National College of Ireland

Automatic Test Data Generation in Banking Applications Using Deep Learning

MSc Research Project
Data Analytics

Taniya Bagh
Student ID: X22120831

School of Computing
National College of Ireland

Supervisor: Sasirekha Palaniswamy

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: Taniya Bagh……

Student ID: X22120831……

Programme: Data Analytics…… **Year:** …2023……..

Module: MSc Research Project……

Lecturer: Sasirekha Palaniswamy……

Submission Due Date: 14/12/2023……

Project Title: Automatic Test Data Generation in Banking Applications Using Deep Learning

Word Count: …… **Page Count:** ……

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: Taniya Bagh……

Date: 14/12/20……

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):	

Automatic Test Data Generation in Banking Applications Using Deep Learning

Taniya Bagh
Student ID: x22120831

1 Introduction

This module will contain all the documentation and information consisting of hardware, and software configuration along with the components(codes) needed for the implementation of the research paper titled automatic test data generation in banking applications using deep learning.
The brief code walk-through has been mentioned in the manual which was followed exactly for achieving the anticipated results.

2 Configuration of Hardware and Software

Figure 1 shows the technical configuration of the used device with the operating system of the windows upon which the research has been conducted.

Figure 1 Device Specification

Windows Specification is mentioned in Figure 2

Figure 2 Windows Specification

The programming language used for the implementation of this project is Python (Python 3) and the frameworks used for the application of Deep learning models are Tensorflow and Kreas(which is the high end API of Tensorflow). The detailed specification is mentioned in Figure 3:

IDE	Google Colab Pro
Programming Language	Python
Modules	Keras, TensorFlow, Matplotlib,Pandas, Numpy
Computation	CPU
Number of CPU	1
Type	Intel(R) Xeon(R) CPU @ 2.20GHz

Figure 3 Windows Specification

3 Dataset Utilized

The dataset selected to conduct the implementation of this project is taken from Kaggle website. The dataset goes by the name "bank_transactions.csv" consisting of 1M+ transactional data from an Indian bank with 9 fields. The dataset is available for public use. The dataset can be retrieved from the below link:
https://www.kaggle.com/datasets/shivamb/bank-customer-segmentation

The data is stored in the Google Colab Pro platform form where other executions of the codes were done.

4 Implementation of the Models

4.1 EDA, Data Preprocessing and Data Transformation.

Figure: 4 contains all the libraries that are required.

Figure 5 , 6 shows the EDA process where missing values are handled and various plots were plotted (Heatmap, Bar plot, Histogram plot etc)to understand the characteristics of the data.

```
import pandas as pd
import numpy as np
from scipy import stats
import tensorflow as tf
from tensorflow import keras
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder, MinMaxScaler
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.neural_network import MLPClassifier
from keras.layers import Dense,LeakyReLU, BatchNormalization, Reshape, Flatten, Input, SimpleRNN, Lambda, TimeDistributed
from keras.models import Sequential, load_model,Model
from keras.optimizers import Adam
from keras.applications.inception_v3 import InceptionV3
from keras.applications.inception_v3 import preprocess_input
from keras import backend as K
from scipy.stats import entropy
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, accuracy_score, f1_score, recall_score,mean_squared_error, mean_absolute_error, precision_score
from keras.models import Sequential
from keras.losses import binary_crossentropy
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from keras.preprocessing.sequence import pad_sequences
#from tensorflow.keras.constraints import
import seaborn as sns
from sklearn.mixture import GaussianMixture
from sklearn.impute import SimpleImputer
from sklearn.decomposition import PCA
```

Figure 4 Imported Libraries

EDA

```
[9]  # Handling missing values
     df.dropna(inplace=True)
```

Figure 5 handling missing terms

```
# Pairplot for numerical columns
sns.pairplot(df, diag_kind='kde')
plt.suptitle("Pairplot of Numerical Columns")
plt.show()

# Correlation heatmap
correlation_matrix = df.corr()
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f', linewidths=0.5)
plt.title("Correlation Heatmap")
plt.show()

#Distinct count of the values present in CustGender

distinct_count = df['CustGender'].nunique()
print("Number of distinct values in 'CustGender':", distinct_count)

distinct_values_counts = df['CustGender'].value_counts()

print("Distinct values and their counts in 'CustGender':")
print(distinct_values_counts)


# Bar plot for TransactionAmount by CustGender
filtered_df = df[df['CustGender'].isin(['M', 'F'])]
plt.figure(figsize=(10, 6))
sns.barplot(x='CustGender', y='TransactionAmount (INR)', data=filtered_df, ci=None)  # ci=None removes error bars
plt.title("Bar Plot of TransactionAmount by CustGender")
plt.show()
```

Figure 6 EDA

The next step is Data transformation, converting the categorical value to numerical using a Label encoder and standardized numerical data as shown in Figure 7. In Figure 8 the outliers are detected and removed. The dataset is split into train and test where custGender field is taken as the target value(Figure 9).

```
[12] # Convert categorical variables to numerical representation
     label_encoders = {}
     for column in df.select_dtypes(include='object').columns:
         le = LabelEncoder()
         df[column] = le.fit_transform(df[column])
         label_encoders[column] = le

     # Standardize numerical features
     scaler = StandardScaler()
     df[df.columns] = scaler.fit_transform(df[df.columns])
     print(df.head())
```

Figure 7 Data Transformation

```
# Outlier detection and handling for numerical features
numerical_features = df.select_dtypes(include=['float64', 'int64']).columns
Q1 = df[numerical_features].quantile(0.25)
Q3 = df[numerical_features].quantile(0.75)
IQR = Q3 - Q1
data = df[~((df[numerical_features] < (Q1 - 1.5 * IQR)) | (df[numerical_features] > (Q3 + 1.5 * IQR))).any(axis=1)]
```

Figure 8 Outlier Detection using Interquartile range.

```
# Split the dataset into features (X) and target variable (y)
X = df.drop("CustGender", axis=1)
y = df["CustGender"]

# Choose a threshold value based on your problem
threshold_value = 0.5

y_binary = (y > threshold_value).astype(int)  #changing continuous value to categorical value

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y_binary, test_size=0.2, random_state=42)
```

Figure 9 Splitting the data.

4.2 Implementation of GAN

The architecture of the Generative Adversarial Network model consists of a Generator and a discriminator. The Generator is fed some noise and consists of the dense layer which is again followed by LeakyRelu activation function which brings nonlinearity and assists in reducing the vanishing gradient problem and batch normalization is used to keep the model stable. The output layer of the generator model consists of a tanh activation function which keeps the data generated in a specific range. The discriminator model also consists of dense layers and leakyRelu activation function. The discrimator is fed the train data from real data which is then compared with the synthetic samples generated by the generator (Figure 10). The loss function used in GAN model is

shown in Figure 11. The model is trained using 30000 epochs(Figure 12). After the training is completed, the generator produces the required synthetic data (Figure 13)

```
# Define GAN architecture

def build_generator(latent_dim, output_dim):
    model = Sequential()
    model.add(Dense(256, input_dim=latent_dim))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(1024))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(output_dim, activation='tanh'))
    return model

def build_discriminator(input_dim):
    model = Sequential()
    model.add(Dense(1024, input_dim=input_dim))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dense(256))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dense(1, activation='sigmoid'))
    return model

def build_gan(generator, discriminator):
    discriminator.trainable = False
    model = Sequential()
    model.add(generator)
    model.add(discriminator)
    return model
```

Figure 10 Architecture of GAN (D and G)

```
# Create GAN components
latent_dim = 100
output_dim = X_train.shape[1]

generator = build_generator(latent_dim, output_dim)
discriminator = build_discriminator(output_dim)
discriminator.compile(optimizer=Adam(0.0002, 0.5), loss='binary_crossentropy', metrics=['accuracy'])
generator.compile(optimizer=Adam(0.0002, 0.5), loss='binary_crossentropy')
gan = build_gan(generator, discriminator)
gan.compile(optimizer=Adam(0.0002, 0.5), loss='binary_crossentropy')

# Create a scaler and fit it on the real data
scaler = StandardScaler()
scaler.fit(X_train)
```

Figure 11 Loss Function

```
# Training the GAN
epochs = 30000
batch_size = 64

for epoch in range(epochs):
    # Select a random batch of real samples
    idx = np.random.randint(0, X_train.shape[0], batch_size)
    real_samples = X_train.iloc[idx]

    # Generate a batch of synthetic samples
    noise = np.random.normal(0, 1, (batch_size, latent_dim))
    generated_samples = generator.predict(noise)

    # Label real and synthetic samples
    real_labels = np.ones((batch_size, 1))
    fake_labels = np.zeros((batch_size, 1))

    # Reshape real_samples and generated_samples for training
    real_samples = np.array(real_samples)
    real_samples = np.reshape(real_samples, (batch_size, output_dim))
    generated_samples = np.reshape(generated_samples, (batch_size, output_dim))

    # Train the discriminator
    d_loss_real = discriminator.train_on_batch(real_samples, real_labels)
    d_loss_fake = discriminator.train_on_batch(generated_samples, fake_labels)
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

    # Train the generator
    noise = np.random.normal(0, 1, (batch_size, latent_dim))
    valid_labels = np.ones((batch_size, 1))
    g_loss = gan.train_on_batch(noise, valid_labels)
```

Figure 12 Training the model

```
# Evaluate the performance of the generator
num_samples = 500000
noise = np.random.normal(0, 1, (num_samples, latent_dim))
generated_samples = generator.predict(noise)

# Convert synthetic samples back to the original scale
generated_samples = scaler.inverse_transform(generated_samples)

# Display generated samples
print(pd.DataFrame(generated_samples, columns=X_train.columns).head(100))
```

Figure 13 Generating synthetic data.

In Figures 14 and 15, The evaluation of the model is done using MSE,MAE, RMSE and visualization of the data using distribution plot.

```
[29] # Display generated samples
     generated_df = pd.DataFrame(generated_samples, columns=X_train.columns)
     fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(10, 4))
     for i in range(4):  # Adjust the range to match the number of subplots
         ax = axes[i // 5, i % 5]
         ax.hist(generated_df.iloc[:100000, i], bins=30, color='black', alpha=0.5, label='Generated')  # Limit to the first 1000 samples
         ax.hist(X_train.iloc[:, i], bins=30, color='orange', alpha=0.5, label='Real')
         ax.set_title(X_train.columns[i])
         ax.legend()

     plt.tight_layout()
     plt.show()
```

Executing (51m 43s) <cell line: 103> › error_handler() › fit() › steps() › numpy() › read_value() › _read_variable_op() › read_and_set_handle() › r

Figure 15 Visualization of Data

4.3 Implementation of RNN

In Figure 16 the architecture of RNN is shown. The relevant features were selected. In Figure 17, the debugging function is used for checking nulls if any, and reshaping the test and train subsets to match that of the generated data.

```
# Train the model
for epoch in range(5):
    for i in range(len(X_train_reshaped)):
        inputs, targets = X_train_reshaped[i:i+1], y_train_reshaped[i:i+1]
        loss = train_step(inputs, targets)

    print(f"Epoch {epoch + 1}/{50}, Loss: {loss.numpy()}")

# Generate synthetic data using the trained model
synthetic_data = []

# Initialize seed_sequence with the first sequence from the test set
seed_sequence = X_test_subset[0, :-1, :].reshape(1, sequence_length - 1, len(features))

# Batch prediction
batch_size = 32
num_batches = len(X_test_subset) // batch_size

for i in range(num_batches):
    start_idx = i * batch_size
    end_idx = (i + 1) * batch_size

    # Batch of sequences for prediction
    batch_sequences = X_test[start_idx:end_idx, :-1, :]

    # Predict the next values in the sequence for the current batch
    predicted_values_batch = model.predict(batch_sequences)

    # Append the predicted values to the synthetic data
    synthetic_data.extend(predicted_values_batch[:, -1, :])
```

Figure 16 RNN Model

```
# Split the data into training and testing sets
X_train, X_test = train_test_split(sequences, test_size=0.2, shuffle=False)
# Reduce the size of X_test to the first 10,000 sequences
X_test_subset = X_test[:10000]
# Reshape the data for RNN compatibility
X_train_reshaped = X_train[:, :-1, :]  # Use all features except the last time step
y_train_reshaped = X_train[:, 1:, :]   # Use all features except the first time step

# Define the RNN model
model = Sequential()
model.add(SimpleRNN(units=10, activation='relu', input_shape=(sequence_length - 1, len(features)), return_sequences=True))
model.add(Dense(units=len(features)))

# Reducing learning rate and add gradient clipping
optimizer = Adam(learning_rate=0.0001, clipnorm=1.0)
model.compile(optimizer=optimizer, loss='mse')

# Debugging check for NaN values during training
@tf.function
def train_step(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs)
        # Cast predictions to float32 to match the type of targets
        predictions = tf.cast(predictions, dtype=tf.float32)
        loss = tf.reduce_mean(tf.square(predictions - tf.cast(targets, dtype=tf.float32)))
        tf.debugging.check_numerics(loss, "Loss contains NaN or Inf values")
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    return loss
```

Figure 17 RNN Model

Figure 18 shows the generation of synthetic data and the statistical computation for evaluation.

```
# Extracting the last time step predictions for comparison
predicted_values_last_step = predicted_values[:, -1, :]

true_values_original_scale = scaler.inverse_transform(true_values.reshape(-1, len(features)))
predicted_values_original_scale = scaler.inverse_transform(predicted_values_last_step)

# Calculating Mean Squared Error (MSE)
mse = mean_squared_error(true_values_original_scale, predicted_values_original_scale)
print(f"Mean Squared Error (MSE): {mse}")

# Calculate R-squared (MAE)
accuracy_mae = mean_absolute_error(true_values_original_scale, predicted_values_original_scale)
print(f"Mean Absolute Error (MAE): {mse}")

# Plotting true vs. predicted values for each feature
for i, feature in enumerate(features):
    plt.figure(figsize=(10, 6))
    plt.scatter(true_values_original_scale[:, i], predicted_values_original_scale[:, i], color='blue')
    plt.plot([min(true_values_original_scale[:, i]), max(true_values_original_scale[:, i])],
             [min(true_values_original_scale[:, i]), max(true_values_original_scale[:, i])], color='red', linestyle='--')
    plt.title(f'True vs. Predicted values for {feature}')
    plt.xlabel(f'True values for {feature}')
    plt.ylabel(f'Predicted values for {feature}')
    plt.show()

# Visualizing the overall performance
plt.figure(figsize=(10, 6))
plt.scatter(np.arange(len(true_values_original_scale)), true_values_original_scale[:, 0], label='True values', color='blue')
plt.scatter(np.arange(len(predicted_values_original_scale)), predicted_values_original_scale[:, 0], label='Predicted values', color='orange')
plt.title('True vs. Predicted values for the first feature')
```

Figure 18 Generation and Evaluation

4.4 Implementation of Ensembled VAE

Figure 19 shows the libraries needed for the hybrid model with, the whole data split into train and test in the ratio of 80:20. It has also defined the structure of VAE which consists of an encoder and decoder. The trained data is standardized to keep all the inputs on the same scale. Figure 20 shows the building of VAE model and sampling is used to maintain the stochasticity. 3 VAE models are used to achieve the desired model. Figure 21 shows the number of epochs (100 each), the model has to iterate to learn the data before the generation of output. Figure 22 and 23shows the evaluation of the data and model.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.preprocessing import LabelEncoder
import tensorflow as tf
from keras import layers, models, callbacks
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split


# Split data into train and test sets
train_data, test_data = train_test_split(data, test_size=0.2, random_state=42)

# Preprocess data for VAE
vae_data = data.copy()

# Standardize numerical features for VAE
scaler_vae = MinMaxScaler()
vae_data[vae_data.columns] = scaler_vae.fit_transform(vae_data[vae_data.columns])

# Define VAE model
def build_vae_model():
    input_dim = len(data.columns)
    latent_dim = 2
    vae_model = models.Sequential([
        layers.InputLayer(input_shape=(input_dim,)),
        layers.Dense(256, activation='relu'),
        layers.Dense(128, activation='relu'),
        layers.Dense(64, activation='relu'),
        layers.Dense(latent_dim),
    ])
```

Figure 19 VAE model

```
    class Sampling(layers.Layer):
        def call(self, inputs):
            mean = inputs
            log_var = tf.zeros_like(mean)
            return mean + tf.exp(log_var / 2) * tf.random.normal(tf.shape(mean))

    # Build VAE model
    vae_input = layers.Input(shape=(input_dim,))
    vae_intermediate = vae_model(vae_input)
    vae_z = Sampling()(vae_intermediate)
    vae_decoder = models.Sequential([
        layers.InputLayer(input_shape=(latent_dim,)),
        layers.Dense(64, activation='relu'),
        layers.Dense(128, activation='relu'),
        layers.Dense(256, activation='relu'),
        layers.Dense(input_dim, activation='sigmoid'),
    ])
    vae_output = vae_decoder(vae_z)

    vae_model = models.Model(inputs=vae_input, outputs=vae_output)

    return vae_model

# Create an ensemble of VAEs
num_vaes = 3
vae_models = [build_vae_model() for _ in range(num_vaes)]

# Compile each VAE model
for vae_model in vae_models:
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001)
    vae_model.compile(optimizer=optimizer, loss='mean_squared_error')
```

Figure 20 Ensembled VAE

```
    vae_model.compile(optimizer=optimizer, loss='mean_squared_error')

# Train each VAE model
for vae_model in vae_models:
    vae_model.fit(
        vae_data, vae_data,
        epochs=100,
        batch_size=64,
        validation_split=0.2,
        verbose=1
    )

# Generate synthetic samples using the ensemble of VAEs
synthetic_samples_vaes = np.mean([vae_model.predict(test_data[data.columns]) for vae_model in vae_models], axis=0)
```

Figure 21 Model Iteration and Generation of Data

```
[33] # Calculate accuracy (mean squared error, mean absolute error, root mean squared error, r2 score)
     accuracy_mse = mean_squared_error(test_data[data.columns], synthetic_samples_vaes)
     accuracy_mae = mean_absolute_error(test_data[data.columns], synthetic_samples_vaes)
     accuracy_rmse = np.sqrt(accuracy_mse)
     #accuracy_r2 = r2_score(test_data[data.columns], synthetic_samples_vaes)

     # Visualize histogram plots for numerical features
     #for feature in data.columns:
      #   plt.figure(figsize=(10, 5))
       #  plt.hist(test_data[feature], bins=50, alpha=0.5, label='Original Data', color='blue')
        # plt.hist(synthetic_samples_vaes[feature], bins=50, alpha=0.5, label='Synthetic Data (Ensemble of VAEs)', color='orange')
         #plt.title(f'Histogram of {feature}')
         #plt.legend()
         #plt.show()

     # Display the accuracies
     print(f"Model Accuracy (Mean Squared Error): {accuracy_mse:.4f}")
     print(f"Model Accuracy (Mean Absolute Error): {accuracy_mae:.4f}")
     print(f"Model Accuracy (Root Mean Squared Error): {accuracy_rmse:.4f}")
     #print(f"Model Accuracy (R2 Score): {accuracy_r2:.4f}")
```

✓ 24m 5s completed at 15:57

Figure 22 Statistical Analysis

```
# Assuming column_index is the same for both test_data and synthetic_samples_vaes
column_index = 0  # Adjust the index as needed

for feature in data.columns:
    if feature in test_data.columns:
        plt.figure(figsize=(10, 5))
        plt.hist(test_data[feature], bins=50, alpha=0.5, label='Original Data', color='blue')

        # Check if synthetic_samples_vaes is a DataFrame
        if isinstance(synthetic_samples_vaes, pd.DataFrame) and feature in synthetic_samples_vaes.columns:
            plt.hist(synthetic_samples_vaes[feature], bins=50, alpha=0.5, label='Synthetic Data (Ensemble of VAEs)', color='orange')
        elif isinstance(synthetic_samples_vaes, np.ndarray) and synthetic_samples_vaes.shape[1] > column_index:
            plt.hist(synthetic_samples_vaes[:, column_index], bins=50, alpha=0.5, label='Synthetic Data (Ensemble of VAEs)', color='orange')
        else:
            print(f"Feature '{feature}' not found in synthetic_samples_vaes.")

        plt.title(f'Histogram of {feature}')
        plt.legend()
        plt.show()
    else:
        print(f"Feature '{feature}' not found in test_data.")
```

Figure 23 Visualization of the Generated Data

4.5 Implementation of VAEGAN

This model consists of two deep-learning models, Figure 24 shows the VAE structure and Figure 25 consists GAN model structure.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split
import tensorflow as tf
from keras import layers, models, callbacks


# Split data into train and test sets
train_data, test_data = train_test_split(data, test_size=0.2, random_state=42)

# Preprocess data for VAE
vae_data = data.copy()

# Standardize numerical features for VAE
scaler_vae = MinMaxScaler()
vae_data[vae_data.columns] = scaler_vae.fit_transform(vae_data[vae_data.columns])

# Define VAE model
input_dim = len(data.columns)
latent_dim = 2
vae_model = models.Sequential([
    layers.InputLayer(input_shape=(input_dim,)),
    layers.Dense(128, activation='relu'),
    layers.Dense(latent_dim),
])
```

Figure 24 VAE structure of VAEGAN

```
# Define GAN model
generator = models.Sequential([
    layers.InputLayer(input_shape=(latent_dim,)),
    layers.Dense(128, activation='relu'),
    layers.Dense(256, activation='relu'),
    layers.Dense(input_dim, activation='sigmoid'),
])

discriminator = models.Sequential([
    layers.InputLayer(input_shape=(input_dim,)),
    layers.Dense(512, activation='relu'),
    layers.Dense(256, activation='relu'),
    layers.Dense(128, activation='relu'),
    layers.Dense(1, activation='sigmoid'),
])

gan_model = models.Sequential([generator, discriminator])

# Compile the GAN model
gan_model.compile(optimizer='adam', loss='binary_crossentropy')
```

Figure 24 GAN structure of VAEGAN

```
# Compile the GAN model
gan_model.compile(optimizer='adam', loss='binary_crossentropy')

# Connect VAE and GAN models
gan_output = gan_model(vae_z)
combined_model = models.Model(inputs=vae_input, outputs=[vae_output, gan_output])

# Compile the combined model
combined_model.compile(optimizer='adam', loss=['mean_squared_error', 'binary_crossentropy'], loss_weights=[1.0, 0.1])

# Define early stopping callback
early_stopping = callbacks.EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

# Train the combined model
combined_model.fit(
    vae_data, [vae_data, np.ones((len(vae_data), 1))],
    epochs=30,
    batch_size=64,
    validation_split=0.2,
     callbacks=[early_stopping],  # Add the early stopping callback
    verbose=1
)

# Generate synthetic samples using the combined model
synthetic_samples_combined = combined_model.predict(test_data)[0]

# Ensure that the shapes of the predicted and true values are compatible
synthetic_samples_combined = pd.DataFrame(synthetic_samples_combined, columns=data.columns)
```

Figure 25 VAEGAN

The combined model is shown in Figure 25, callback function is used for early stopping and 30 epochs were used for model training. Once training is done combined data is generated.

Figure 26 shows, the evaluation of the model

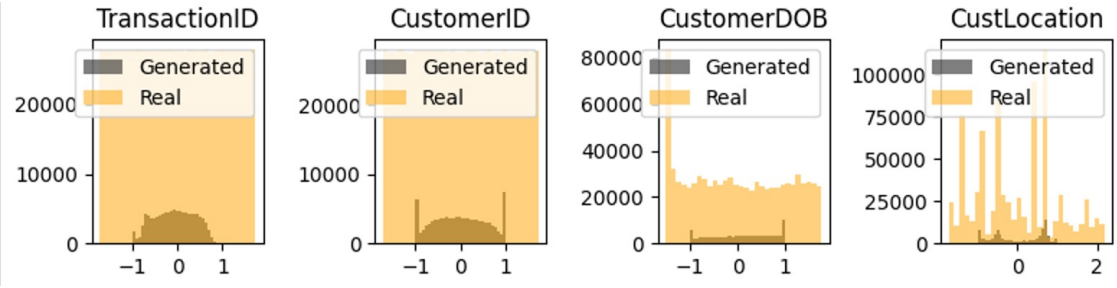
```
# Calculate accuracy (mean squared error, mean absolute error, root mean squared error, r2 score)
accuracy_mse = mean_squared_error(test_data.values, synthetic_samples_combined.values)
accuracy_mae = mean_absolute_error(test_data.values, synthetic_samples_combined.values)
accuracy_rmse = np.sqrt(accuracy_mse)
accuracy_r2 = r2_score(test_data.values, synthetic_samples_combined.values)

# Visualize histogram plots for numerical features
for feature in data.columns:
    plt.figure(figsize=(10, 5))
    plt.hist(test_data[feature], bins=50, alpha=0.5, label='Original Data', color='blue')
    plt.hist(synthetic_samples_combined[feature], bins=50, alpha=0.5, label='Synthetic Data (Combined VAE-GAN)', color='orange')
    plt.title(f'Histogram of {feature}')
    plt.legend()
    plt.show()

# Display the accuracies
print(f"Model Accuracy (Mean Squared Error): {accuracy_mse:.4f}")
print(f"Model Accuracy (Mean Absolute Error): {accuracy_mae:.4f}")
print(f"Model Accuracy (Root Mean Squared Error): {accuracy_rmse:.4f}")
#print(f"Model Accuracy (R2 Score): {accuracy_r2:.4f}")
```

Figure 26 Evaluation

5 Data Evaluation/ Visualization.

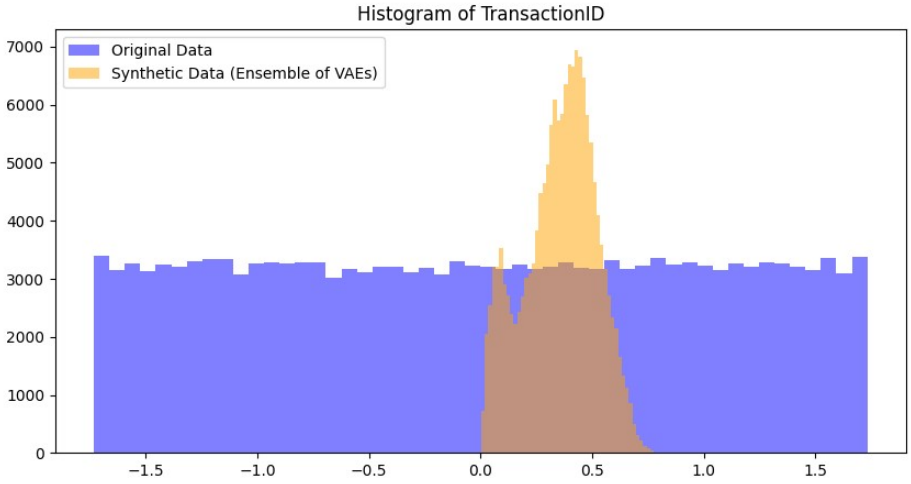
5.1 For GAN Model

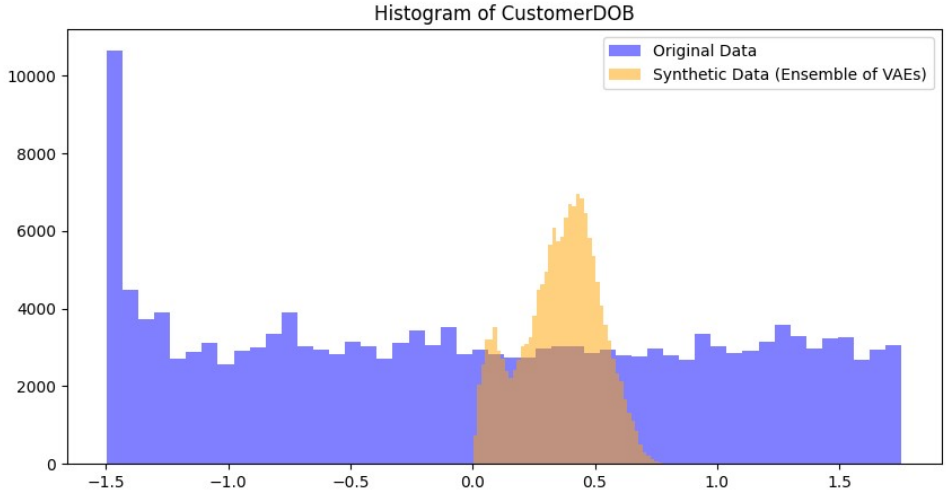
Figure 27 shows the distribution of the generated data.

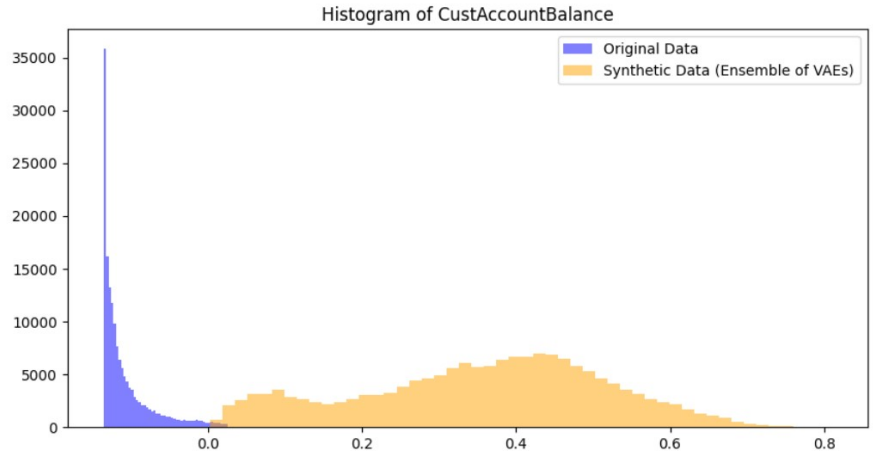


Data distribution of GAN

5.2 For Ensembled VAE

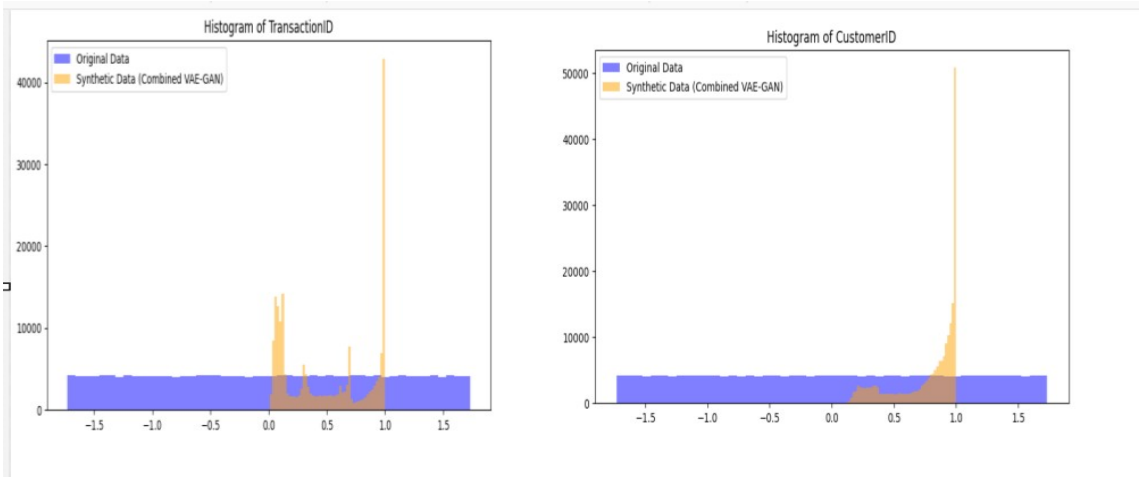






Data distribution of Ensembled VAE

5.3 For VAEGAN



Data distribution of VAEGAN