Assignment-3

|  |  |
| --- | --- |
| **Summary** | Part 1:  Create a Streamlit interface to execute 5 TPC-DS queries using Snowflake SQLAlchemy, implementing Qualification Substitution Parameters as variables, and validating input values.  Part 2:  Build a CLV model using the TPC-DS dataset, host it, and develop a Streamlit interface for users to input features, invoke the model, and display the CLV predictions. |
| **URL** | https://ne-tpcds-customer-lifetime-valuecustomer-lifetime-value-au9kfr.streamlit.app/ |
| **Category** | Web |
| **Environment** | SnowPark, Jupyter, Streamlit |
| **Status** | Published |
| **Author** | Mohit Negi  Prithvi Vasireddy |

[Introduction](#_ok7k5uux6)

Part-1: Validating and running Queries Via SnowFlake Alchemy in Python

[Getting set up](#_bwo0af2iwk90)

Implementation

Defining Queries and indexing them

Create Engine() and retreive creds from config file

Customer Lifecycle Value Model and Streamlit interface

Part-2: Training and Modeling an XGB model for CLV prediction

Getting Started

Create databases, warehouses, schemas and configure them as relevant

Feature Engineering

[Congratulations](#_s22qmcj4zu61)

# Introduction

Here, we are building a streamlit interface that connects to the Snowflake database, executes TPC-DS queries with Qualification Substitution Parameters as variables, validates user inputs, and provides personalized marketing recommendations based on the estimated lifetime value of each customer.

**Last Updated:** 2023-03-17

## Part-1 – Validating and Running TPCDS Queries via Snowflake SQLAlchemy in Python:

In this part, we build a user-friendly interface for executing TPC-DS queries using Snowflake SQLAlchemy in Python. This interface allows users to interact with the underlying database without needing to write complex SQL queries or understand the intricacies of the database schema. The main goal is to make it easy for users to input necessary values and retrieve the desired results, simplifying the process of data exploration and analysis.

To achieve this, we use Streamlit, a popular open-source framework for creating web applications in Python, as our front-end interface. Streamlit allows us to quickly build interactive and dynamic web applications with minimal effort, making it an ideal choice for this project.

In the back-end, we use Snowflake SQLAlchemy, a powerful and versatile library for connecting to and querying Snowflake databases in Python. By leveraging SQLAlchemy, we can interact with Snowflake seamlessly and execute complex TPC-DS queries with ease.

Throughout the project, we follow the TPC-DS documentation to ensure that we correctly implement the Qualification Substitution Parameters as variables. We also validate user inputs, such as date fields and other specific constraints, to maintain the integrity of the data and the results we generate.

# Getting set up

Bash, SnowFlake

## Installing the required libraries and setting up the snowflake environment with the relevant user/account credentials and warehouse settings as suggested for the Snowflake TPCDS dataset.

1. Create a conda environment using the provided *environment.yml* file.
   * conda env create -f environment.yml
   * Activate that created conda environment by conda activate snowpark\_ml\_test
2. Edit the *creds.json* file to with your account information to connect to your account.
3. Load Jupyter or equivalent notebook to begin executing the notebook.

# Implementation

Jupyter

/\* Import required libraries \*/

|  |
| --- |
| import pandas as pd  import numpy as np  from snowflake.sqlalchemy import URL  from sqlalchemy import create\_engine  import streamlit as st  import sqlalchemy  from streamlit\_lottie import st\_lottie  import requests  import plotly.express as px |

accuracy\_calc() and run\_query().

\**Generate SQL queries based on the input parameters provided to the run\_query() function.* These queries are used to retrieve data on customers' predictions and actual sales, taking into consideration the customers' demographic and financial information.

accuracy\_calc(df) function:

Takes a DataFrame df as input, which is expected to have columns 'predicted\_value' and 'actual\_sales'. Returns the accuracy by calculating the absolute difference between the predicted and actual sales values, divided by the actual sales, and then subtracting this value from 1.

run\_query() function:

Takes the following input parameters: dob\_list, education\_option, gender\_option, dept\_option, credit\_option, and marital\_option. These parameters represent the filter criteria for the query.

The function initializes two base SQL queries, query\_prime1 and query\_prime12, both of which retrieve the sum of predictions and actual sales from a table called 'predictions'.

Then, the function sets up a series of conditionals to check which filter criteria are present in the input parameters.

If a filter criteria has only one value, the corresponding position in the str\_pos list is set to -1, and the query will use an exact match for that filter. If there are multiple values for a filter, the corresponding position in str\_pos is set to 1, and the query will use an IN clause for the filter. If no values are provided for a filter, the corresponding position in str\_pos is set to 0, and the filter is ignored.

Based on the values in the str\_pos list, the function appends the relevant filter conditions to the base SQL queries query\_prime1 and query\_prime12.

For the query\_prime12 query, it finally appends a GROUP BY and ORDER BY clause to group the results by the customers' birth year.

The function returns both generated queries: query\_prime1 and query\_prime12.

Hence, we generate queries based on input parameters to be filtered and retrieve data about customers' predictions and actual sales while considering their demographic and financial information. It also defines an accuracy calculation function for evaluating the predictions.

|  |
| --- |
| **# Define the accuracy\_calc function that takes a DataFrame as input and returns the accuracy value**  **def accuracy\_calc(df):**  **return df['predicted\_value'] / (df['predicted\_value'] + df['actual\_sales'])**  **# Define the run\_query function that takes several input parameters and returns two SQL query strings**  **def run\_query(dob\_list, education\_option, gender\_option, dept\_option, credit\_option, marital\_option):**  **# Initialize query\_prime1 and query\_prime12 with their base SQL query strings**  **query\_prime1 = """select sum(prediction) Predicted, sum(actual\_sales) Actual\_Sales**  **from predictions**  **where c\_birth\_year between """ + str(dob\_list[0]) + " and " + str(dob\_list[1])**  **query\_prime12 = """select c\_birth\_year, sum(prediction) Predicted\_value, sum(actual\_sales) Actual\_Sales**  **from predictions**  **where c\_birth\_year between """ + str(dob\_list[0]) + " and " + str(dob\_list[1])**  **# Initialize an array of flags for query conditions**  **str\_pos = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]**  **# Update str\_pos and gender\_option based on the length of gender\_option**  **if len(gender\_option) == 1:**  **str\_pos[5] = -1**  **str\_pos[0] = 0**  **elif len(gender\_option) >= 1:**  **str\_pos[5] = 0**  **str\_pos[0] = 1**  **else:**  **gender\_option = gender\_option + (1,)**  **str\_pos[5] = 0**  **str\_pos[0] = 0**  **# Update str\_pos and marital\_option based on the length of marital\_option**  **if len(marital\_option) == 1:**  **str\_pos[6] = -1**  **str\_pos[1] = 0**  **elif len(marital\_option) >= 1:**  **str\_pos[6] = 0**  **str\_pos[1] = 1**  **else:**  **marital\_option = marital\_option + (1,)**  **str\_pos[6] = 0**  **str\_pos[1] = 0**  **# Update str\_pos and dept\_option based on the length of dept\_option**  **if len(dept\_option) == 1:**  **str\_pos[7] = -1**  **str\_pos[2] = 0**  **elif len(dept\_option) >= 1:**  **str\_pos[7] = 0**  **str\_pos[2] = 1**  **else:**  **dept\_option = dept\_option + (1,)**  **str\_pos[7] = 0**  **str\_pos[2] = 0**  **# Update str\_pos and credit\_option based on the length of credit\_option**  **if len(credit\_option) == 1:**  **str\_pos[8] = -1**  **str\_pos[3] = 0**  **elif len(credit\_option) >= 1:**  **str\_pos[8] = 0**  **str\_pos[3] = 1**  **else:**  **credit\_option = credit\_option + (1,)**  **str\_pos[8] = 0**  **str\_pos[3] = 0**  **# Update str\_pos and education\_option based on the length of education\_option**  **if len(education\_option) == 1:**  **str\_pos[9] = -1**  **str\_pos[4] = 0**  **elif len(education\_option) >= 1:**  **str\_pos[9]** |

The main() function here sets up two containers for different sections of the Streamlit app. The first container displays "Part - 1" and calls the st\_part1 function, while the second container displays "Part - 2: Customer Lifetime Valuation using XGBoost" and calls the st\_part2 function.

|  |
| --- |
| # Define a function to load a Lottie animation from a URL  def load\_lottieurl(url: str):  r = requests.get(url)  if r.status\_code != 200:  return None  return r.json()  # Define the main function for the Streamlit app  def main():  # Create two containers for different sections of the app  with st.container():  st.header("Part - 1")  st\_part1()  with st.container():  st.header("Part - 2: Customer Lifetime Valuation using XGBoost")  st\_part2()  # Define a function for the first part of the Streamlit app  def st\_part1():  # Create two columns for layout within the container  col1, col2 = st.columns(2, gap='small')  with col1:  # Load the Lottie animation from a URL  lottie\_url\_hello = "https://raw.githubusercontent.com/Negi97Mohit/snowpark-python-demos/main/tpcds-customer-lifetime-value/97474-data-center.json"  lottie\_hello = load\_lottieurl(lottie\_url\_hello)    # Display the Lottie animation  st\_lottie(lottie\_hello,  reverse=True,  height=400,  width=400,  speed=1,  loop=True,  quality='high',  key='Car'  )    # Display the code for generating the query table within an expander  with st.expander("Query table generator code"):  p1code = """import pandas as pd  from snowflake.sqlalchemy import URL  from sqlalchemy import create\_engine |

## Defining Queries (and indexing them)

We first define a list called queries containing five SQL queries in the form of multiline strings. Each SQL query performs a different analytical task on various tables such as customer, store\_sales, web\_sales, catalog\_sales, item, date\_dim, etc. Here's a brief description of each query:

Query11.sql: This query retrieves information about customers who have a higher growth in online sales compared to in-store sales within the given date range. The result is ordered by customer\_id, customer\_first\_name, customer\_last\_name, and customer\_preferred\_cust\_flag and is limited to 100 rows.

Query12.sql: This query calculates the revenue and revenue ratio for items in the 'Sports', 'Books', and 'Home' categories sold within a specific date range. The result is ordered by i\_category, i\_class, i\_item\_id, i\_item\_desc, and revenueratio, and is limited to 100 rows.

Query13.sql: This query calculates the average sales quantities, average sales prices, average wholesale costs, and the total sum of wholesale costs for various customer demographics, store locations, and product price ranges. It also applies specific filters on marital status, education status, and dependent count of the customers.

Query14.sql: This query finds the total sales and number of sales for items sold through three different channels (store, catalog, and web) within a specific date range. The results are aggregated by brand, class, and category, and they are filtered to include only those rows where the sales are greater than the average sales calculated in a subquery. The final result is grouped by rollup (channel, i\_brand\_id, i\_class\_id, i\_category\_id) and is limited to 100 rows.

Query15.sql: This query calculates the total sales price for catalog sales within a given date range, grouped by the zip code of the customer's address. It filters the results based on specific zip codes, states, or sales prices. The result is ordered by ca\_zip and limited to 100 rows.

|  |
| --- |
| queries=[  '''  with year\_total as (  select c\_customer\_id customer\_id  ,c\_first\_name customer\_first\_name  ,c\_last\_name customer\_last\_name  ,c\_preferred\_cust\_flag customer\_preferred\_cust\_flag  ,c\_birth\_country customer\_birth\_country  ,c\_login customer\_login  ,c\_email\_address customer\_email\_address  ,d\_year dyear  ,sum(ss\_ext\_list\_price-ss\_ext\_discount\_amt) year\_total  ,'s' sale\_type  from customer  ,store\_sales  ,date\_dim  where c\_customer\_sk = ss\_customer\_sk  and ss\_sold\_date\_sk = d\_date\_sk  group by c\_customer\_id  ,c\_first\_name  ,c\_last\_name  ,c\_preferred\_cust\_flag  ,c\_birth\_country  ,c\_login  ,c\_email\_address  ,d\_year  union all  select c\_customer\_id customer\_id  ,c\_first\_name customer\_first\_name  ,c\_last\_name customer\_last\_name  ,c\_preferred\_cust\_flag customer\_preferred\_cust\_flag  ,c\_birth\_country customer\_birth\_country  ,c\_login customer\_login  ,c\_email\_address customer\_email\_address  ,d\_year dyear  ,sum(ws\_ext\_list\_price-ws\_ext\_discount\_amt) year\_total  ,'w' sale\_type  from customer  ,web\_sales  ,date\_dim  where c\_customer\_sk = ws\_bill\_customer\_sk  and ws\_sold\_date\_sk = d\_date\_sk  group by c\_customer\_id  ,c\_first\_name  ,c\_last\_name  ,c\_preferred\_cust\_flag  ,c\_birth\_country  ,c\_login  ,c\_email\_address  ,d\_year  )  select  t\_s\_secyear.customer\_id  ,t\_s\_secyear.customer\_first\_name  ,t\_s\_secyear.customer\_last\_name  ,t\_s\_secyear.customer\_preferred\_cust\_flag  from year\_total t\_s\_firstyear  ,year\_total t\_s\_secyear  ,year\_total t\_w\_firstyear  ,year\_total t\_w\_secyear  where t\_s\_secyear.customer\_id = t\_s\_firstyear.customer\_id  and t\_s\_firstyear.customer\_id = t\_w\_secyear.customer\_id  and t\_s\_firstyear.customer\_id = t\_w\_firstyear.customer\_id  and t\_s\_firstyear.sale\_type = 's'  and t\_w\_firstyear.sale\_type = 'w'  and t\_s\_secyear.sale\_type = 's'  and t\_w\_secyear.sale\_type = 'w'  and t\_s\_firstyear.dyear = 2001  and t\_s\_secyear.dyear = 2001+1  and t\_w\_firstyear.dyear = 2001  and t\_w\_secyear.dyear = 2001+1  and t\_s\_firstyear.year\_total > 0  and t\_w\_firstyear.year\_total > 0  and case when t\_w\_firstyear.year\_total > 0 then t\_w\_secyear.year\_total / t\_w\_firstyear.year\_total else 0.0 end  > case when t\_s\_firstyear.year\_total > 0 then t\_s\_secyear.year\_total / t\_s\_firstyear.year\_total else 0.0 end  order by t\_s\_secyear.customer\_id  ,t\_s\_secyear.customer\_first\_name  ,t\_s\_secyear.customer\_last\_name  ,t\_s\_secyear.customer\_preferred\_cust\_flag  limit 100''',  '''select i\_item\_id  ,i\_item\_desc  ,i\_category  ,i\_class  ,i\_current\_price  ,sum(ws\_ext\_sales\_price) as itemrevenue  ,sum(ws\_ext\_sales\_price)\*100/sum(sum(ws\_ext\_sales\_price)) over  (partition by i\_class) as revenueratio  from  web\_sales  ,item  ,date\_dim  where  ws\_item\_sk = i\_item\_sk  and i\_category in ('Sports', 'Books', 'Home')  and ws\_sold\_date\_sk = d\_date\_sk  and d\_date between cast('1999-02-22' as date)  and dateadd(day,30,cast('1999-02-22' as date) )  group by  i\_item\_id  ,i\_item\_desc  ,i\_category  ,i\_class  ,i\_current\_price  order by  i\_category  ,i\_class  ,i\_item\_id  ,i\_item\_desc  ,revenueratio  limit 100''',  ''' select avg(ss\_quantity)  ,avg(ss\_ext\_sales\_price)  ,avg(ss\_ext\_wholesale\_cost)  ,sum(ss\_ext\_wholesale\_cost)  from store\_sales  ,store  ,customer\_demographics  ,household\_demographics  ,customer\_address  ,date\_dim  where s\_store\_sk = ss\_store\_sk  and ss\_sold\_date\_sk = d\_date\_sk and d\_year = 2001  and((ss\_hdemo\_sk=hd\_demo\_sk  and cd\_demo\_sk = ss\_cdemo\_sk  and cd\_marital\_status = 'M'  and cd\_education\_status = 'Advanced Degree'  and ss\_sales\_price between 100.00 and 150.00  and hd\_dep\_count = 3  )or  (ss\_hdemo\_sk=hd\_demo\_sk  and cd\_demo\_sk = ss\_cdemo\_sk  and cd\_marital\_status = 'S'  and cd\_education\_status = 'College'  and ss\_sales\_price between 50.00 and 100.00  and hd\_dep\_count = 1  ) or  (ss\_hdemo\_sk=hd\_demo\_sk  and cd\_demo\_sk = ss\_cdemo\_sk  and cd\_marital\_status = 'W'  and cd\_education\_status = '2 yr Degree'  and ss\_sales\_price between 150.00 and 200.00  and hd\_dep\_count = 1  ))  and((ss\_addr\_sk = ca\_address\_sk  and ca\_country = 'United States'  and ca\_state in ('TX', 'OH', 'TX')  and ss\_net\_profit between 100 and 200  ) or  (ss\_addr\_sk = ca\_address\_sk  and ca\_country = 'United States'  and ca\_state in ('OR', 'NM', 'KY')  and ss\_net\_profit between 150 and 300  ) or  (ss\_addr\_sk = ca\_address\_sk  and ca\_country = 'United States'  and ca\_state in ('VA', 'TX', 'MS')  and ss\_net\_profit between 50 and 250  ))  ''',  '''-- start query 14 in stream 0 using template query14.tpl and seed QUALIFICATION  with cross\_items as  (select i\_item\_sk ss\_item\_sk  from item,  (select iss.i\_brand\_id brand\_id  ,iss.i\_class\_id class\_id  ,iss.i\_category\_id category\_id  from store\_sales  ,item iss  ,date\_dim d1  where ss\_item\_sk = iss.i\_item\_sk  and ss\_sold\_date\_sk = d1.d\_date\_sk  and d1.d\_year between 1999 AND 1999 + 2  intersect  select ics.i\_brand\_id  ,ics.i\_class\_id  ,ics.i\_category\_id  from catalog\_sales  ,item ics  ,date\_dim d2  where cs\_item\_sk = ics.i\_item\_sk  and cs\_sold\_date\_sk = d2.d\_date\_sk  and d2.d\_year between 1999 AND 1999 + 2  intersect  select iws.i\_brand\_id  ,iws.i\_class\_id  ,iws.i\_category\_id  from web\_sales  ,item iws  ,date\_dim d3  where ws\_item\_sk = iws.i\_item\_sk  and ws\_sold\_date\_sk = d3.d\_date\_sk  and d3.d\_year between 1999 AND 1999 + 2)  where i\_brand\_id = brand\_id  and i\_class\_id = class\_id  and i\_category\_id = category\_id  ),  avg\_sales as  (select avg(quantity\*list\_price) average\_sales  from (select ss\_quantity quantity  ,ss\_list\_price list\_price  from store\_sales  ,date\_dim  where ss\_sold\_date\_sk = d\_date\_sk  and d\_year between 1999 and 1999 + 2  union all  select cs\_quantity quantity  ,cs\_list\_price list\_price  from catalog\_sales  ,date\_dim  where cs\_sold\_date\_sk = d\_date\_sk  and d\_year between 1999 and 1999 + 2  union all  select ws\_quantity quantity  ,ws\_list\_price list\_price  from web\_sales  ,date\_dim  where ws\_sold\_date\_sk = d\_date\_sk  and d\_year between 1999 and 1999 + 2) x)  select channel, i\_brand\_id,i\_class\_id,i\_category\_id,sum(sales), sum(number\_sales)  from(  select 'store' channel, i\_brand\_id,i\_class\_id  ,i\_category\_id,sum(ss\_quantity\*ss\_list\_price) sales  , count(\*) number\_sales  from store\_sales  ,item  ,date\_dim  where ss\_item\_sk in (select ss\_item\_sk from cross\_items)  and ss\_item\_sk = i\_item\_sk  and ss\_sold\_date\_sk = d\_date\_sk  and d\_year = 1999+2  and d\_moy = 11  group by i\_brand\_id,i\_class\_id,i\_category\_id  having sum(ss\_quantity\*ss\_list\_price) > (select average\_sales from avg\_sales)  union all  select 'catalog' channel, i\_brand\_id,i\_class\_id,i\_category\_id, sum(cs\_quantity\*cs\_list\_price) sales, count(\*) number\_sales  from catalog\_sales  ,item  ,date\_dim  where cs\_item\_sk in (select ss\_item\_sk from cross\_items)  and cs\_item\_sk = i\_item\_sk  and cs\_sold\_date\_sk = d\_date\_sk  and d\_year = 1999+2  and d\_moy = 11  group by i\_brand\_id,i\_class\_id,i\_category\_id  having sum(cs\_quantity\*cs\_list\_price) > (select average\_sales from avg\_sales)  union all  select 'web' channel, i\_brand\_id,i\_class\_id,i\_category\_id, sum(ws\_quantity\*ws\_list\_price) sales , count(\*) number\_sales  from web\_sales  ,item  ,date\_dim  where ws\_item\_sk in (select ss\_item\_sk from cross\_items)  and ws\_item\_sk = i\_item\_sk  and ws\_sold\_date\_sk = d\_date\_sk  and d\_year = 1999+2  and d\_moy = 11  group by i\_brand\_id,i\_class\_id,i\_category\_id  having sum(ws\_quantity\*ws\_list\_price) > (select average\_sales from avg\_sales)  ) y  group by rollup (channel, i\_brand\_id,i\_class\_id,i\_category\_id)  order by channel,i\_brand\_id,i\_class\_id,i\_category\_id  limit 100;''',    '''select ca\_zip  ,sum(cs\_sales\_price)  from catalog\_sales  ,customer  ,customer\_address  ,date\_dim  where cs\_bill\_customer\_sk = c\_customer\_sk  and c\_current\_addr\_sk = ca\_address\_sk  and ( substr(ca\_zip,1,5) in ('85669', '86197','88274','83405','86475',  '85392', '85460', '80348', '81792')  or ca\_state in ('CA','WA','GA')  or cs\_sales\_price > 500)  and cs\_sold\_date\_sk = d\_date\_sk  and d\_qoy = 2 and d\_year = 2001  group by ca\_zip  order by ca\_zip  limit 100'''  ] |

### Create\_Engine() and retrieve snowflake credentials from configuration file – define tables:

Define two Snowflake database connections using the create\_engine function from the SQLAlchemy library. These connections are used to query data from two different schemas within the same Snowflake account.

tables: This is a list of strings containing the names of the tables (query1, query2, query3, query4, and query5) that will be used later in the code.

These two connections, engine\_old and engine\_new, can be used to execute SQL queries on the respective databases and schemas.

For improved readability and maintainability, store sensitive information like user, password, and account details in environment variables or a configuration file. Here's the updated code:

import os

from sqlalchemy import create\_engine

from snowflake.sqlalchemy import URL

tables = ['query1', 'query2', 'query3', 'query4', 'query5']

engine\_old = create\_engine(URL(

account = os.environ['SNOWFLAKE\_ACCOUNT'],

user = os.environ['SNOWFLAKE\_USER'],

password = os.environ['SNOWFLAKE\_PASSWORD'],

database = 'SNOWFLAKE\_SAMPLE\_DATA',

schema = 'TPCDS\_SF10TCL',

warehouse = 'COMPUTE\_WH',

role = 'ACCOUNTADMIN',

))

engine\_new = create\_engine(URL(

account = os.environ['SNOWFLAKE\_ACCOUNT'],

user = os.environ['SNOWFLAKE\_USER'],

password = os.environ['SNOWFLAKE\_PASSWORD'],

database = 'midterm',

schema = 'public',

warehouse = 'COMPUTE\_WH',

role = 'ACCOUNTADMIN',

))

|  |
| --- |
| tables=['query1','query2','query3','query4','query5']  engine\_old= create\_engine(URL(  account = 'pinvdzu-ljb05593',  user = 'gamer9797123',  password = 'Gamer9797123)',  database = 'SNOWFLAKE\_SAMPLE\_DATA',  schema = 'TPCDS\_SF10TCL',  warehouse = 'COMPUTE\_WH',  role='ACCOUNTADMIN',  ))  engine\_new = create\_engine(URL(  account = 'pinvdzu-ljb05593',  user = 'gamer9797123',  password = 'Gamer9797123)',  database = 'midterm',  schema = 'public',  warehouse = 'COMPUTE\_WH',  role='ACCOUNTADMIN',  )) |

## CLV Model AND Streamlit Interface

|  |
| --- |
| # Iterate through the list of queries and corresponding table names  for query, table\_name in zip(queries, tables):      # Connect to the old database engine      connection = engine\_old.connect()      print(table\_name, "old Connected")        # Read data from the old database using the query      res = pd.read\_sql(query, connection)        # Dispose the old database connection      engine\_old.dispose()        # Connect to the new database engine      connection = engine\_new.connect()        # Write the data to the new database table      res.to\_sql(table\_name, connection, index=False, if\_exists='replace')      print(table\_name, "new  Connected")        # Dispose the new database connection      engine\_new.dispose()    # Display the code block in the Streamlit app  st.code(p1code, language='python')    # In the second column, iterate through the list of questions, table names, and tabs  for table\_name, tab, question in zip(tables, tab\_names, questions):      with tab:          # Create a SQL query to fetch data from the table          query = "select \* from " + str(table\_name)            # Connect to the part1\_engine database          connection = part1\_engine.connect()            # Read data from the table using the query          res = pd.read\_sql(query, connection)            # Display the query number and question in the Streamlit app          title = "Query number: " + str(table\_name)          st.header(title)          st.code(question)            # Display the result in the Streamlit app          st.write(res)        # Dispose the part1\_engine database connection      part1\_engine.dispose()    # Function to run the queries for the Streamlit app part 2  def run\_query(dob\_list, education\_option, gender\_option, dept\_option, credit\_option, marital\_option):      # Define the SQL queries with the selected options      # Replace this comment with the actual SQL query definitions      return query, query2    # Main function for the Streamlit app  def main():      # Define the part2\_engine database connection      part2\_engine = create\_engine(URL(          account = 'pinvdzu-ljb05593',          user = 'gamer9797123',          password = 'Gamer9797123)',          database = 'TPCDS\_XGBOOST',          schema = 'DEMO',          warehouse = 'COMPUTE\_WH',          role='ACCOUNTADMIN',      ))      # Connect to the part2\_engine database      connection = part2\_engine.connect()        # Use columns and widgets to create the Streamlit app layout      col1, col2 = st.columns(2)      with col1:          # Add Streamlit widgets for user inputs, like sliders and multiselects          # Replace the following comments with actual Streamlit widgets          # dob\_slider = ...          # education\_option = ...          # gender\_option = ...          # dept\_option = ...          # credit\_option = ...          # marital\_option = ...            # Call the run\_query function to get the SQL queries          query, query2 = run\_query(dob\_list, education\_option, gender\_option, dept\_option, credit\_option, marital\_option)            # Fetch data from the database using the queries and display the results in the app          res2 = pd.read\_sql(query, connection)         # Fetch data from the database using the queries and display the results in the app  # Execute the SQL query by passing it to the `read\_sql` function along with the database connection  **res2 = pd.read\_sql(query, connection)**  # The `res2` variable now contains a DataFrame with the results of the executed SQL query.  # We assume that the first row (0) and the first column (0) of the DataFrame contain the predicted sales value.  # Similarly, we assume that the first row (0) and the second column (1) of the DataFrame contain the actual sales value.  # To display the predicted sales value, we access the value using the `loc` function of the DataFrame:  # res2.loc[0][0] - This accesses the value in the first row and the first column of the DataFrame.  # We then divide the value by 1,000,000,000 (1 billion) to convert it into billions.  # Finally, we round the value to 2 decimal places using the `round` function and display it in the Streamlit app.  **st.write('Predicted Sales:', round(res2.loc[0][0] / 1000000000, 2), 'billion USD')**  # Similarly, to display the actual sales value, we access the value using the `loc` function of the DataFrame:  # res2.loc[0][1] - This accesses the value in the first row and the second column of the DataFrame.  # We then divide the value by 1,000,000,000 (1 billion) to convert it into billions.  # Finally, we round the value to 2 decimal places using the `round` function and display it in the Streamlit app.  **st.write('Actual Sales:', round(res2.loc[0][1] / 1000000000, 2), 'billion USD')** |

Customer\_lifetime\_value.py -- running this will run: https://ne-tpcds-customer-lifetime-valuecustomer-lifetime-value-au9kfr.streamlit.app/

\*\*Part 1 of the application provides an interface to execute five assigned TPC-DS queries using Snowflake SQLAlchemy in Python. Users can input the required variables following the Qualification Substitution Parameters in the TPC-DS documentation. The application allows users to validate their input values for specific constraints, such as date fields. The interface displays the results of the executed queries.

Part 2 of the application allows users to input customer

features and obtain their customer lifetime value (CLV) prediction. The

application employs a trained CLV model using the TPC-DS dataset. The feature

engineering and model training are executed through the xgboost\_tpcds.ipynb

Jupyter notebook, where the model is trained and hosted on a platform. The

## **Customer\_lifetime\_value.py** file is the main Streamlit application that interacts with

the hosted model, allowing users to input customer features and view their CLV

prediction.

To use the CLV application, users can input customer

features, such as the birth year, gender, marital status, credit rating,

education status, and dependents count. The application uses these features as

inputs to the trained XGBoost model to calculate the customer's expected

lifetime value. The interface displays the predicted CLV value, allowing users

to understand the potential value of each customer to their business.

Overall, the web application provides a user-friendly

interface for both querying the TPC-DS dataset and predicting customer lifetime

value. The application is built using Snowflake SQLAlchemy in Python, and it

provides a great example of how to build and deploy machine learning models

with Snowflake's data platform.

## Part-2 – Training and modeling an XGBoost model for CLV prediction (xgboost\_tpcds.ipynb)

## Calculating Customer Lifetime Value (CLV) from provided TPC-DS dataset

The objective here is to build a model for calculating Customer Lifetime Value (CLV) using the provided TPC-DS dataset. CLV is a key performance indicator for any business that is looking to grow and retain its customer base. The CLV model aims to predict the total value that a customer will generate for a company over the lifetime of their relationship.

To achieve this goal, the first step is to perform feature engineering and data preprocessing on the TPC-DS dataset. This includes aggregating sales data across all channels (web, store, catalogue) and joining it to customer demographic data. The resulting dataset is then split into training and testing sets, and a machine learning model is trained on the prepared data.

In this project, an XGBoost model is used for training, which is a popular gradient boosting framework that has been shown to perform well on many different types of datasets. Once the model is trained, it is saved and hosted on a cloud platform to be accessed by the Streamlit interface.

The Streamlit interface developed in this project allows users to input the necessary features for CLV calculation, such as birth year, gender, marital status, credit rating, education status, and zip code. Upon invoking the model, the interface displays the predicted CLV for the given customer.

# Getting set up

Import necessary libraries and sets up a connection to a Snowflake database using Snowpark, which is a DataFrame API provided by Snowflake for data processing and transformation.

Import libraries:

The code imports the required libraries and modules, including snowflake.snowpark, functions as F, Session, and version as v. It also imports the json library to handle JSON data.

Load connection details from a JSON file:

The code reads a JSON file named connection.json to obtain the Snowflake connection details, such as the username, password, account, and warehouse.

Set up connection parameters:

The CONNECTION\_PARAMETERS dictionary is created using the details loaded from the JSON file.

Create a Snowpark session:

The Session object is created by calling the Session.builder.configs() method and passing the CONNECTION\_PARAMETERS dictionary. The create() method is then called to establish the connection to the Snowflake database.

|  |
| --- |
| // importing libraries and setting up snowflake  import snowflake.snowpark  from snowflake.snowpark import functions as F  from snowflake.snowpark.session import Session  from snowflake.snowpark import version as v  import json  with open('connection.json') as f:  data = json.load(f)  USERNAME = data['user']  PASSWORD = data['password']  SF\_ACCOUNT = data['account']  SF\_WH = data['warehouse']  CONNECTION\_PARAMETERS = {  "account": SF\_ACCOUNT,  "user": USERNAME,  "password": PASSWORD,  }  session = Session.builder.configs(CONNECTION\_PARAMETERS).create() |

# Create Databases, warehouses, schemas and configure them

Set up the Snowflake environment by creating necessary databases, schemas, and warehouses, and configuring warehouse settings to ensure the proper resources are allocated for the tasks at hand.

|  |
| --- |
| # Create the 'snowflake\_sample\_data' database if it doesn't exist, using the shared 'sample\_data' from 'sfc\_samples'  **session.sql('''create database if not exists snowflake\_sample\_data from share sfc\_samples.sample\_data''').collect()**  # Create the 'tpcds\_xgboost' database if it doesn't exist  session.sql('CREATE DATABASE IF NOT EXISTS tpcds\_xgboost').collect()  # Create the 'demo' schema within the 'tpcds\_xgboost' database if it doesn't exist  **session.sql('CREATE SCHEMA IF NOT EXISTS tpcds\_xgboost.demo').collect()**  # Create or replace a warehouse named 'FE\_AND\_INFERENCE\_WH' with a warehouse size of '3X-LARGE'  **session.sql("create or replace warehouse FE\_AND\_INFERENCE\_WH with warehouse\_size='3X-LARGE'").collect()**  # Create or replace a warehouse named 'snowpark\_opt\_wh' with a warehouse size of 'MEDIUM' and type 'SNOWPARK-OPTIMIZED'  **session.sql("create or replace warehouse snowpark\_opt\_wh with warehouse\_size = 'MEDIUM' warehouse\_type = 'SNOWPARK-OPTIMIZED'").collect()**  # Set the maximum concurrency level of the 'snowpark\_opt\_wh' warehouse to 1  **session.sql("alter warehouse snowpark\_opt\_wh set max\_concurrency\_level = 1").collect()**  # Use the 'FE\_AND\_INFERENCE\_WH' warehouse for the session  **session.use\_warehouse('FE\_AND\_INFERENCE\_WH')**  # Instructions for selecting the TPC-DS Dataset size (100 or 10) to use and the recommended warehouse size |

\*\*\*\*See the link provided for more information on the dataset size

Set the TPC-DS dataset size, defines the Snowflake Sample Database name, and determines the schema based on the dataset size. Then, it accesses the relevant tables in the sample database using the determined schema.

|  |
| --- |
| # Set the TPC-DS dataset size parameter (choose either 100 or 10)  TPCDS\_SIZE\_PARAM = 100  # Define the name of the Snowflake Sample Database (might be different depending on your setup)  SNOWFLAKE\_SAMPLE\_DB = 'SNOWFLAKE\_SAMPLE\_DATA'  # Determine the TPCDS schema based on the chosen dataset size  **if TPCDS\_SIZE\_PARAM == 100:**  **TPCDS\_SCHEMA = 'TPCDS\_SF100TCL'**  **elif TPCDS\_SIZE\_PARAM == 10:**  **TPCDS\_SCHEMA = 'TPCDS\_SF10TCL'**  **else:**  **# Raise an error if an invalid dataset size is selected**  **raise ValueError("Invalid TPCDS\_SIZE\_PARAM selection")**  # Access the relevant tables in the sample database using the determined schema  **store\_sales = session.table(f'{SNOWFLAKE\_SAMPLE\_DB}.{TPCDS\_SCHEMA}.store\_sales')**  **catalog\_sales = session.table(f'{SNOWFLAKE\_SAMPLE\_DB}.{TPCDS\_SCHEMA}.catalog\_sales')**  **web\_sales = session.table(f'{SNOWFLAKE\_SAMPLE\_DB}.{TPCDS\_SCHEMA}.web\_sales')**    **date = session.table(f'{SNOWFLAKE\_SAMPLE\_DB}.{TPCDS\_SCHEMA}.date\_dim')**  **dim\_stores = session.table(f'{SNOWFLAKE\_SAMPLE\_DB}.{TPCDS\_SCHEMA}.store')**  **customer = session.table(f'{SNOWFLAKE\_SAMPLE\_DB}.{TPCDS\_SCHEMA}.customer')**  **address = session.table(f'{SNOWFLAKE\_SAMPLE\_DB}.{TPCDS\_SCHEMA}.customer\_address')**  **demo = session.table(f'{SNOWFLAKE\_SAMPLE\_DB}.{TPCDS\_SCHEMA}.customer\_demographics')** |

## Feature Engineering

Aggregate sales data across all channels (web, store, and catalog) and joining the resulting data with customer demographic data. It calculates total sales for each customer and then unions the results for all channels. Finally, it joins the aggregated data with the customer, address, and customer\_demographics tables to create a complete dataset for further analysis.

|  |
| --- |
| # Set the TPC-DS dataset size parameter (choose either 100 or 10)  TPCDS\_SIZE\_PARAM = 100  # Define the name of the Snowflake Sample Database (might be different depending on your setup)  # Aggregate sales by customer across all channels (web, store, catalog) and join that to customer demographic data  # Group by customer ID and calculate total sales for store\_sales  store\_sales\_agged = store\_sales.group\_by('ss\_customer\_sk').agg(F.sum('ss\_sales\_price').as\_('total\_sales'))  # Group by customer ID and calculate total sales for web\_sales  web\_sales\_agged = web\_sales.group\_by('ws\_bill\_customer\_sk').agg(F.sum('ws\_sales\_price').as\_('total\_sales'))  # Group by customer ID and calculate total sales for catalog\_sales  catalog\_sales\_agged = catalog\_sales.group\_by('cs\_bill\_customer\_sk').agg(F.sum('cs\_sales\_price').as\_('total\_sales'))  # Rename columns to match customer\_sk across all tables  store\_sales\_agged = store\_sales\_agged.rename('ss\_customer\_sk', 'customer\_sk')  web\_sales\_agged = web\_sales\_agged.rename('ws\_bill\_customer\_sk', 'customer\_sk')  catalog\_sales\_agged = catalog\_sales\_agged.rename('cs\_bill\_customer\_sk', 'customer\_sk')  # Union store\_sales\_agged and web\_sales\_agged  total\_sales = store\_sales\_agged.union\_all(web\_sales\_agged)  # Union the result with catalog\_sales\_agged  total\_sales = total\_sales.union\_all(catalog\_sales\_agged)  # Group by customer\_sk and calculate the total\_sales  total\_sales = total\_sales.group\_by('customer\_sk').agg(F.sum('total\_sales').as\_('total\_sales'))  # Select relevant columns from the customer table  customer = customer.select('c\_customer\_sk', 'c\_current\_hdemo\_sk', 'c\_current\_addr\_sk', 'c\_customer\_id', 'c\_birth\_year')  # Join the customer table with the address table  customer = customer.join(address.select('ca\_address\_sk', 'ca\_zip'), customer['c\_current\_addr\_sk'] == address['ca\_address\_sk'])  # Join the customer table with the customer\_demographics table  customer = customer.join(demo.select('cd\_demo\_sk', 'cd\_gender', 'cd\_marital\_status', 'cd\_credit\_rating', 'cd\_education\_status', 'cd\_dep\_count'),  customer['c\_current\_hdemo\_sk'] == demo['cd\_demo\_sk'])  # Rename 'c\_customer\_sk' column to 'customer\_sk'  customer = customer.rename('c\_customer\_sk', 'customer\_sk')  customer.show() |

# Congratulations

Congratulations, we now have a Streamlit interface that connects to the Snowflake database using sqlalchemy in Python and executes the TPC-DS queries. The interface allows users to input Qualification Substitution Parameters as variables and validates user inputs to ensure they are within the allowed range of values. We also have a model that estimates the lifetime value of each customer and segments them based on their value to the business. We can use this information to create personalized marketing campaigns to retain high-value customers. We can host the model using cloud services and create a Streamlit interface that allows users to input customer data and receive personalized marketing recommendations.