National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Moyosoreoluwa Monsurat Ayomide Odufuwa
Student ID: x2338491

School of Computing
National College of Ireland

Supervisor: Abid Yaqoob

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: Moyosoreoluwa Monsurat Ayomide Odufuwa

Student ID: X23338491

Programme: MSc Data Analytics **Year:** 2024/2025

Module: Research Practicum

Lecturer: Abid Yaqoob

Submission Due Date: 11/08/2025

Project Title: Promoting Financial Inclusion: Evaluating CTGAN-Based Synthetic Alternative data for the Credit Invisible

Word Count: 1326 **Page Count:** 17

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Moyosoreoluwa Monsurat Ayomide Odufuwa

Date: 10/08/2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Moyosoreoluwa Monsurat Ayomide Odufuwa
Student ID: x23338491

Introduction

This manual provides thorough step-by-step instruction for setting up and carrying out the implementation of this project.

1 Environment

This section contains the all-round environmental setup of the project and the system requirements. The code implementation is executed on Google Colab using the python programming language.

1.1 System Configuration

Software Configuration

- **Programming Language**: Python Version 3
- **Tools**: Google Colab
- **Google Drive**: Access to Google Drive for data access
- **Operating System**: Windows 11

Hardware Configuration

- **Processor**: AMD Ryzen 3 3250U with Radeon Graphics 2.60 GHz
- **RAM**: 4GB (a higher RAM is advised)
- **Computational Resources**: Google Colab's L4 GPU 22.5GB RAM

1.2 Google Colab Setup

The datasets and the python file containing the code implementation are stored in Google drive. This allows for easy access in Google Colab.

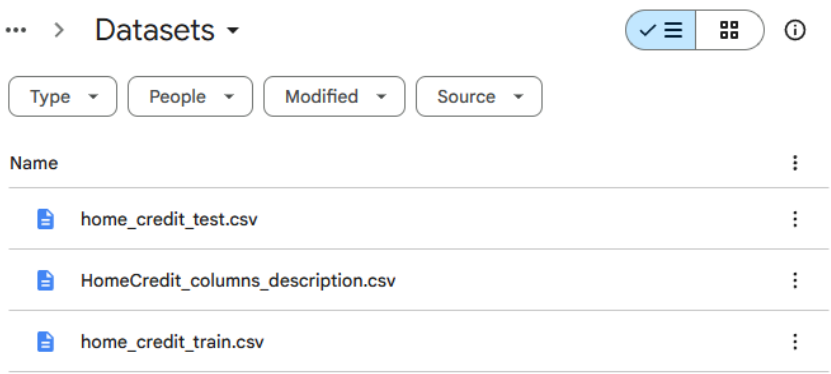

Fig 1: Datasets stored in Google Drive

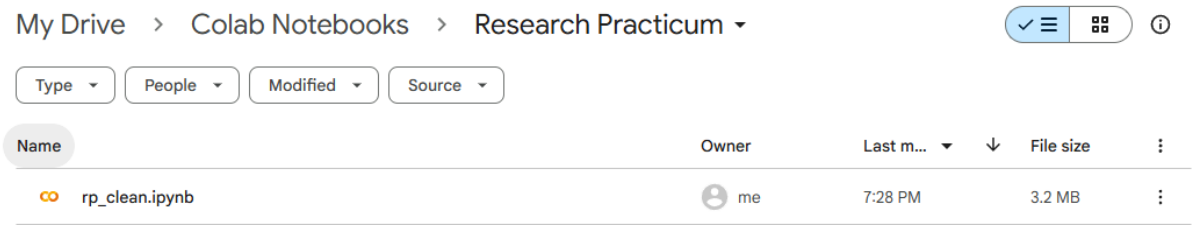

Fig 2: Code Implementation notebook stored in Google Drive

The mount function, connects Google Drive to the Colab environment. This allows the notebook in the Colab environment to access files stored in Google Drive, making it possible to load datasets as well as save results directly to ones Drive.

Fig 3: Google Colab Mount

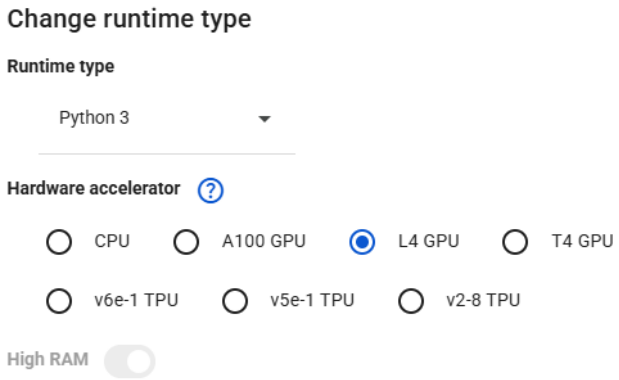

Fig 4: Runtime Configuration

In figure 4, the hardware accelator resources available by Colab are highlighetd with L4 GPU being selected for being to handle powereful computational tasks.

1.3 Datasets

The dataset was sourced from Kaggle, and the training set of the application table is used to carry out this project. This consists of 307,511 rows and 122 columns

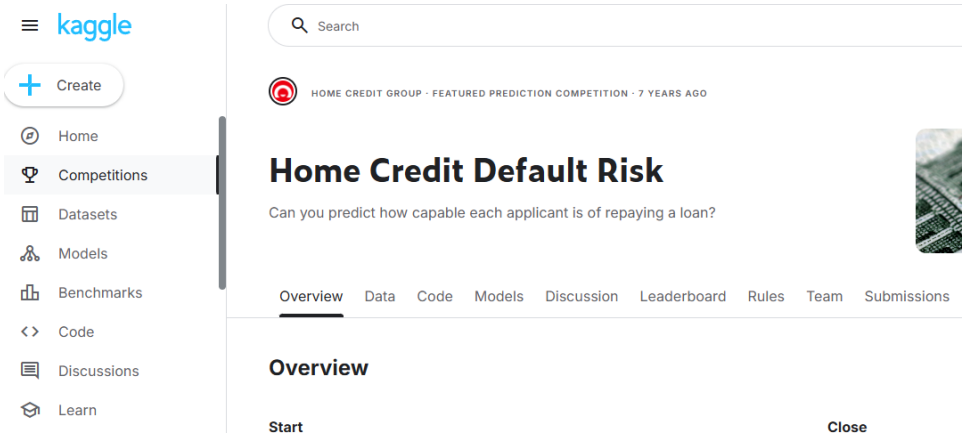

Fig 5: Data Source

1.4 Necessary Libraries

The python libraries required to bring this project to a completion are imported into the Colab environment.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OrdinalEncoder
from sklearn.feature_selection import SelectKBest, mutual_info_classif
from sklearn.impute import SimpleImputer
from ctgan import CTGAN
from table_evaluator import load_data, TableEvaluator
from sklearn.compose import ColumnTransformer, make_column_selector as selector
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import StratifiedKFold, RandomizedSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    roc_auc_score, average_precision_score, log_loss, brier_score_loss
)
from scipy.stats import ks_2samp
from scipy.stats import spearmanr
```

Fig 6: Imported Libraries

2 Data Preparation

2.1 Exploratory Data Analysis (EDA)

EDA is carried out before any data altering steps are performed to provide a clear distribution of the original data.

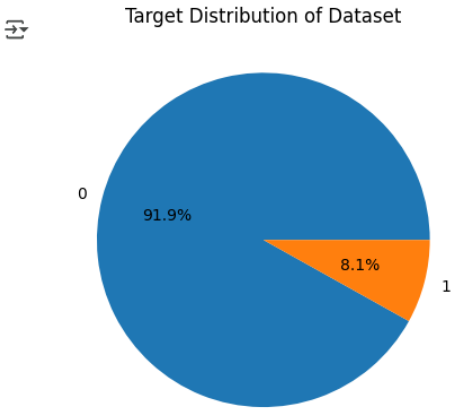

Fig 7: Target Distribution

2.2 Data Preprocessing

First a copy of the original dataset is created, and its size is reduced to 150, 000 rows to preserve computational resources. This new dataset is split into train/test sets using a 70/30 ratio. Using this train set, all preprocessing steps are carried out.

Feature Selection

Feature Selection is achieved using SelectKBest selection method. To maintain the data and avoid mishaps, a copy of the train set is passed to SelectKBest method which utilizes the mutual_info_classif score function to rate each feature based on mutual information with the target. Based on the results, features (columns) were reduced from 122 to 46.

iv: Apply SelectKBest for Feature Selection

```
[ ]  print(f"x_kbest_train shape: {x_kbest.shape}")
     print(f"y_train shape: {y_train.shape}")
```

```
x_kbest_train shape: (105000, 120)
y_train shape: (105000,)
```

```
#Feature selection using SelectKBest
from sklearn.feature_selection import SelectKBest, mutual_info_classif

# Calculate the mutual information scores for each feature
k = len(x_kbest.columns)
selector = SelectKBest(score_func=mutual_info_classif, k=k)
selector.fit(x_kbest, y_kbest)
```

Fig 8: Feature selection using SelectKBest

Create train/test splits with selected features. This will be our original data

```
x_base_train = x_train[selected_features]  # Selected features from training data
x_base_test = x_test[selected_features]    # Same features from test data

y_base_train = y_train.copy()  # Original training labels
y_base_test = y_test.copy()

# 3. Verify shapes
print("\nShape of train/test sets after feature selection:")
print(f"Train features: {x_base_train.shape}")
print(f"Test features:  {x_base_test.shape}")
print(f"Train target:   {y_base_train.shape}")
print(f"Test target:    {y_base_test.shape}")
```

```
Shapes after feature selection:
Train features: (105000, 47)
Test features:  (45000, 47)
Train target:   (105000,)
Test target:    (45000,)
```

Fig 9: Shape of original data train/test set on 70/30 ratio split after reducing feature size

CTGAN Preprocessing

Before passing the original training and testing data to the CTGAN model, some preprocessin steps have to be made to ensure the data is in complaince with the model requirements.

i. Discrete (categorical) data must be represented as strings or (encoded as) integers

```
#Store 'TARGET' as an integer
data_ctgan['TARGET'] = (
    data_ctgan['TARGET']
      .astype('int8')  #make sure datatype is integer
)

# Build the list of discrete columns
cat_cols_cleaned = data_ctgan.select_dtypes(include=['object', 'category']).columns # Select columns with object or category dtype
flag_cols = [c for c in data_ctgan.columns if data_ctgan[c].dropna().isin([0.0, 1.0]).all()] #boolean feature columns

#change boolean feature datatype to int
data_ctgan[flag_cols] = data_ctgan[flag_cols].astype('int8')

#create a dataframe of discreate features
discrete_cols = list(cat_cols_cleaned) + flag_cols
print(data_ctgan[discrete_cols].dtypes)
```

Fig 10: step i. preprocessing for CTGAN

ii. Continuous data must be represented as floats

```
#Select continuous (numeric) columns
cont_cols = (data_ctgan
             .select_dtypes(include='number')
             .columns
             .difference(discrete_cols))

#convert numeric features to float values
data_ctgan[cont_cols] = (
    data_ctgan[cont_cols].astype('float32')   # or 'float64'
)

print("Continuous features converted to float.")
print("Data types after conversion: \n", data_ctgan[cont_cols].dtypes)
```

Fig 11. Step ii. preprocessing for CTGAN

iii. The data should not contain any missing values

```
[38] data_ctgan.isnull().sum().sum()
```

```
0
```

Fig 12. Step iii. The data must not contain any missing values.

3 CTGAN Data Generation

After preprocessing the model is trained on the original dataset based on the following hyperparameters.

```
#Define a list with column names for categorical variables.
categorical_features = discrete_cols
print("Categorical features: ", categorical_features)

# Instantiate the CTGAN using optimized hyperparameters
ctgan = CTGAN(
    epochs=150, #Number of times to train the GAN. Each new epoch can improve the model.
    discriminator_dim=(256,256), #Size of the output samples for each one of the Discriminator Layers.
    generator_dim=(256,256), #Size of the output samples for each one of the Residuals
    discriminator_lr = 0.0002, #Learning rate for the Discriminator.
    generator_lr = 0.0002, #Learning rate for the Generator.
    verbose=True) #print out the results of each epoch
    #batch_size=256, #Number of data samples to process in each step.

# Fit the CTGAN model to original train data
ctgan.fit(data_ctgan, categorical_features)

print("\n CTGAN model trained.")
```

Fig 13: Training the CTGAN model

Generate Synthetic data using trained CTGAN model

```
# number of samples as the original dataset
n_rows = len(data_ctgan)

# Generate synthetic data the same size as the original dataset
synthetic_data = ctgan.sample(n_rows)

print("Synthetic data generated.")
print("\nShape of synthetic data:", synthetic_data.shape)


# Verify synthetic data balance
print("\nOriginal Distribution:\n", y_train.value_counts(normalize=True))
print("\nSynthetic Distribution:\n", synthetic_data['TARGET'].value_counts(normalize=True))

display(synthetic_data.head())
```

Fig 14: Synthetic data generation using the trained CTGAN model

Synthetic Data Evaluation

With the help of the table_evaluator library, the synthetic data was evaluated on how well it matches the original following a feature by feature comparison

Evaluate synthetic data: A feature by feature comparision between the generated data and the actual data. We will use python's table_evaluator library to compare the features.

```
# Load the original data and the synthetic data into the TableEvaluator for evaluation
table_evaluator =  TableEvaluator(data_ctgan, synthetic_data, categorical_features)
table_evaluator.visual_evaluation()
```

Fig 15: Feature by Feature Comparison

Passing the target column and the target type (classification) the evaluate method from table_evaluate library measures the quality and similarity of the synthetic data compared to the real data

```
table_evaluator.evaluate(
    target_col='TARGET',
    target_type='class')
```

Fig 16: Synthetic and original dataset evaluation

4 Data Modelling

Synthetic Data Train/Test Split

Synthetic Data: Train/Test Split

```
# Create train/test split for the synthetic data
x_syn = synthetic_data.drop('TARGET', axis=1).copy()
y_syn = synthetic_data['TARGET'].copy()

x_syn_train, x_syn_test, y_syn_train, y_syn_test = train_test_split(x_syn, y_syn, test_size=0.3, stratify=y_syn, random_state=42)

print("Synthetic shape:", x_syn.shape)
```

Fig 17: 70/30 ratio train/test split on synthetic dataset

Combined Data Train/Test Split

Combined Data = Original train + Synthetic data: Train/Test Split

```
# Concatenate the original training data with the synthetic training data to create an augmented dataset.
x_aug = pd.concat([x_base_train, x_syn_train], ignore_index=True)
y_aug = pd.concat([y_base_train, y_syn_train], ignore_index=True)
```

Fig 18: 70/30 ratio train/test split on combined dataset

Model Classification

Predictive ability of all 3 datasets

```
#identy numerical and categorical selectors
num_sel = selector(dtype_include=np.number)
cat_sel = selector(dtype_exclude=np.number)

# One-hot for categoricals
ohe = OneHotEncoder(handle_unknown="ignore")

preprocess_common = ColumnTransformer(
    transformers=[
        ("num", "passthrough", num_sel),
        ("cat", ohe, cat_sel),
    ],
    remainder="drop"
)

# For Logistic Regression: scale numeric columns
preprocess_lr = ColumnTransformer(
    transformers=[
        ("num", Pipeline([("scaler", StandardScaler())]), num_sel),
        ("cat", ohe, cat_sel),
    ],
    remainder="drop"
)
```

```
#Pipeline for each classifier
pipe_lr  = Pipeline([("prep", preprocess_lr),  ("clf", LogisticRegression(max_iter=1000, random_state=42))])
pipe_rf  = Pipeline([("prep", preprocess_common), ("clf", RandomForestClassifier(random_state=42))])
pipe_xgb = Pipeline([("prep", preprocess_common), ("clf", XGBClassifier(
    eval_metric="logloss", use_label_encoder=False, random_state=42, n_jobs=-1
))])

# LR parameters
param_lr = {
    "clf__C": np.logspace(-3, 3, 7),
    "clf__penalty": ["l2"],
    "clf__solver": ["lbfgs", "liblinear"],
    "clf__class_weight": ["balanced"]
}

# Random Forest parameters
param_rf = {
    "clf__n_estimators": [200, 300, 500],
    "clf__max_depth": [None, 10, 20],
    "clf__min_samples_split": [2, 5, 10],
    "clf__min_samples_leaf": [1, 2, 4],
    "clf__class_weight": ["balanced"]
}
```

```
#XGBoost Parameters
base_xgb = {
    "clf__n_estimators": [300, 500],
    "clf__max_depth": [3, 5, 7],
    "clf__learning_rate": [0.03, 0.1],
    "clf__subsample": [0.8, 1.0],
    "clf__colsample_bytree": [0.8, 1.0],
    #'scale_pos_weight': [(y_base_train==0).sum()/(y_base_train==1).sum()]
}


cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)
scoring = "roc_auc"
n_iter = 5

datasets = {
    "Baseline": (x_base_train, y_base_train),
    "Combined": (x_aug, y_aug),
    "Synthetic": (x_syn_train, y_syn_train)
}

# Store results here
results_rows = []
best_models = {}
```

```
for name, (Xtr, ytr) in datasets.items():
    print(f"\n {name} dataset")

    # Logistic Regression modelling
    lr_search = RandomizedSearchCV(pipe_lr, param_lr, n_iter=n_iter, cv=cv, scoring=scoring,
                                   n_jobs=-1, verbose=1, random_state=42)
    lr_search.fit(Xtr, ytr)
    best_models[f"LR_{name}"] = lr_search.best_estimator_
    results_rows.append({
        "Dataset": name,
        "Model": "Logistic Regression",
        "CV ROC-AUC": lr_search.best_score_
    })

    # Random Forest modelling
    rf_search = RandomizedSearchCV(pipe_rf, param_rf, n_iter=n_iter, cv=cv, scoring=scoring,
                                   n_jobs=-1, verbose=0, random_state=42)
    rf_search.fit(Xtr, ytr)
    best_models[f"RF_{name}"] = rf_search.best_estimator_
    results_rows.append({
        "Dataset": name,
        "Model": "Random Forest",
        "CV ROC-AUC": rf_search.best_score_
    })
```

```
    # XGBoost (set imbalance weight per dataset)
    pos_ratio = (ytr == 0).sum() / max(1, (ytr == 1).sum())
    param_xgb = {**base_xgb, "clf__scale_pos_weight": [pos_ratio]}
    xgb_search = RandomizedSearchCV(pipe_xgb, param_xgb, n_iter=n_iter, cv=cv, scoring=scoring,
                                    n_jobs=-1, verbose=1, random_state=42)

    xgb_search.fit(Xtr, ytr)
    best_models[f"XGB_{name}"] = xgb_search.best_estimator_
    results_rows.append({
        "Dataset": name,
        "Model": "XGBoost",
        "CV ROC-AUC": xgb_search.best_score_
    })

# Convert to DataFrame
results_df = pd.DataFrame(results_rows)

# Format result
results_df = results_df.sort_values(["Dataset", "Model"]).reset_index(drop=True)
results_df["CV ROC-AUC"] = results_df["CV ROC-AUC"].round(4)

results_df
```

Fig 19: Classifier Modelling

In fig 19, various classification models (Logistic Regression, Random Forest, and XGBoost) are trained on three different datasets: the original baseline data, the combined original and synthetic data, and the synthetic data alone. It then evaluates their performance using cross-validation.

5 EVALUATION

Kolmogorov-Smirnov (KS) Tests & Pearson and Spearman Correlation Similarity

Two statistical tests are performed to compare the distributions and relationships within the real and synthetic numerical features. This helps assess how well the synthetic data captures the statistical properties of the original data.

These quantitative measures (KS statistics and correlation similarity) and a visual representation (KS bar plot) help understand if the synthetic data's numerical features have similar distributions and inter-feature relationships as the original data.

```
# Confirm we have same columns between real and synthetic
real_df = x_base_train.copy()
synthetic_df = x_syn_train.copy()

# Identify numeric columns
num_cols = real_df.select_dtypes(include=np.number).columns

# Kolmogorov-Smirnov (KS) statistic
ks_results = []
for col in num_cols:
    ks_stat, ks_p = ks_2samp(real_df[col], synthetic_df[col])
    ks_results.append({"Feature": col, "KS_stat": ks_stat, "p_value": ks_p})
ks_df = pd.DataFrame(ks_results).sort_values("KS_stat").reset_index(drop=True)

print("Kolmogorov-Smirnov Test Results (P-value < 0.05 = poor match):")
display(ks_df)
```

```python
# Plot KS statistics
colors = ["red" if val > 0.1 else "skyblue" for val in ks_df["KS_stat"]]

plt.figure(figsize=(10, 6))
plt.barh(ks_df["Feature"], ks_df["KS_stat"], color="green")
plt.axvline(0.1, color="grey", linestyle="--", label="Threshold = 0.1")
plt.xlabel("KS Statistic")
plt.ylabel("Feature")
plt.title("Kolmogorov-Smirnov Statistic per Feature (Real vs Synthetic)")
plt.gca().invert_yaxis()
plt.grid(axis="x", linestyle="--", alpha=0.7)
plt.legend()
plt.show()

# Pairwise correlation similarity
def corr_similarity(df1, df2, method="pearson"):
    corr1 = df1.corr(method=method)
    corr2 = df2.corr(method=method)
    # Flatten upper triangle
    mask = np.triu(np.ones(corr1.shape), k=1).astype(bool)
    vals1 = corr1.where(mask).stack().values
    vals2 = corr2.where(mask).stack().values
    return np.corrcoef(vals1, vals2)[0, 1]
```

```python
pearson_sim = corr_similarity(real_df[num_cols], synthetic_df[num_cols], method="pearson")
spearman_sim = corr_similarity(real_df[num_cols], synthetic_df[num_cols], method="spearman")

print(f"Pearson correlation similarity: {pearson_sim:.4f}")
print(f"Spearman correlation similarity: {spearman_sim:.4f}")
```

Fig 20: Kolmogorov-Smirnov (KS) Tests & Pearson and Spearman Correlation Similarity

Discrimination & calibration evaluation

The LogLoss and Brier score are calculated on all datasets against the original test set. These metrics assess the quality of the model's predicted probabilities and not just the final classification outcome.

```python
# Evaluate tuned models on original test set

def get_proba(m, X):
    "Return positive-class probabilities for any sklearn/xgb pipeline/estimator."
    if hasattr(m, "predict_proba"):
        return m.predict_proba(X)[:, 1]
    if hasattr(m, "decision_function"):
        s = m.decision_function(X).ravel()
        return 1 / (1 + np.exp(-s))
    # fallback (not ideal)
    return m.predict(X).astype(float)
```

```python
def evaluate_on_test(models: dict, X_test, y_test, threshold=0.5) -> pd.DataFrame:
    rows = []
    for name, mdl in models.items():
        p = get_proba(mdl, X_test)
        yhat = (p >= threshold).astype(int)
        rows.append({
            "Dataset": name.split("_", 1)[1] if "_" in name else "",
            "Model": name.split("_", 1)[0] if "_" in name else name,
            "Accuracy": accuracy_score(y_test, yhat),
            "Precision": precision_score(y_test, yhat, zero_division=0),
            "Recall": recall_score(y_test, yhat, zero_division=0),
            "F1": f1_score(y_test, yhat, zero_division=0),
            "ROC-AUC": roc_auc_score(y_test, p),
            "PR-AUC": average_precision_score(y_test, p),
            "LogLoss": log_loss(y_test, np.c_[1-p, p], labels=[0,1]),
            "Brier": brier_score_loss(y_test, p),
        })
    df = pd.DataFrame(rows)
    cols_order = ["Dataset","Model","Accuracy","Precision","Recall","F1","ROC-AUC","PR-AUC","LogLoss","Brier"]
    df = df[cols_order].sort_values(["Dataset","Model"]).reset_index(drop=True)
    return df.round(4)

test_results_df = evaluate_on_test(best_models, x_base_test, y_base_test, threshold=0.5)
test_results_df
```

Fig 21: Calibration and calibration evaluation

Fairnes Evaluation

Models trained all three datasets are analyzed on whether bias exists with respect to the defined protected attributes by quantifying differences in predictive performance across subgroups. Lower values for SPD, EOD, and AOD generally indicate better fairness.

```python
# Fairness on all datasets against the original test set for protected attributes
protected_cols = ["NAME_INCOME_TYPE", "NAME_EDUCATION_TYPE", "NAME_HOUSING_TYPE"]
present_cols = [c for c in protected_cols if c in data_cleaned.columns]
if len(present_cols) == 0:
    raise ValueError("Protected columns not found in data_cleaned. Check your dataframe.")

# Align protected attributes to the test rows
prot_df = data_cleaned.loc[x_base_test.index, present_cols].copy()
```

```python
def per_group_metrics(y_true, y_proba, groups: pd.Series, threshold=0.5):
    df = pd.DataFrame({"y": np.asarray(y_true), "p": np.asarray(y_proba), "g": groups})
    df = df.dropna(subset=["g"]).copy()
    df["yhat"] = (df["p"] >= threshold).astype(int)

    rows = []
    for g, sub in df.groupby("g"):
        neg = (sub["y"]==0).sum()
        pos = (sub["y"]==1).sum()
        row = {"group": g, "support": len(sub)}
        if sub["y"].nunique() > 1:
            row["roc_auc"] = roc_auc_score(sub["y"], sub["p"])
        else:
            row["roc_auc"] = np.nan
        row["fpr"] = ((sub["yhat"]==1) & (sub["y"]==0)).sum() / neg if neg>0 else np.nan
        row["fnr"] = ((sub["yhat"]==0) & (sub["y"]==1)).sum() / pos if pos>0 else np.nan
        row["tpr"] = ((sub["yhat"]==1) & (sub["y"]==1)).sum() / pos if pos>0 else np.nan
        row["precision"] = precision_score(sub["y"], sub["yhat"], zero_division=0)
        row["recall"] = recall_score(sub["y"], sub["yhat"], zero_division=0)
        rows.append(row)
    grp_df = pd.DataFrame(rows).sort_values("support", ascending=False).reset_index(drop=True)
```

```
    # Summary deltas
    sp_rates = df.groupby("g")["yhat"].mean()         # predicted positive rates
    spd = sp_rates.max() - sp_rates.min()             # Statistical Parity Difference
    tpr_by_g = grp_df.set_index("group")["tpr"]
    eod = (tpr_by_g.max() - tpr_by_g.min()) if tpr_by_g.notna().any() else np.nan   # Equal Opportunity Diff
    fpr_by_g = grp_df.set_index("group")["fpr"]
    aod = 0.5 * ((fpr_by_g.max() - fpr_by_g.min()) + (tpr_by_g.max() - tpr_by_g.min()))  # Average Odds Diff
    return grp_df, {"SPD": spd, "EOD": eod, "AOD": aod}

# Build fairness summary across models & attributes
fair_rows = []
per_group_tables = {}  # dict[(model_key, attribute)] = per-group DataFrame

for model_key, mdl in best_models.items():
    p = mdl.predict_proba(x_base_test)[:,1] if hasattr(mdl, "predict_proba") else get_proba(mdl, x_base_test)
    for attr in present_cols:
        grp_tbl, summ = per_group_metrics(y_base_test, p, prot_df[attr], threshold=0.5)
        per_group_tables[(model_key, attr)] = grp_tbl
        fair_rows.append({
            "Model Key": model_key,
            "Dataset": model_key.split("_",1)[1] if "_" in model_key else "",
            "Model": model_key.split("_",1)[0] if "_" in model_key else model_key,
            "Attribute": attr,
            **summ
        })
```

```
fairness_summary_df = pd.DataFrame(fair_rows).sort_values(["Attribute","Dataset","Model"]).reset_index(drop=True)
fairness_summary_df.round(4)

# Pivot table transformation
df_pivot = fairness_summary_df.drop(columns=["Model Key"])
df_pivot = df_pivot.pivot_table(
    index=["Attribute", "Model"],
    columns="Dataset",
    values=["SPD", "EOD", "AOD"]
)
df_pivot.columns = [f"{dataset} {metric}" for metric, dataset in df_pivot.columns]
df_pivot = df_pivot.reset_index()
df_pivot = df_pivot.sort_values(["Attribute", "Model"]).reset_index(drop=True)
df_pivot = df_pivot.round(4)

df_pivot
```

Fig 22: Fairness evaluation against the three datasets

Spearman Rank Correlation

Similarity of feature importances derived from an XGBoost model trained on the original real data and another XGBoost model trained on the synthetic data is evaluated using the spearman rank correlation. A high Spearman rank correlation (closer to 1) suggests that the features are ranked similarly in importance by both models.

```
# Identify numerical features common in both datasets
cols = x_base_train.select_dtypes(include=np.number).columns.intersection(x_syn_train.columns)

# Initializes two instances of the XGBoost classifier with the same hyperparameters.
# (xgb_r) will be trained on the real data, and the (xgb_s) on the synthetic data
xgb_r = XGBClassifier(n_estimators=300, max_depth=5, learning_rate=0.1, subsample=0.8, colsample_bytree=0.8, random_state=42)
xgb_s = XGBClassifier(n_estimators=300, max_depth=5, learning_rate=0.1, subsample=0.8, colsample_bytree=0.8, random_state=42)

xgb_r.fit(x_base_train[cols], y_base_train)
xgb_s.fit(x_syn_train[cols], y_syn_train)

# Extract feature importance from each trained model.
imp_r = pd.Series(xgb_r.feature_importances_, index=cols)
imp_s = pd.Series(xgb_s.feature_importances_, index=cols)
fi_spearman = spearmanr(imp_r, imp_s).correlation
print("Feature-importance rank correlation (real vs synthetic):", round(fi_spearman, 4))
```

Fig 23: Spearman rank correlation

References

References should be formatted using APA or Harvard style as detailed in NCI Library Referencing Guide available at https://libguides.ncirl.ie/referencing
You can use a reference management system such as Zotero or Mendeley to cite in MS Word.

Beloglazov, A. and Buyya, R. (2015). Openstack neat: a framework for dynamic and energy-efficient consolidation of virtual machines in openstack clouds, *Concurrency and Computation: Practice and Experience* 27(5): 1310–1333.

Feng, G. and Buyya, R. (2016). Maximum revenue-oriented resource allocation in cloud, *IJGUC* 7(1): 12–21.

Gomes, D. G., Calheiros, R. N. and Tolosana-Calasanz, R. (2015). Introduction to the special issue on cloud computing: Recent developments and challenging issues, *Computers & Electrical Engineering* 42: 31–32.

Kune, R., Konugurthi, P., Agarwal, A., Rao, C. R. and Buyya, R. (2016). The anatomy of big data computing, *Softw., Pract. Exper.* 46(1): 79–105.