Architecture

Back Order Prediction

Document Version:1.0

Control Version Date: 22-Feb-24

Author : Akshay Kumar BM

Comments: Introduction & Architecture Defined

Context

Architecture Document

1. Introduction

1.1. What is Low-Level Design Document?

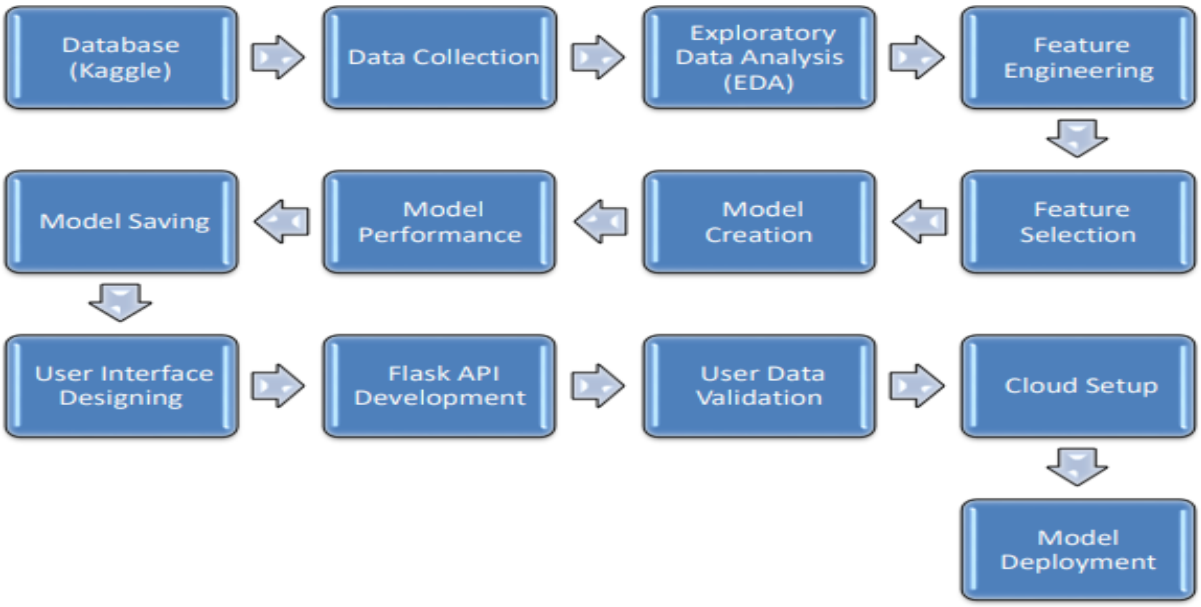
A low-level design document (LLD) is a detailed blueprint of a software product, which outlines the specific components of a system and their interactions.

1.2. Scope

This document covers the architecture of a machine learning model for predicting backorders in inventory management.

2. Architecture

The architecture consists of several components, each responsible for a specific aspect of the system, such as data ingestion, data transformation, model training, and prediction.



3. Architecture Description

3.1. Database (Kaggle)

The data used for this project is sourced from Kaggle. It contains various features related to product orders.

3.2. Data Collection

The `DataIngestion` class handles data collection. It reads the data from CSV files and returns the paths of the train and test data.

3.3. Exploratory Data Analysis (EDA)

EDA is performed to understand the underlying patterns of the data, identify outliers, and understand the relationship between different variables.

3.4. Feature Engineering

The `DataTransformation` class handles feature engineering. It includes handling missing values, scaling numerical features, and encoding categorical features.

3.5. Feature Selection

Relevant features are selected for the model. This includes ‘national_inv’, ‘lead_time’, ‘in_transit_qty’, ‘forecast_3_month’, ‘sales_1_month’, ‘min_bank’, and ‘perf_6_month_avg’.

3.6. Model Creation

The `ModelTrainer` class handles model creation. It includes the training of different models and evaluation based on the F1 score.

3.7. Model Performance

The performance of the model is evaluated using the F1 score. The model with the highest F1 score is selected as the best model.

3.8. Model Saving

The best model is saved using the `pickle` module for future use.

3.9. User Interface Designing

A user interface is designed to allow users to input the required features and get the prediction results.

3.10. Flask API Development

A Flask API is developed to serve the model predictions. It takes the user input from the user interface, passes it to the model, and returns the prediction.

3.11. User Data Validation

The `CustomData` class handles user data validation. It ensures that the user input is in the correct format before it is passed to the model.

3.12. Cloud Setup

The application is set up on a cloud server to allow users to access the model predictions remotely.

3.13. Model Deployment

The model is deployed on the cloud server. The Flask API serving the model predictions is also hosted on the server.

4. Low-Level Documentation

This project is structured into several Python scripts, each serving a specific purpose:

logger.py: This script sets up the logging configuration. It creates a log file with a timestamp in its name and stores it in a directory named “logs”. The logging level is set to INFO.

utils.py: This script contains utility functions:

save_object(file_path, obj): Saves a Python object to a file using pickle.

load_object(file_path): Loads a Python object from a file using pickle.

outlier_remover(df): Removes outliers from a dataframe based on the Z-score.

target_column_Encoding(df): Encodes the target column “went_on_backorder” from Yes/No to 1/0.

exception.py: This script defines a custom exception class CustomException for handling exceptions throughout the project. It logs the file name, line number, and error message when an exception occurs.

data_ingestion.py: This script is responsible for ingesting the data from CSV files. It reads the train and test data from the paths specified in DataIngestionConfig and returns these paths.

data_transformation.py: This script is responsible for preprocessing the data. It handles missing values, scales numerical features, encodes categorical features, balances the data, and extracts features. It saves the preprocessor object to a file and returns the transformed train and test data along with the path to the preprocessor object.

model_trainer.py: This script is responsible for training the model. It evaluates multiple models (Random Forest, Decision Tree, SGD, K-

Nearest Neighbors, and Gradient Boosting) and selects the best one based on F1 score. It saves the best model to a file.

prediction_pipeline.py: This script is responsible for making predictions on new data using the trained model. It loads the preprocessor and model objects from their respective files, scales the features using the preprocessor, and makes predictions using the model.

training_pipeline.py: This script initiates the training pipeline which includes data ingestion, data transformation, and model training.

To use this project, you need to create an instance of the Training_Pipeline class and call the initiate_training_pipeline method to train the model. To make predictions on new data, create an instance of the CustomData class with the feature values, convert it to a dataframe, create an instance of the PredictPipeline class, and call the predict method.

Please note that this project is a basic implementation and can be further improved by fine-tuning the models, adding more features, or using more advanced models. Always validate the model with new data to ensure its effectiveness.

Please handle any exceptions that may occur during the execution of the scripts. The CustomException class can be used for this purpose.