Explanation of Ride Pricing Model Implementation

1 Overview

This document provides a detailed explanation of a Python-based data processing and modeling pipeline for predicting ride durations and generating dynamic pricing coefficients for a ride-hailing service.

2 Library Imports

```
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import RidgeCV, LassoCV, ElasticNetCV
from datetime import timedelta
from sqlalchemy import create_engine
from dotenv import load_dotenv
import os
import urllib
```

We import necessary libraries:

- `pandas`, `numpy`: for data manipulation and analysis.
- `sklearn.linear_model`: for fitting various regression models.
- `datetime.timedelta`: for time window calculations.
- `sqlalchemy`, `urllib`: for database connection and query execution.
- `dotenv`, `os`: for secure loading of database credentials from an `.env` file.

3 Environment and Database Configuration

```
load_dotenv()
DB_HOST = os.getenv('SQLSERVER_HOST')
DB_PORT = os.getenv('SQLSERVER_PORT', '1433')
DB_NAME = os.getenv('SQLSERVER_DB')
DB_USER = os.getenv('SQLSERVER_USER')
DB_PASSWORD = os.getenv('SQLSERVER_PASSWORD')
```

This section loads environment variables for secure database credentials.

```
odbc_str = (
    f"DRIVER=ODBC␣Driver␣17␣for␣SQL␣Server;"
    ...
)
connection_uri = f"mssql+pyodbc:///?odbc_connect={urllib.parse.quote_plus(
   odbc_str)}"
engine = create_engine(connection_uri)
```

An ODBC connection string is constructed and parsed via `sqlalchemy`'s `create_engine`.

4 Data Extraction

```
df_features = pd.read_sql("SELECT␣*␣FROM␣ride_pricing_features", con=engine)
...
```

Four datasets are loaded from the SQL Server:

- `ride_pricing_features`
- `ride_requests`
- `ride_completion_delay`
- `ride_acceptance_delay`

5 Data Cleaning

Column names are cleaned to remove extraneous quotes:

```
df.columns = df.columns.str.strip().str.replace(’""’, ’’).str.replace(’"’, ’’)
```

Comma-based decimal separators are converted to dots:

```
df[col] = df[col].astype(str).str.replace(’,’, ’.’)
df[col] = pd.to_numeric(df[col])
```

6 Datetime Parsing

Relevant datetime fields across all datasets are parsed using:

```
pd.to_datetime(..., format=’%Y-%m-%d␣%H:%M:%S.%f’)
```

7 Hourly Statistics Computation

Function: `compute_hourly_stats`

This function computes per-hour historical statistics from the previous week's data for:

- Request counts and their standard deviation
- Average and standard deviation of acceptance times
- Ride durations
- Z-scores for outlier detection

Filtering is done using precise datetime ranges based on each ride's request hour:

```
prev_week_start = base_date - timedelta(weeks=1) + timedelta(hours=lag_hour)
...
df_requests['request_time'].dt.hour.between(lag_hour, lead_hour)
```

Z-score Calculation

Z-scores are computed to assess how far current metrics deviate from weekly averages:

```
z = (current - mean) / std_dev
```

8 Merging and Filtering

Computed statistics are merged back to the features using:

```
df_features.merge(stats_df, on='ride_id', how='left')
```

Rows with only zero statistics are filtered out:

```
df_merged = df_merged[df_merged[stat_columns].ne(0).any(axis=1)]
```

9 Modeling

Feature Selection

The model uses the following features:

- `distance_km`
- `current_accept_time`
- `request_count`, `avg_accept_time`, `avg_ride_duration`
- `z_request_count`, `z_accept_time`

```
X = df_merged[feature_cols]
y = df_merged['duration_min']
```

Model Fitting

Four types of regression models are trained in this project. Each of these models serve the goal of predicting a continuous target variable based on input features, but they differ in how they handle regularization, complexity, and overfitting.

- **Linear Regression**
  Linear Regression is the most basic form of regression, where the model attempts to find a linear relationship between the independent variables (features) and the dependent variable (target). It estimates coefficients that minimize the residual sum of squares between the observed and predicted values. This model assumes no multicollinearity and is sensitive to outliers and overfitting in high-dimensional data.
- **Ridge Regression (cross-validated)**
  Ridge Regression is a regularized version of Linear Regression that adds an L2 penalty term to the loss function. This penalty term shrinks the coefficients toward zero but does not eliminate them entirely. The purpose of Ridge is to prevent overfitting, especially when dealing with multicollinearity or when the number of predictors exceeds the number of observations. Cross-validation is used to find the optimal regularization parameter (alpha), ensuring the model generalizes well to unseen data.
- **Lasso Regression (cross-validated)**
  Lasso Regression (Least Absolute Shrinkage and Selection Operator) adds an L1 penalty to the loss function, which can drive some coefficients exactly to zero. This results in a sparse model that performs feature selection by eliminating less important predictors. Lasso is useful when the goal is not only prediction but also interpretability and identifying the most relevant features. Cross-validation is used to select the best alpha for minimizing prediction error.
- **ElasticNet Regression (cross-validated)**
  ElasticNet combines both L1 (Lasso) and L2 (Ridge) penalties in its loss function. It balances the benefits of both methods: the feature selection capability of Lasso and the stability of Ridge. ElasticNet is particularly effective when there are multiple correlated features or when neither Lasso nor Ridge alone yields satisfactory results. The mixing parameter (usually denoted by ρ or α) and the overall regularization strength are both tuned using cross-validation to achieve optimal performance.

10 Generating Pricing Coefficients

Mapping Features to Pricing Parameters

```
feature_to_pricing = {
    'distance_km': 'baseRate',
    'request_count': 'coeffRequests',
    ...
}
```

Non-distance features are normalized:

```
norm_coef = abs(coef) / total_sum
```

A special variable `stdDevFactor` is computed as the mean of normalized z-score-based features.

Stored Procedure Code Generation

Output is formatted as:

```
DECLARE @coeffRequests FLOAT = 0.23;
...
DECLARE @stdDevFactor FLOAT = 0.14;
```

This prepares parameters for integration into a stored procedure like `CalculateDynamicPrice`.

11 Conclusion

The script integrates database access, data cleaning, feature engineering, statistical profiling, model training, and parameter generation for a dynamic pricing system. It enables data-driven decision-making by linking operational metrics (like ride duration and acceptance delay) to a predictive model for fair and efficient pricing.