SQL Assignment Report

Module: Data Mining
Student: Taissir Boukrouba
ID: 22084758
Github: https://github.com/stardustyangel/Generating-synthetic-data-using-python.git

Database General Info :

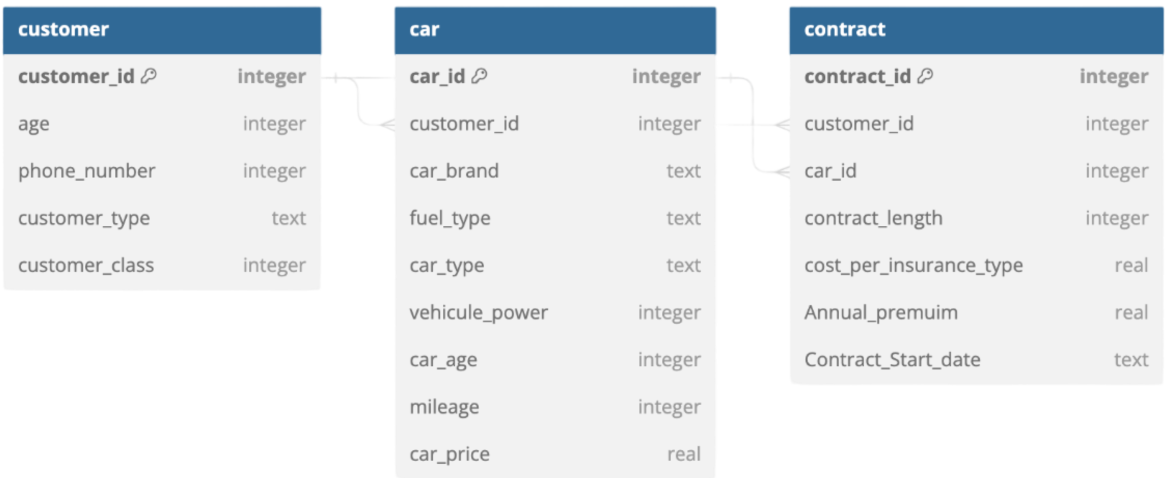
- The database is constructed as follows :

Tables	Features
Customer Table	7 Features
Car Table	9 Features
Contract Table	6 Features

- The database has the following keys :

Table	Attribute	Type
Customer	`customer_id`	Primary Key
Car	`car_id`	Primary Key
Car	`customer_id`	Foreign Key
Contract	`contract_id`	Primary Key
Contract	`(customer_id,car_id)`	Composite Key

Class Diagram :



Schema from SQLite :

- This is the SQLite Schema that we get from :

```
PRAGMA table_info(table_name);
```

	cid	name	type	notnull	dflt_value	pk
1	0	Contract_ID	INTEGER	1	*NULL*	1
2	1	Customer_ID	INTEGER	1	*NULL*	0
3	2	Car_ID	INTEGER	1	*NULL*	0
4	3	Contract_length	INTEGER	0	*NULL*	0
5	4	Insurance_type	TEXT	0	*NULL*	0
6	5	Cost_Per_Insurance_Type	INTEGER	0	*NULL*	0
7	6	Annual_premuim	REAL	0	*NULL*	0
8	7	Contract_Start_date	TEXT	0	*NULL*	0

	cid	name	type	notnull	dflt_value	pk
1	0	customer_ID	INTEGER	1	*NULL*	1
2	1	Age	INTEGER	0	*NULL*	0
3	2	Phone	INTEGER	0	*NULL*	0
4	3	Customer_Type	TEXT	0	*NULL*	0
5	4	Customer_Class	INTEGER	0	*NULL*	0

	cid	name	type	notnull	dflt_value	pk
1	0	Car_ID	INTEGER	1	*NULL*	1
2	1	Customer_ID	INTEGER	1	*NULL*	0
3	2	Car_Brand	TEXT	0	*NULL*	0
4	3	Fuel_Type	TEXT	0	*NULL*	0
5	4	Car_Type	TEXT	0	*NULL*	0
6	5	Vehicule_Power	INTEGER	0	*NULL*	0
7	6	Car_Age	INTEGER	0	*NULL*	0
8	7	Mileage	INTEGER	0	*NULL*	0
9	8	Car_Price	REAL	0	*NULL*	0

- As Asked for the different types of data are present in this database tables:

Types	Features
Nominal	`Car_Brand` , `Name` , `Customer_ID` ...etc
Ordinal Data	`Customer_class` , `Insurance_type` ..etc
Ratio	`Age` , `Vehicule_Power` , `Mileage` ...ect
Interval	`Annual_Premuim` , `Car_price` , `Cost_Per_Insurance_Type` ...etc

Database Generation :

I - Generating features :

Generating IDs :

IDs require the values to be numerical (efficient storage) and unique (identifiers) , which is why we used the following :

- `np.random.randint()` : to generate numerical integers within sensible range
- `np.unique()` : put inside a loop , to ensure uniqueness of the Identifiers

```
#id
customer_id = []
while len(customer_id) < count:
    customer_id = np.unique(np.random.randint(23000000, 24000000, count))

customer_id = customer_id.astype(int)
```

Generating simple features :

Some of these features are basic that used some methods within their sensible range , and we used :

- `np.random.randint()` : to generate random numerical integer features
- `np.random.choice()` : to generate list categorical features

```
#age
age = np.random.randint(20, 90, count)

#phone number
phone_numbers = np.random.randint(10000000000,70000000000,count)

#customer type
customers = ["Individual","Company"]
customer_type = np.random.choice(customers, count,p=[0.95,0.05])
```

Generating huge categorical features :

- Some categorical features like `car brand` require external generated list to make them reasonable and real as follows :

```
#brand
brands = np.loadtxt('/content/Car Brands.csv', delimiter=',',unpack=True,
                    dtype=str)
```

Generating simple dependent features :

- Some features are linked to others , and require some sort of **mapping** (especially when one of the features is categorical) done using a user-defined function as follows :

```
def insurance_cost(insurance_type):
    """
    Calculate the insurance cost based on the provided insurance type.

    Parameters:
    - insurance_type (str): Type of insurance coverage.

    Returns:
    - cost (float or None): The calculated insurance cost as a percentage.
      Returns None if the provided insurance type is not in the switch
      dictionary.
    """
    values = {
        "Liability Insurance": 0.15,
        "Collision Coverage": 0.25,
        "Comprehensive Coverage": 0.15,
        "Uninsured/Underinsured Motorist": 0.1,
        "Personal Injury Protection (PIP)": 0.15,
        "Gap Insurance": 0.05,
        "Custom Parts and Equipment (CPE)": 0.03,
        "Classic Car Insurance": 0.02,
    }
```

Generating complex dependent features :

Generating complex dependent features requires the use of a certain distribution known to the variable :

```
# car ages (right skewed )
num_cars = 1000
shape_parameter = 5  # for skewness
scale_parameter = 1.5  # for the spread
np.random.seed(42)
car_age = np.random.gamma(shape_parameter, scale_parameter, num_cars)
```

> **IMPORTANT :**
>
> > Because there are more younger cars than older cars, and the number of cars decreases as the age increases . This reflects the fact that newer cars are constantly being produced, while older cars gradually get scrapped or exported which is why we used a right-skewed distribution

Generating estimate features (using a formula) :

- Estimate features are the ones we usually try to fit or predict, which means they're complex and dependent on multiple features , which is the reason behind using a formula (linear combination in this case) as follows :

$$car\ price = (a_1 * mileage + a_2 * car\ age + a_3 * vehicule\ power + b) + \epsilon$$

`Car price` , in reality depends on more variables but to make it simple in our case we only used the most basic ones which is the `mileage` , the `car age` and the `vehicule power` :

- `mileage` and `car age` has a ***negative correlation with the car price***
- `vehicule power` has a **positive correlation with car price**

```
# car prices
np.random.seed(0)
error = np.random.normal(500, 5000, count)
b = 7000
a1 = -0.3
a2 = -10
a3 = 500

car_price = (b +(a1 * mileage) + (a2 * car_age) + (a3 * vehicule_power))+ error
```

> **IMPORTANT :**
>
> > Because `mileage` and `car age` are proportionally bigger than `vehicule power` we gave them smaller coefficients to not bias their importance , but in real life these values are to be ***normalised***

II - Creating datasets :

- To create a dataframe , i defined a function that takes the column values and their labels as an input and returns a dataframe as an output as follows :

```
contract_cols = [customer_id,car_id,contract_length,insurance_type,
                 insurance_cost,premuim,Contract_start_date]
contract_labels = ["Customer_ID","Car_ID","Contract_length","Insurance_type",
                   "Cost_Per_Insurance_Type","Annual_premuim",
                   "Contract_Start_date"]
```

```python
# creating dataframes :
def make_frame(columns,labels) :
  """
    Create a DataFrame from given columns and labels.

    Parameters:
    - columns (list of lists): Data for each column.
    - labels (list): Column labels.

    Returns:
    - df (DataFrame): Created DataFrame.
  """
  data = dict(zip(labels, columns))
  df = pd.DataFrame(data)
  return df

beta_customer_df = make_frame(customer_cols,customer_labels)
beta_car_df = make_frame(car_cols,car_labels)
beta_contract_df = make_frame(contract_cols,contract_labels)
```

III - Adding Missing Values :

- To add missing values , i also defined a function that takes a dataframe as an input with the percentage of missing values and return a new dataframe with missing values :

```python
def add_missing_values(df,percentage) :
    """
    Adds missing values to a DataFrame randomly.

    Parameters:
    - df (pandas.DataFrame): The input DataFrame.
    - percentage (float): The percentage of missing values to add to each
    column.

    Returns:
    pandas.DataFrame: DataFrame with added missing values.

    """

    np.random.seed(0)
    num_missing_values = int(np.round(df.shape[0] * percentage))

    for col in df.columns :
      # "ID" col specification
      if 'ID' not in col:
        random_indices = np.random.randint(0, df.shape[0], num_missing_values)
        for i in random_indices :
          df.loc[i,col] = np.nan
    return df


customer_df = add_missing_values(beta_customer_df,0.2)
car_df = add_missing_values(beta_car_df,0.3)
contract_df = add_missing_values(beta_contract_df,0.25)
```

IMPORTANT :

> Because `IDs` are generated from the programmer in the database and not collected real world values , it wouldn't make sense to let them have null values which is why i specified the ID condition in the function

```
customer_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 5 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   customer_ID     1000 non-null   int64  
 1   Age             816 non-null    float64
 2   Phone           814 non-null    float64
 3   Customer_Type   817 non-null    object 
 4   Customer_Class  818 non-null    float64
dtypes: float64(3), int64(1), object(1)
memory usage: 39.2+ KB
```

IV - Exporting the tables :

- Finally , we export the dataframes (tables) as CSV files which we will add to the SQL database .

```
customer_df.to_csv("customer_df.csv",index=False)
car_df.to_csv("car_df.csv",index=False)
contract_df.to_csv("contract_df.csv",index=False)
```

Report Justification

We have created an insurance database with 3 tables , why?

- **Business Logic** : the `car` , `customer` and `contract` represent real world entities relationships which makes the data model easier to understand
- **Accessibility Control :** Separate tables limit access to specific data based on roles (for example customer service might only access customer data while claim agent might access both customer and contract data) which is why i created `customer_ID` key and `car_ID` to control accessibility
- **Threat prevention** : Separating table into `car` , `customer` and `contract` will separate the data breach threat , where a threat in `car` table doesn't affect the `customer` or `contract` table
- **Data Minimisation :** This would be an ***ethical*** approach to take as minimum amount of data as possible from the customer (which reflects the reason why we didn't consider for example `customer address` or `customer name` which can be considered **invasion of privacy**)
- **Optimisation** : Storing data in separate tables , help optimise in terms of time and efficiency

Example Queries :

- Top 10 Cars used by customers aged 30+ :

```sql
SELECT Car_Brand , COUNT(Car_Brand) AS Count FROM car
JOIN customer
ON customer.customer_ID == car.customer_ID
WHERE customer.Age > 30
GROUP BY car.Car_Brand
ORDER BY COUNT(Car_Brand) DESC
LIMIT 10
```

	Car _Brand	Count
1	Ford	58
2	Chevrolet	54
3	Dodge	33
4	Mitsubishi	29
5	Pontiac	27
6	Mercedes-Benz	26
7	Mazda	25
8	Nissan	24
9	GMC	24
10	Volkswagen	22

- Average Annual Premium Cost Per Customer for each insurance Type :

```sql
SELECT Insurance_type ,ROUND( SUM(Annual_premuim) / COUNT(customer.customer_ID) )
AS Averege_Annual_Cost_Per_Customer
FROM contract
INNER JOIN customer
ON contract.Customer_ID == customer.customer_ID
WHERE Insurance_type IS NOT NULL
GROUP BY Insurance_type
ORDER BY Averege_Annual_Cost_Per_Customer
```

	Insurance_type	Averege_Annual_Cost_Per_Customer
1	Classic Car Insurance	4553.0
2	Custom Parts and Equipment (CPE)	7235.0
3	Gap Insurance	10854.0
4	Uninsured/Underinsured Motorist	23306.0
5	Personal Injury Protection (PIP)	33242.0
6	Liability Insurance	34851.0
7	Comprehensive Coverage	39830.0
8	Collision Coverage	59122.0

- Maximum Vehicule Power Per Car Type

```sql
SELECT Car_Type , MAX(Vehicule_Power)
AS Maximum_Engine_Power FROM car
WHERE Car_Type IS NOT NULL
GROUP BY Car_Type
ORDER BY  Maximum_Engine_Power DESC
```

	Car _Type	Maximum_Engine_Power
1	Sports Car	490
2	Electric Car (EV)	487
3	Pickup Truck	449
4	Sedan	398
5	SUV	397
6	Coupe	396
7	Convertible	350
8	Crossover	349
9	Hatchback	299
10	Minivan	297

- Minimum Mileage Per Fuel Type

```sql
SELECT Fuel_Type, ROUND(MIN(Mileage))
AS Minimum_Mileage FROM car
WHERE Fuel_Type IS NOT NULL
GROUP BY Fuel_Type
ORDER BY Minimum_Mileage DESC
```

	Fuel_Type	Minimum_Mileage
1	Electric	27662.0
2	Hybrid	27111.0
3	Diesel	25464.0
4	Petrol	18559.0