Low Level Design

CPA (Customer Personality Analysis)

Written By	Katta Sai Pranav Reddy
Document Version	0.1
Last Revised Date	16-10-2024

Document Control

Change Record:

Version	Date	Author	Comments
0.1	12-10-2024	Katta Sai Pranav Reddy	Introduction and architecture define
0.2	14-10-2024	Katta Sai Pranav Reddy	Architecture ,Architecture Description

Reviews:

Version	Date	Reviewer	Comments
0.2	16 – oct - 2024	Khusali	Document Content, Version Control and Unit Test Cases to be added

Approval Status:

Version	Review Date	Reviewed By	Approved By	Comments

Contents

3.9.3. GitHub Actions and AWS EC2 Integration:

3.10. Deployment

1 Introduction

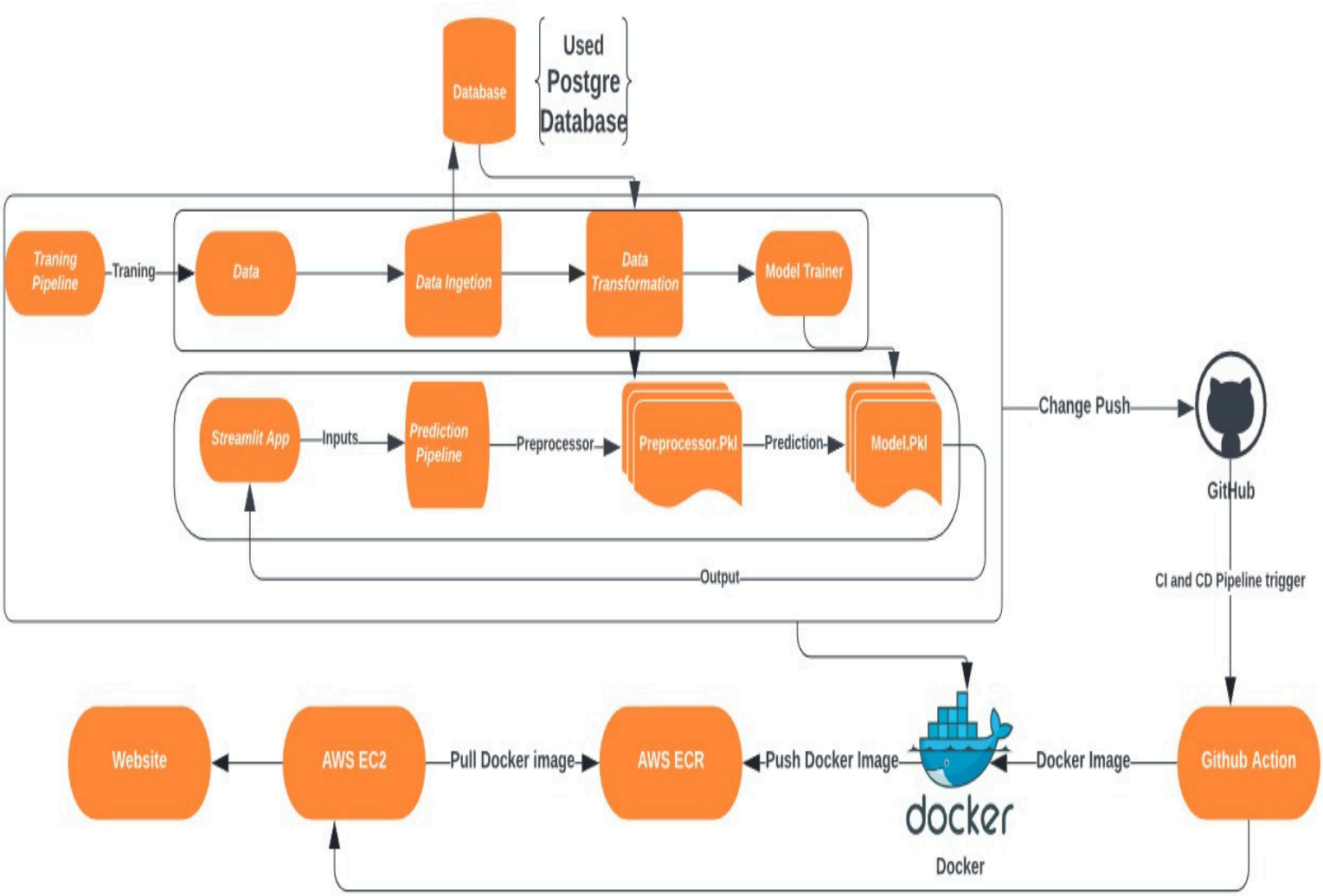
3.11. What is Low-Level design document?

The goal of LLD or a low-level design document (LLDD) is to give the internal logical design of the actual program code for Food Recommendation System. LLD describes the class diagrams with the methods and relations between classes and program specs. It describes the modules so that the programmer can directly code the program from the document.

3.12. Scope

Low-level design (LLD) is a component-level design process that follows a step-by-step refinement process. This process can be used for designing data structures, required software architecture, source code and ultimately, performance algorithms. Overall, the data organization may be defined during requirement analysis and then refined during data design work

4. Architecture



5. Architecture Description

5.1. Data Description

Column Name	Description
id	Unique identifier for each customer.
Year_birth	Birth year of the customer.
Education	Highest level of education completed by the customer (e.g., Graduation).
Marital_status	Marital status of the customer (e.g., Single, Together).
Income	Annual income of the customer.
Kidhome	Number of young children in the customer's household.
Teenhome	Number of teenagers in the customer's household.
dt_customer	Date when the customer became a client of the company.
recency	Number of days since the customer's last purchase.
mntwines	Total spending on wines by the customer.
mntfruits	Total spending on fruits by the customer.
mntmeatproducts	Total spending on meat products by the customer.
mntfishproducts	Total spending on fish products by the customer.
mntsweetproducts	Total spending on sweet products by the customer.
mntgoldprods	Total spending on gold products by the customer.
numdealspurchases	Number of purchases made using deals or discounts.
numwebpurchases	Number of purchases made through the company's website.
numcatalogpurchases	Number of purchases made through catalogs.
numwebvisitsmonth	Number of visits to the company's website in a month.
numstorepurchases	Number of purchases made in physical stores.

acceptedcmp3	Whether the customer accepted marketing campaign 3 (1 for accepted, 0 for not accepted).
acceptedcmp4	Whether the customer accepted marketing campaign 4 (1 for accepted, 0 for not accepted).
acceptedcmp5	Whether the customer accepted marketing campaign 5 (1 for accepted, 0 for not accepted).
acceptedcmp1	Whether the customer accepted marketing campaign 1 (1 for accepted, 0 for not accepted).
acceptedcmp2	Whether the customer accepted marketing campaign 2 (1 for accepted, 0 for not accepted).
complain	Whether the customer has filed a complaint (1 for yes, 0 for no).
z_costcontact	An internal variable related to the cost of contacting the customer.
response	The customer's response to marketing campaigns (1 for responded, 0 for not responded).

5.2. Data Ingestion

5.2.1. Introduction

Data ingestion is a crucial step in the data processing pipeline. It involves transferring data from various sources, such as local CSV files, to a centralized database like PostgreSQL. In this section, we will discuss how data is ingested from local CSV files into the PostgreSQL database and the importance of this process.

5.2.2. Data Ingestion Process

To facilitate the data ingestion process, we have created two key functions using the SQL Alchemy Python library and ingested the CSV format data into the Postgres database:

a. Function for Uploading Data to PostgreSQL

This function is responsible for taking a DataFrame containing data from local CSV files and transferring it to the PostgreSQL database. The SQLAlchemy library provides a robust and efficient way to perform this operation. The function establishes a connection to the PostgreSQL database, maps the DataFrame to the database table, and inserts the data.

b.Function for Retrieving Data from PostgreSQL

After data ingestion, it is often necessary to retrieve data from the PostgreSQL database for analysis or further processing. We have implemented a function to fetch data from the database and return it as a DataFrame.

5.3. Data Transformation

3.2.1 Data Transformation Overview

The dataset underwent a series of transformations to prepare it for analysis and modeling. These transformations include data type conversion, feature engineering, data cleaning, and encoding. The transformations are outlined below:

3.2.2 Date Conversion

The 'dt_customer' column, which represents the date when a customer became a client, was converted from a string format ('%d-%m-%Y') to a datetime format for date-based calculations.

3.2.3 Feature Engineering

- 'customer_for' column: Calculates customer tenure in months based on 'dt_customer'.
- 'Age' column: Computes age by subtracting birth year from the current year.
- 'Spent' column: Generates total spending across product categories.
- 'living_with' column: Simplifies marital status, mapping 'Married'/'Together' to 'Partner'.
- 'children' column: Calculates total children by summing 'kidhome' and 'teenhome'.
- 'Family_Size' column: Represents family size as 1 for 'Single' and 2 for 'Partner' households.
- 'Is_Parent' column: Identifies if a customer is a parent based on children presence.
- 'education' column: Standardizes education levels as 'Undergraduate', 'Graduate', or 'Postgraduate'.
- 'AgeGroup' column: Categorizes customers into 'Teen', 'Adult', 'Middle-Aged Adult', or 'Senior Adult' based on age.

3.2.4 Data Cleaning

Outliers were removed from the dataset by filtering out customers with an age exceeding 100 years or an income above $120,000.

Several unnecessary columns, including 'marital_status', 'dt_customer', 'z_costcontact', 'z_revenue', 'year_birth', 'id', 'AgeGroup', and 'living_with', were dropped from the dataset as they were no longer needed for analysis or modeling.

3.2.5 Encoding

Label encoding was applied to categorical columns using the LabelEncoder from the scikit-learn library. This encoding converts categorical values into numerical values for machine learning algorithms.

Following the aforementioned transformations, two tables were saved in the PostgreSQL database: one without encoding and the other after encoding, both for future use, and all of that process is done within the Data_process.py file.

5.4. Data Pre-processing

5.4.1. Introduction

In the data processing phase, we employ various techniques to prepare the data for analysis or modeling. This section discusses how we have implemented data processing using the Scikit-Learn library's pipeline. Specifically, the pipeline involves data imputation, scaling, and Principal Component Analysis (PCA).

5.4.2. Data Processing Pipeline

To streamline the data processing workflow, we've designed a pipeline that

incorporates thefollowingsteps.

a. Data Imputation

Data often contains missing values, which can hinder analysis or modeling. We use Scikit-Learn's imputation techniques to fill in missing data with appropriate values. This step ensures that our dataset is complete and ready for further processing.

b. Data Scaling:

Data may have varying scales, which can affect the performance of many machine learning algorithms. To mitigate this, we apply data scaling to bring all features to a similar scale. Scikit-Learn provides tools for the standardization or normalization of data.

c. Principal Component Analysis (PCA)

PCA is a dimensionality reduction technique that helps in reducing the number of features while preserving most of the data's variance. This step is particularly useful when dealing with high-dimensional data. Example:

5.4.3. Creating the Processing Pipeline

Now, we create a data processing pipeline by combining the above steps in the desired order. Each step in the pipeline is executed sequentially:

```
data_pipeline = Pipeline(
    steps=[
        ('imputer', SimpleImputer(strategy='median')),
        ('scaler', StandardScaler()),
        ('pca', PCA(n_components=3))
    ]
)
```

In the data transformation process, we selected five columns (Age, Spent, customer_for, children, income) for model training. We then saved the preprocessing pipeline as a .pkl file and returned the preprocessed DataFrame along with the saved preprocessing file path.

5.5. Model Building

5.5.1. KMeans++ Clustering

The model training process begins with KMeans++ clustering, a popular unsupervised machine learning technique. KMeans++ initializes cluster centroids in a way that improves convergence compared to the standard KMeans algorithm. This step groups data points into clusters based on their similarity.

5.5.2. Using KneeLocator for Cluster Determination

To determine the optimal number of clusters (k), we employ the KneeLocator algorithm. KneeLocator identifies the "elbow point" in the within-cluster sum of squares (WCSS) curve. This point represents an optimal balance between model complexity and performance.

5.5.3. Merging Clusters with the Data Frame

After clustering, we merge the cluster assignments back into the original DataFrame. This step adds a new column indicating the cluster to which each data point belongs. This information is valuable for downstream analysis or predictions.

5.5.4. Dividing Clusters into Income and Spending Groups

Post-prediction, the clusters are further divided into four groups based on income and spending. We use mapping to replace cluster labels with more descriptive group names: 'Platinum', 'Silver', 'Gold', and 'Bronze'.

```
df.replace({'Clusters': {3: 'Bronze', 0: 'Platinum', 1: 'Silver', 2: 'Gold'}},inplace=True)
```

5.5.5. Saving the Model as a PKL File

Once the optimal number of clusters is determined, we save the KMeans model as a .pkl file. This file format allows us to serialize and store the trained model for later use, ensuring reproducibility and ease of deployment.

5.6. Training Pipeline

5.6.1. Introduction

In the context of our data processing pipeline, we have developed an integrated training pipeline that encompasses data ingestion, data processing, data transformation, and model training. This section outlines the key steps and components of this training pipeline.

```
if __name__=='__main__':
    start_time = time.time()

    obj=DataIngestion()
    table_name=obj.initiate_data_ingestion()
    DataProcess=DataCleaning()
    cleantable_nam
    data_transform
    train_df,_=dat
    model_trainer=ModelTrainer()
    model_trainer.initiate_model_training(train_df,Without_encoding)
```

```
class ModelTrainer()
Full name: src.components.model_trainer.ModelTrainer
```

```
_dimensionality(table_name)
le_name)
```

5.6.2. Explanation:

5.6.2.1. Data Ingestion:

- The process begins with data ingestion, handled by the DataIngestion class.
- The data ingestion returns the table name in which data is stored that name stored into the variable table_name.

5.6.2.2. Data Processing and Dimensionality Reduction:

- Following data ingestion, the DataCleaning class is utilized for data cleaning and dimensionality reduction.
- This step produces a cleaned table named cleaned_table_name and an unencoded DataFrame called Without_encoding.

5.6.2.3. Data Transformation:

- Data transformation is the next step, performed using the DataTransformation class.
- The result is a transformed training DataFrame, train_df.

5.6.2.4. Model Training:

- Finally, the ModelTrainer class is employed to initiate the model training process.
- The processed data and unencoded DataFrame are used to train the machine learning model.

5.7. Prediction pipeline

5.7.1. Prediction Function: This function is responsible for making predictions using a machine learning model. It loads both the preprocessor (.pkl) file and the trained model. The input data is passed through the preprocessor to prepare it for prediction, and then the model is applied to generate predictions.

5.7.2. Data Conversion Function: This function is designed to take input data and convert it into a DataFrame format suitable for prediction. It ensures that the input data is properly formatted and processed before passing it to the prediction function.

5.7.3. Here is a high-level overview of how these functions work together:

- The user provides input data, which may come in various formats.
- The data conversion function processes the input data to ensure it is in the correct format for prediction and converts it into a DataFrame.
- The prediction function loads the preprocessor and the trained model.
- The input data, now in DataFrame format, is passed through the preprocessor to prepare it for prediction. This may include data cleaning, encoding, and any necessary transformations to match the format the model expects.
- The preprocessed data is then fed into the trained machine learning model, which generates predictions.
- By separating the data conversion and prediction functions, we ensure that input data is consistently and correctly prepared for prediction, making the prediction pipeline more robust and user-friendly.

5.8. Streamlit app

5.8.1. Folder Structure: Create a folder named "pages" to organize your app's different pages.

5.8.1.1. Page 1: Form Page (Home Page)

- This is the landing page of your app.
- Display a form where users can input their data.
- Include a submit button that, when clicked, triggers the prediction process.
- After submission, automatically redirect the user to the result page.

5.8.1.2. Page 2: Result Page

- Upon submission, users are redirected here to view the predicted results.
- Display the predicted results in a chat format to make it user-friendly.

5.8.1.3. Page 3: Charts Page (Analysis of Clusters)

- Provide visualizations and analysis of the clusters created during data preprocessing.
- Include charts and insights related to the clusters for better understanding.

5.8.1.4. Page 4: Make Your Own Chat Page

- Integrate the PyGWalk library to create a chat interface.
- Provide pre-existing chat data related to all the clusters.
- Allow users to interact with the chat data and make their own conversations.

5.8.2. Integration with Prediction Pipeline:

- In the Form Page, connect the input data to your prediction pipeline.
- Upon submission, pass the input data through the prediction pipeline to generate predictions.
- Display the predicted results in the Result Page.

5.8.3. Streamlit App Workflow:

- The user visits the Home Page (Form Page) and fills in the required data.
- Upon hitting the submit button, the app triggers the prediction process.
- The user is redirected to the Result Page, where the predicted
- results are displayed in a chat format.
  The user can navigate to the Charts Page to see cluster analysis and insights.
- The Make Your Own Chat Page allows users to interact with existing chat data and create their own conversations.

By organizing your Streamlit app in this manner, you provide a user-friendly interface for data input, prediction display, cluster analysis, and interactive chat functionality. This structure should help users easily navigate and interact with your app.

5.9. CI/CD Pipeline Integration with Docker, Amazon ECR, GitHub Actions, and AWS EC2

Overview:

In this section, we will elaborate on the integration of a CI/CD (Continuous Integration/Continuous Deployment) pipeline for our application. We've harnessed the capabilities of Docker, Amazon Elastic Container Registry (ECR), GitHub Actions, and AWS EC2 to automate our software development lifecycle.

5.9.1. Docker Build Integration:

5.9.1.1. Docker Configuration: Our CI/CD pipeline leverages Docker, allowing us to encapsulate our application and its dependencies into a container image.

5.9.1.2. Dockerfile: Within our project, a Dockerfile is meticulously crafted. This file outlines the precise steps necessary to construct the Docker image. It specifies the base image, necessary dependencies, and the setup of our application.

5.9.1.3. Docker Build Step: A pivotal step within our CI workflow involves the execution of the docker build command. This command constructs the Docker image in accordance with the instructions detailed in the Dockerfile.

5.9.1.4. Image Tagging: To facilitate version management, we've implemented a tagging mechanism for our Docker images. This involves labeling images with version numbers or commit hashes, enabling us to track different iterations of our application.

5.9.2. Amazon Elastic Container Registry (ECR):

5.9.2.1. Private Image Storage: Amazon ECR serves as our preferred private Docker image registry. It provides a secure and scalable solution for the storage of Docker container images.

5.9.2.2. Authentication: To ensure the secure push of Docker images from our CI/CD pipeline to the designated ECR repository, we've configured robust authentication mechanisms. This typically entails the use of AWS credentials and IAM (Identity and Access Management) roles.

5.9.2.3. Image Push: After the successful construction of the Docker image, our CI/CD pipeline seamlessly pushes the image to the designated ECR repository, keeping our images safe and accessible.

5.9.3. GitHub Actions and AWS EC2 Integration:

5.9.3.1. GitHub Actions Workflow: Our CI/CD pipeline is realized through GitHub Actions. This feature-rich CI/CD framework operates directly within our GitHub repository, streamlining the automation of key workflows, such as build, test, and deployment. We've leveraged YAML configuration files to define these workflows explicitly.

5.9.3.2. AWS EC2 Instance: As our chosen deployment target, we've established an AWS EC2 instance to serve as a robust and scalable hosting environment for our application.

5.9.3.3 Deployment Step: A fundamental component of our CD workflow is the deployment step. This step efficiently deploys our Docker container onto the AWS EC2 instance. To achieve this, we've employed a self-hosted runner configured with a YAML file. This runner pulls the Docker image, performs necessary configurations specified in the Dockerfile, and deploys the application on the EC2 instance. This process may also involve pulling the Docker image from the Amazon ECR repository, ensuring a seamless deployment.

5.9.3.4. AWS Integration: To ensure secure communication between GitHub Actions and the AWS EC2 instance, we've meticulously configured AWS services. This includes the setup of IAM roles and security groups, maintaining the integrity and confidentiality of our CI/CD pipeline. The IAM roles are utilized to grant specific permissions for the self-hosted runner to interact with AWS services securely. Security groups are defined to control incoming and outgoing traffic, enhancing the overall security posture of our deployment process.

This comprehensive approach ensures that our CI/CD pipeline is orchestrated effectively, utilizing YAML configuration files for defining workflows and incorporating self-hosted runners to streamline Docker image handling and application deployment on the AWS EC2 instance. The AWS integration further enhances the security and reliability of our CI/CD pipeline.

5.10. Deployment

In the deployment phase, we've hosted the entire website on an Amazon Elastic Compute Cloud (EC2) instance. This EC2 instance serves as the hosting environment for our website, making it accessible to users over the internet. This setup allows us to deploy and run our web application in a scalable and reliable manner on the AWS infrastructure.