Pipeline and Goals

1. Each student implements **a generative model for tabular data**.
2. Their model should:
   - take **raw data**,
   - **discretize** (or otherwise preprocess) it,
   - **generate synthetic tabular data**,
   - **evaluate quality via TSTR** (train on synthetic, test on real),
   - **save outputs + metrics** to disk.
3. All students must share:
   - the **same train/test split**,
   - the **same preprocessing tools** (for fairness),
   - the **same evaluation pipeline and metrics**.

You will implement **your own generative model for tabular data**.
Your model will:

1. Take a **raw tabular dataset** (CSV).
2. **Discretize / preprocess** it using the **shared tools**.
3. **Train a generative model** on the training split.
4. **Generate synthetic tabular data**.
5. Evaluate the synthetic data by:
   - training classifiers on synthetic data,
   - testing on **real** data (TSTR: Train on Synthetic, Test on Real),
   - saving all outputs & metrics to disk.

All students **must use the same pipeline and utilities** so results are comparable.

Codebase Components You MUST Use

Below is the pipeline you must follow, with exact functions + file locations.

1- Preprocessing / Discretization (from RAW → DISCRETE CSV)

You are NOT allowed to write your own discretizer.
You must use the shared function.

- **What to use**
    - discretize_preprocess(file_path, output_path, bins=10, strategy='uniform')
    - **File:** utils.py
- **What it does**
    - Loads raw CSV (file_path).
    - Treats '?' as missing and drops all-NaN columns.
    - Splits into:
        - X = all columns except the last,
        - y = last column (assumed to be the target/label).
    - For **numerical features**:
        - converts to numeric,
        - fills missing values with the column median,
        - discretizes into integer bins using KBinsDiscretizer.
    - For **categorical features**:
        - fills missing with 'Missing',
        - encodes with LabelEncoder.
    - For the **target column**:
        - fills missing with 'Missing',
        - encodes with LabelEncoder.
    - Saves a fully **discrete integer** dataset to output_path.

So your first step is always something like:

```
from utils import discretize_preprocess

discretize_preprocess(
    file_path="raw_data/mydataset.csv",
    output_path="discretized_data/mydataset.csv",
    bins=10,
    strategy="uniform",
)
```

Preprocessing / discretization

- Discretize raw data
    - `discretize_preprocess(file_path, output_path, bins, strategy)`
      → **file:** `utils.py`
    - Takes raw CSV, cleans it, discretizes numeric columns into integer bins, label-encodes categoricals & target, and saves a **fully discrete** dataset.
- Alternate continuous+encoded preprocessing
    - `encode_preprocess(file_path, output_path)`
      → **file:** `utils.py`
    - Cleans numeric columns (no binning), encodes categoricals, encodes target.

2- Train/Test Split (SHARED FOR EVERYONE)

All students must use the **same split** logic.

- **What to use**
  - `split_dataset(input_csv, output_dir, test_size=0.2, seed=42, ...)`
  - **File:** `split_dataset.py`
- **How it's invoked in the pipeline**

  The function is called inside:

  - `TrainTestSplitPipeline.run(...)`
    - **File:** `katabatic.pipeline.train_test_split.pipeline`
      (you saw this earlier as `TrainTestSplitPipeline`)
- **What it does**
  - Loads the **preprocessed** CSV (`input_csv`).
  - Runs a **stratified** train/test split on the **last column** (the label).
  - Saves:
    - `train_full.csv` , `test_full.csv`
    - `x_train.csv` , `y_train.csv`
    - `x_test.csv` , `y_test.csv`
  - All in `output_dir` .

You must not change this function or splitting logic.

3- Your Generative Model (STUDENT CODE)

Generative model example (CoDi)

- `class CODI(Model)`
  → **file:** `codi.py`
  - `train(dataset_dir, synthetic_dir, **kwargs)` :
    - loads `x_train.csv` (+ optional `y_train.csv`) from `dataset_dir` ,
    - infers schema, encodes data (using **CoDi-specific** helpers),
    - builds diffusion models,
    - trains them,
    - generates synthetic samples,
    - saves `x_synth.csv` **and** `y_synth.csv` into `synthetic_dir` ,
    - saves `schema.json` and `metadata.json` .
- Low-level utilities for CoDi
  → **file:** `codiutils.py`
  - `infer_schema` , `encode_dataframe` , `decode_dataframe` , `TabularUNet` , `GaussianDiffusionTrainer` , `GaussianDiffusionSampler` , `MultinomialDiffusion` , etc.

- **Where to look for an example**
  - `class CODI(Model)`
    - **File:** `codi.py`
- **What your `train(...)` MUST do**

  Given:

  - `dataset_dir` containing:
    - `x_train.csv`
    - `y_train.csv` (optional but recommended)
  - `synthetic_dir` = directory to write your synthetic data

  Your `train(...)` must:

  1. Load the training data:

     ```python
     x_train_path = os.path.join(dataset_dir, "x_train.csv")
     y_train_path = os.path.join(dataset_dir, "y_train.csv")  # if used
     ```
  2. Fit your generative model on this training data.
  3. Generate **synthetic samples** with the **same feature structure**.
  4. Save to **these exact file names**:
     - `synthetic_dir/x_synth.csv`
     - `synthetic_dir/y_synth.csv`

     (column order and label name must be consistent with the original.)
- **Optional helpers you can reuse (from CoDi)**

  From `codiutils.py`:

  - `infer_schema(df, categorical_threshold=20)`
  - `encode_dataframe(df, schema)`
  - `decode_dataframe(df_encoded, schema)`
  - `TabularUNet`, `GaussianDiffusionTrainer`, `GaussianDiffusionSampler`, `MultinomialDiffusion`, `get_device`, `set_global_seed`

You can completely ignore CoDi internals and implement a different generative model (e.g., CTG, Gaussian copula, simple noise model) **as long as** you:

- obey the `Model` interface,
- read from `x_train.csv` / `y_train.csv`,
- write `x_synth.csv` / `y_synth.csv` in the same style.

4- Shared Pipeline to Run Everything

TSTR evaluation

- `class TSTREvaluation(Evaluation)`
  → file: `evaluation.py`
  - `load_data(synthetic_dir, real_test_dir)` loads:
    - synthetic: `x_synth.csv`, `y_synth.csv`
    - real: `x_test.csv`, `y_test.csv`
  - `evaluate()`:
    - Trains **four classifiers** on synthetic data:
      - LogisticRegression
      - MLPClassifier
      - RandomForestClassifier
      - XGBClassifier
    - Evaluates them on **real test data.**
    - Metrics:
      - `Accuracy`
      - `F1 Score` (weighted)
      - `AUC` (if binary labels)
    - Saves results to:
      - `Results/{dataset_name}/{model_name}_tstr.csv`

All the infrastructure needed for students to plug in their own generative models and still share:

- the same **preprocessing** (if they all call the same `utils.py` functions),
- the same **train/test split** (`split_dataset.py`),
- the same **evaluation pipeline & metrics** (`TSTREvaluation`),
- the same **overall pipeline orchestration** (`TrainTestSplitPipeline`).

constraints / interface

Students must implement a class like:

- `class MyModel(Model):`
  - Must have:

    `def train(self, dataset_dir: str, synthetic_dir: str, **kwargs) -> "MyModel":`
  - Inside `train`, they **must**:
    - read `x_train.csv` (+ `y_train.csv` if needed) from `dataset_dir`,
    - train their generative model,
    - generate synthetic data,
    - write to:
      - `synthetic_dir/x_synth.csv`
      - `synthetic_dir/y_synth.csv`

      (same format as CoDi)

If they follow **exactly this interface**, they can be dropped into:

```python
pipeline = TrainTestSplitPipeline(model=lambda: MyModel(...))
pipeline.run(
    input_csv=preprocessed_csv,
    output_dir=output_dir,
    synthetic_dir=synthetic_dir,
    real_test_dir=output_dir,
)
```

2. **Where discretization happens**

   Right now, discretization is done **outside** the pipeline (e.g., in the notebook):

```python
from utils import discretize_preprocess

dataset_path = ROOT / "raw_data" / "car.csv"
output_path = ROOT / "discretized_data" / "car.csv"
discretize_preprocess(str(dataset_path), str(output_path))
```

Note: Before running the pipeline, preprocess raw data using discretize_preprocess from utils.py

Extended pipeline overview (with pointers to code)

Here's a checklist, showing each required step and exactly which function/file to use.

Step 1 — Discretize raw data

Requirement: "Discretizes data"

- Use:
  discretize_preprocess(file_path, output_path, bins=10, strategy='uniform')
- Found in:
  utils.py
- What it does:
  - Loads raw CSV.
  - Cleans missing values.
  - Discretizes numeric columns into integer bins using KBinsDiscretizer.
  - Label-encodes categoricals and target.
  - Saves a fully discrete CSV where:
    - All feature columns are integers.
    - The last column is the label.

Students should not write their own discretizer — they should call this one.

Step 2 — Train/test split (shared for all models)

Requirement: "same train/test split"

- Use (indirect):
  split_dataset(input_csv, output_dir, test_size=0.2, seed=42, ...)
- Found in:
  split_dataset.py
- Usually invoked by:
  TrainTestSplitPipeline.run(...)
  → file: (your earlier TrainTestSplitPipeline file)
- What it does:
  - Loads the preprocessed (discrete) CSV.
  - Performs a single stratified split (default 80/20).

- Saves:
  - train_full.csv, test_full.csv
  - x_train.csv, y_train.csv
  - x_test.csv, y_test.csv

Every model sees the same train/test indices if they all use the same input_csv and seed.

Step 3 — Student generative model

Requirement: "Each student implements a generative synthetic data model"

- Base class:
  Model
  → file: katabatic.models.base_model (imported in codi.py)
- Example implementation to follow:
  class CODI(Model)
  → file: codi.py
- Required behavior for student models:
  - Implement train(self, dataset_dir: str, synthetic_dir: str, **kwargs) that:
    1. Reads training data from:
       - dataset_dir/x_train.csv
       - Optionally dataset_dir/y_train.csv
    2. Trains their generative model.
    3. Generates synthetic data with the same column structure.
    4. Saves:
       - synthetic_dir/x_synth.csv
       - synthetic_dir/y_synth.csv
         (label column name should match the original label name)

If students want to use a CoDi-like encoding/decoding for mixed-type data, they can reuse:

- infer_schema, encode_dataframe, decode_dataframe
  → file: codiutils.py

But that's optional; the assignment might allow any model as long as it writes x_synth/y_synth in the same format.

Step 4 — Use the shared pipeline

Requirement: “all models should use the same pipeline”

- Use:
  TrainTestSplitPipeline(model=...)
  → file: your TrainTestSplitPipeline file (shown earlier)
- Key behavior inside run(...):
  1. Calls split_dataset(input_csv, output_dir, ...)
     → file: split_dataset.py
  2. Instantiates model:
     current_model = self.model()
  3. Trains the generative model and generates synthetic data:
     current_model.train(output_dir, synthetic_dir=synthetic_dir, ...)
     - dataset_dir argument here is output_dir (where x_train/y_train live)
     - synthetic_dir is where the model must write x_synth.csv, y_synth.csv.
  4. Runs all evaluations in _evaluations, which includes TSTREvaluation.

Step 5 — TSTR evaluation (same for all models)

Requirement: “same evaluation pipeline / same metrics”

- **Use:**
  TSTREvaluation(synthetic_dir, real_test_dir)
  → **file:** evaluation.py
  (automatically used inside TrainTestSplitPipeline)
- **It uses:**
  - x_synth.csv, y_synth.csv from synthetic_dir
  - x_test.csv, y_test.csv from real_test_dir
- **Models trained:**
  - Logistic Regression (LR)
  - MLPClassifier (MLP)
  - RandomForest (RF)
  - XGBoost (XGBoost)
- **Metrics computed:**
  - Accuracy

- F1 Score (weighted)
- AUC (if binary classification)

- **Saved outputs:**
    - CSV file:
      Results/{dataset_name}/{model_name}_tstr.csv
      created by save_results_to_csv(results, synthetic_dir) in evaluation.py.

This ensures **every student** is judged by exactly the same TSTR protocol.

Here’s the full pipline map:

- **Discretizes data**
  → discretize_preprocess(file_path, output_path, bins, strategy)
  file: utils.py
- **Splits into train/test (same for everyone)**
  → split_dataset(input_csv, output_dir, test_size=0.2, seed=42)
  file: split_dataset.py
  (called from TrainTestSplitPipeline.run)
- **Defines model interface all students must follow**
  → class Model (base)
  file: katabatic.models.base_model
  → Example: class CODI(Model)
  file: codi.py
- **Implements one generative model (CoDi example)**
  → class CODI(Model) with train(...), sample(...)
  file: codi.py
- **Low-level building blocks for CoDi (optional reuse)**
  → infer_schema, encode_dataframe, decode_dataframe, TabularUNet, GaussianDiffusionTrainer, GaussianDiffusionSampler, MultinomialDiffusion, get_device, set_global_seed
  file: codiutils.py
- **Common training + evaluation pipeline**
  → class TrainTestSplitPipeline(Pipeline)
  file: your pipeline file (imports split_dataset, TSTREvaluation)

- **TSTR evaluation with fixed models & metrics**
  → class TSTREvaluation(Evaluation)
  file: evaluation.py
- **Saving all outputs & metrics to disk**
  - Preprocessed data:
    - discretize_preprocess → saves preprocessed CSV
    - **file:** utils.py
  - Train/test splits:
    - split_dataset → saves train_full.csv, test_full.csv, x_*, y_*
    - **file:** split_dataset.py
  - Synthetic data:
    - CODI.train (and student models) → saves x_synth.csv, y_synth.csv
    - **file:** codi.py or their own model file.
  - Evaluation metrics:
    - TSTREvaluation.save_results_to_csv → saves {model_name}_tstr.csv
    - **file:** evaluation.py