synth.sample(len(adult_df))
```

This would automate a lot, but using it directly might defeat some learning objectives. A compromise is to use it to validate our results or to compare if we have time.

- *Member 3* and others ensure the stat-first pipeline is producing *some* output by end of Week 3. Check basic sanity of the synthetic dataset:
- Do synthetic marginals roughly align with the noisy inputs we gave? (They should, by construction.)
- Are all values valid (no out-of-domain values)?
- For example, compare the distribution of `education` in synthetic vs real: since we intentionally matched the noisy distribution, it should be close to real distribution plus noise differences.
- The more telling differences will be in joint correlations that the simple network might miss (e.g., our model might not directly connect `education` and `income`, except through occupation).

Week 4: Evaluation, Tuning, and Comparison

Goals: Evaluate the two pipelines side-by-side on utility and privacy metrics. Tune parameters if needed to achieve a fair comparison (e.g., both using ε ~ 1 or another common budget). Produce final results (tables, plots) and document findings. Each member contributes to analyzing specific aspects.

4.1 Defining Evaluation Setup

- **Data Splits:** Use the original Adult dataset split (or the one created in Week 2) for evaluation:
- Holdout a test set of real data that was *not* used in training the synthetic generators. If we used all data to generate synth, then for TSTR we should at least have a separate holdout for testing models. We did earlier a 80/20 split (train vs holdout). We can stick to that.
- We have:
  - `Real_train` (used to train CTGAN and to compute privBayes distributions).
  - `Real_holdout` (only used for evaluating model performance).

    - `Synthetic_data_modelFirst` generated from CTGAN (size maybe equal to Real_train).
    - `Synthetic_data_statFirst` generated from PrivBayes (size also similar).
- **Utility Metrics:**
- *TSTR Classification Performance:* As described, train a classifier on synthetic data, test on real holdout. Do this for both synthetic sets. Compare with baseline of training on real_train and testing on holdout (TRTR). We can use accuracy for the income prediction (since it's a binary classification), or AUC if we want more nuance.
- *Visualization & Statistical Checks:* Plot distributions of key features for real vs each synthetic. For numeric features like age or hours, perhaps compare their histograms. For categorical, perhaps a bar chart of category frequencies. Calculate correlation matrices for real and synthetic datasets and see differences (e.g., use Pearson's correlation on one-hot encoded data or appropriate measures for categorical).
- *pMSE / Propensity Score Test:* Merge real_train and each synthetic dataset, label one as 0 and one as 1, and train a logistic regression to classify them [10] . The idea: if the classifier cannot distinguish well (prediction near random 0.5), the synthetic data is very close to real distribution. Compute the AUC or accuracy of this discriminator:
    - If accuracy is high (>>50%), the synthetic data has distinguishable patterns (lower fidelity).
    - We can report something like "A logistic classifier could distinguish real vs model-first synthetic with 70% accuracy, vs distinguishing real vs stat-first synthetic with 80% accuracy" – lower is better for privacy and fidelity.
    - pMSE is essentially the mean squared error of the propensity (predicted probability) minus 0.5; for simplicity, just reporting classifier accuracy or AUC is understandable.

- *Membership Inference Attack:* Since both methods are DP, membership inference success should be theoretically bounded. But we can simulate a naive attack:

    - Take some records from real_train and some from real_holdout (not used in training).
    - See if synthetic data contains any record identical or very close to a real_train record (direct memorization would be a big red flag, though DP should prevent exact copies).
    - Train an attack model: for example, train a model on the classifier's posteriors or just on whether each record appears in synthetic data (as a simplistic rule: "if a record is in synthetic, guess it was in training"). However, exact matches in continuous domain unlikely. Instead, use a distance-based approach: For each real record, find the nearest neighbor in synthetic data. If the distance is below a threshold, mark that as member guess = 1. Evaluate how many training members are identified vs holdout.
    - This is somewhat advanced; a simpler approach:
    - Use the combined classifier method above: label all real_train as members, and real_holdout as non-members, then see if a model can classify them using information from synthetic data. For instance, add a feature like "distance to nearest synthetic neighbor" or "likelihood under synthetic model" and see if that differentiates train vs holdout. If membership inference attack accuracy is no better than random ~50%, that's good.
    - Due to time, a basic check can be done: ensure no verbatim copies. Or perform the logistic regression approach:

```
# Membership inference via logistic regression on presence in
synthetic
real_train['is_member'] = 1
real_holdout['is_member'] = 0
mi_df = pd.concat([real_train, real_holdout])
```

```
# Feature: distance to nearest synthetic record (for each method)
# We'll compute for each record the min Euclidean distance to any
synthetic record
import numpy as np
from sklearn.neighbors import NearestNeighbors

X_model_syn = synthetic_modelFirst.drop('income', axis=1)  #
features only
nbrs = NearestNeighbors(n_neighbors=1).fit(X_model_syn)
dist_model, _ = nbrs.kneighbors(mi_df.drop(['income','is_member'],
axis=1))
mi_df['dist_modelfirst'] = dist_model

X_stat_syn = synthetic_statFirst.drop('income', axis=1)
nbrs2 = NearestNeighbors(n_neighbors=1).fit(X_stat_syn)
dist_stat, _ = nbrs2.kneighbors(mi_df.drop(['income','is_member'],
axis=1))
mi_df['dist_statfirst'] = dist_stat

# Train attack model (logistic) to predict is_member using
distances
attack_features = mi_df[['dist_modelfirst','dist_statfirst']]
attack_labels = mi_df['is_member']
att_clf = LogisticRegression().fit(attack_features, attack_labels)
att_score = att_clf.score(attack_features, attack_labels)
print(f"Attack model training accuracy: {att_score:.2f}")
```

    - If the attack model finds that training records tend to be closer to synthetic data than holdout (thus >50% accuracy), that indicates potential overfitting of synthetic data to training set. Ideally, DP would prevent this, yielding ~50% attack accuracy (random guessing).
    - *Member 4* can lead this membership inference test and interpret results with caution.

- **Team Breakdown:**

- *Member 4 (Evaluation lead):* Runs the TSTR experiments for both pipelines, including training classifiers and computing accuracy metrics. Also generates comparison plots for distributions.
- *Member 1:* Assists with computing privacy metrics, e.g., verifying that the DP-CTGAN's actual ε (as reported by Opacus) matches the intended budget, and calculating the theoretical ε used in PrivBayes. Also helps interpret the membership inference results.
- *Member 2:* Focus on utility analysis: examine which pipeline yields better utility at the same privacy level. Possibly try training a more complex model (e.g., XGBoost classifier) to see if synthetic data supports complex models.
- *Member 3:* Focus on quality of the stat-first synthetic data: are certain feature relationships preserved or lost? They can compute additional statistics (e.g., compare synthetic and real for a pair of features that were not explicitly connected in the Bayesian network to see if correlation dropped).

4.2 Parameter Tuning and Iteration

- After initial evaluation, the team may adjust parameters to improve outcomes:

- **Tuning DP-CTGAN:**
  - If utility is very low, consider increasing the noise multiplier a bit less (trading some privacy for utility) or increasing batch size (larger batch -> lower sample_rate -> less privacy cost per epoch). Remember increasing epochs increases ε, so perhaps find a sweet spot.
  - Check if the GAN is underfitting or mode-collapsing (maybe the DP noise is too high). Potentially try a smaller noise multiplier (with careful note of resulting ε).
  - Experiment with the conditional sampling strategy: the GitHub discussion noted removing the log-frequency sampling hurt utility [11]. Maybe try a mild non-uniform sampling of conditions that is still DP (e.g., sample condition proportional to a DP noisy count of category frequency).
  - Optimize learning rate or use gradient clipping tweaks (Opacus allows per-layer clipping, etc., but that might be advanced).
- **Tuning PrivBayes:**
  - If the stat-first method utility is low, perhaps the network structure is too simple. Maybe allow one more parent for a feature (though implementing 2-parent PrivBayes is harder due to 3D distributions).
  - Alternatively, try the **Independent marginals** approach as a baseline: i.e., each feature independently with noise (ε per feature) and then sample by just independent draws. This will likely have poor joint utility but is a privacy-safe baseline. Compare with PrivBayes to see the value of modeling correlations.
  - Try an existing synthesizer (like SmartNoise MWEM) on a subset of data to see if it yields better utility, for discussion.

- **Epsilon sweep:** If time, generate synthetic data under different ε values for one method (say DP-CTGAN at ε=1, 5, 10) to show the utility curve. Similarly for PrivBayes. This can be a short study within the project to illustrate the privacy-utility trade-off explicitly (could be a nice chart: ε vs accuracy).

- **Note:** It's important both pipelines operate under *comparable privacy budgets* for a fair study. If DP-CTGAN ended up with ε=2 and PrivBayes exactly ε=1 due to how we allocated, we might adjust one or the other. Ideally, fix an ε (like 1 or 2) and ensure both are at that level (DP-CTGAN by training time/noise, PrivBayes by noise scale). If one method fundamentally can't work at a very low ε, document that challenge.

4.3 Results and Documentation

- **Collate Results:** Summarize all findings clearly. Possible artifacts:
- Table of TSTR accuracy for each method (with possibly multiple ε settings).
- Bar chart of distribution differences for a few features.
- Plot of real vs synthetic cumulative distributions for continuous features (to see shape differences).
- Perhaps a small table of correlation coefficients real vs synthetic for select feature pairs.
- Table or statement on membership inference attack accuracy (e.g., "Attacker success ~50% for both methods, indicating no significant leak beyond random guessing" or any differences observed).
- Privacy budget used: report ε, δ for each method and how they were calculated, giving confidence that both approaches indeed satisfy DP.
- **Analysis:** Discuss reasons for results:

- Did the model-first (GAN) approach yield higher utility? Often GANs can capture complex patterns better, but DP noise might have hindered it. If CTGAN did better on capturing certain interactions, point that out (maybe because it tries to model full joint distribution).
- Did the stat-first approach excel in some aspects? Perhaps it exactly preserved some low-dimensional distributions it was built on (by design) – e.g., if PrivBayes modeled (education, occupation) well, then any analysis of the relationship between education and occupation in synthetic data will be strong.
- Compare the difficulty of implementation and runtime: GAN training is iterative and possibly slow (especially with DP-SGD overhead), whereas PrivBayes might have been faster or vice versa (if MWEM was used, it could be heavy).
- Highlight the **trade-off between privacy and utility:** e.g., "At ε=1, both methods produced usable synthetic data, but DP-CTGAN's classifier accuracy was 5% higher than PrivBayes. However, DP-CTGAN training took longer and was trickier to stabilize. At ε=0.5 (more privacy), both methods saw utility drop, with PrivBayes suffering less in utility than GAN" (hypothetical insights).
- If any method struggled (e.g., GAN collapsed for very high noise), mention that and how the simpler stat-first might be more robust in extreme privacy regimes.

- Discuss any ethical or fairness considerations noticed (optional): For instance, did DP or synthetic generation affect certain subgroups differently? (Adult has sensitive attributes like race/sex – differential effects could be a discussion point if data shows it).

- **Final Deliverables:** Each member ensure their part is documented in the final report notebook:

- Include code snippets (as we've outlined) for key steps so the reader can see how it was done.
- Present results with clear headings and explanations.
- The report should read like a guide, so future students can follow the roadmap to implement a similar project.

Below, we illustrate the TSTR evaluation framework with a diagram and explain the process, as it is a central evaluation method we use:

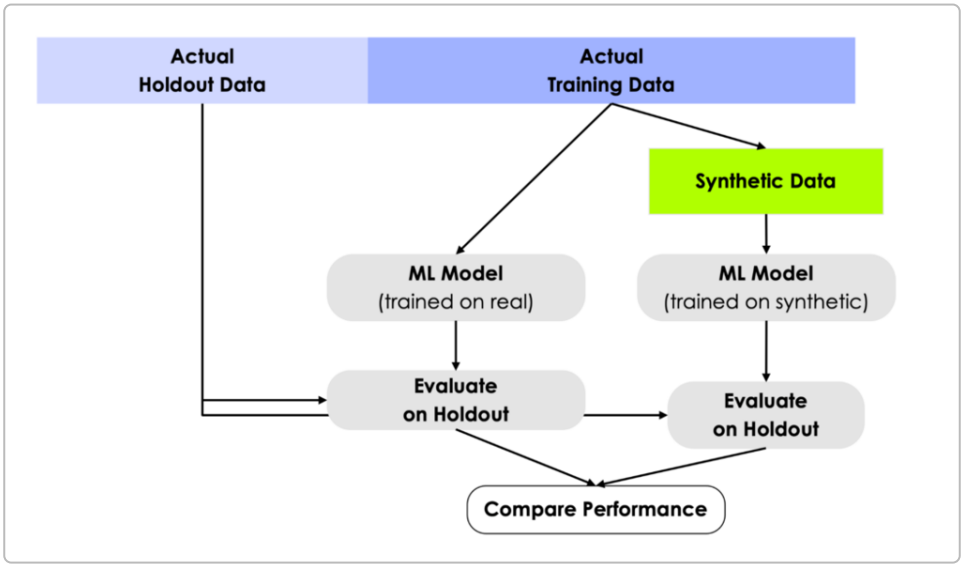

Figure: The Train-Synthetic-Test-Real (TSTR) evaluation workflow. We train one ML model on the synthetic data and another on the real training data, then evaluate both models on the real holdout set [4] [12]*. By*

comparing their performance, we assess how well the synthetic data retains patterns from real data (if the model trained on synthetic performs close to the one trained on real, the synthetic data is of high utility).

4.4 Wrap-Up and Lessons Learned

- **Group Discussion:** At end of week, have a debrief meeting:
- Each member shares what they found challenging or interesting (e.g., Member 2 might note how adding noise via DP-SGD made GAN training unstable and what tricks were needed; Member 3 might share how adding a second parent in PrivBayes drastically increased complexity).
- Discuss real-world relevance: e.g., for releasing a DP synthetic version of Adult data, which approach would we trust more or prefer, and why? (CTGAN might produce more realistic data but PrivBayes is easier to analyze theoretically, etc.)
- Identify any potential extensions if more time: like trying PATE-GAN (another model-first DP method) or an advanced stat method like HRGM or others.
- **Documentation:** Finalize the project notebook/report with clear structure:
- Introduction, Methodology (describing both pipelines), Experiments, Results, Conclusion.
- Ensure code is clean and well-commented, so that the instructor or others can reproduce results.
- Provide references to libraries or papers where appropriate (for instance, cite Opacus for DP-SGD [6], cite PrivBayes paper [3], etc., as we have done throughout this guide).
- **Final Tip:** Emphasize in the report the importance of **tuning privacy parameters**. Many times, students expect a DP method to "just work," but often it requires careful tuning (finding a good noise level, sufficient data preprocessing, etc.). We balanced ε and utility by multiple trials – this is a key takeaway: achieving both privacy and utility is a delicate balance, and comparing two fundamentally different approaches helped illustrate this.

By the end of Week 4, the team will have a comprehensive comparison of model-first vs stat-first DP synthetic data generation on a real dataset, with practical insights into the implementation of each.

---

[1] ml-research.github.io
https://ml-research.github.io/papers/fang2022dpctgan.pdf

[2] Overview of the CTGAN model. A condition is sampled first and passed to... | Download Scientific Diagram
https://www.researchgate.net/figure/Overview-of-the-CTGAN-model-A-condition-is-sampled-first-and-passed-to-the-conditional_fig2_355093029

[3] Techniques - CRC
https://pages.nist.gov/privacy_collaborative_research_cycle/pages/techniques.html

[4] [12] Evaluate synthetic data quality using downstream ML - MOSTLY AI
https://mostly.ai/blog/synthetic-data-quality-evaluation

[5] How to evaluate the quality of the synthetic data – measuring ... - AWS
https://aws.amazon.com/blogs/machine-learning/how-to-evaluate-the-quality-of-the-synthetic-data-measuring-from-the-perspective-of-fidelity-utility-and-privacy/

6 GitHub - pytorch/opacus: Training PyTorch models with differential privacy
https://github.com/pytorch/opacus

7 GitHub - opendp/smartnoise-sdk: Tools and service for differentially private processing of tabular and relational data
https://github.com/opendp/smartnoise-sdk

8 11 DPCTGAN · opendp opendp · Discussion #1241 · GitHub
https://github.com/opendp/opendp/discussions/1241

9 Opacus · Train PyTorch models with Differential Privacy
https://opacus.ai/api/compute_dp_sgd_privacy.html

10 On the Quality of Synthetic Generated Tabular Data - MDPI
https://www.mdpi.com/2227-7390/11/15/3278