Explanation of Ride Pricing Model Implementation

0.1 Overview

This document describes a Python-based pipeline for processing ride-sharing data and modeling ride prices to derive coefficients for a dynamic pricing algorithm. It improves upon an earlier implementation (`linearRegression.py`) that predicted ride durations using uniformly distributed data from `generate_ride_data.p` which did not reflect real-world congestion patterns. The updated pipeline, implemented in `PriceTargetLinearRegression.py`, uses surge-based data from `SurgePricingDataGeneration.py` with demand peaks at 8 AM and 5 PM, targeting ride prices in South African Rand (ZAR) to support dynamic pricing.

0.2 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
```

The pipeline uses:

- `pandas`, `numpy`: For data manipulation and numerical computations.
- `sklearn.linear_model`: For fitting regression models to predict prices.
- `datetime.timedelta`: For calculating time windows in hourly statistics.
- `sqlalchemy`, `urllib`: For SQL Server database connectivity.
- `dotenv`, `os`: For secure loading of database credentials from a `.env` file.

0.3 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')
odbc_str = (
    f"DRIVER=ODBC␣Driver␣17␣for␣SQL␣Server;"
    f"SERVER={DB_HOST},{DB_PORT};"
    f"DATABASE={DB_NAME};"
    f"UID={DB_USER};"
```

```
    f"PWD={DB_PASSWORD};"
    "TrustServerCertificate=yes;"
    "Connection Timeout=30;"
    "Command Timeout=60;"
)
connection_uri = f"mssql+pyodbc:///?odbc_connect={urllib.parse.
   quote_plus(odbc_str)}"
engine = create_engine(
    connection_uri,
    pool_timeout=30,
    pool_recycle=300,
    pool_pre_ping=True
)
```

Environment variables are loaded from a .env file for secure database credentials. An ODBC connection string with timeout settings ensures robust connectivity to a SQL Server database via sqlalchemy's create_engine.

0.4 Data Extraction

```
df_requests = pd.read_sql("SELECT * FROM ride_requests", con=engine)
df_completion = pd.read_sql("SELECT * FROM ride_completion_delay",
   con=engine)
df_acceptance = pd.read_sql("SELECT * FROM ride_acceptance_delay",
   con=engine)
df_matches = pd.read_sql("SELECT * FROM ride_matches", con=engine)
```

Four datasets are extracted from SQL Server:

- ride_requests: Contains ride request details (e.g., ride_id, request_time, pickup_lat).
- ride_acceptance_delay: Records driver acceptance times and delays.
- ride_completion_delay: Includes ride durations and completion times.
- ride_matches: Stores ride matches with price and status information.

Unlike the previous version, which used a precomputed ride_pricing_features table, this pipeline processes raw ride data directly, enabling dynamic feature generation.

0.5 Data Cleaning

```
for df in [df_requests, df_completion, df_acceptance, df_matches]:
    df.columns = df.columns.str.strip().str.replace('"""', '').str.
       replace('"', '')
    for col in df.columns:
        if df[col].dtype == 'object':
            df[col] = df[col].astype(str).str.replace(',', '.')
```

```python
            try:
                df[col] = pd.to_numeric(df[col])
            except:
                pass
```

Column names are cleaned to remove extraneous quotes, and comma-based decimal separators are converted to dots to ensure numerical consistency.

0.6 Datetime Parsing

```python
datetime_cols = {
    'request_time': [df_requests, df_acceptance],
    'pickup_time': [df_completion],
    'accepted_time': [df_acceptance],
    'completed_at': [df_completion, df_matches],
    'matched_at': [df_matches],
    'driver_response_at': [df_matches],
    'started_at': [df_matches],
    'arrived_at': [df_matches]
}
for col_name, dataframes in datetime_cols.items():
    for df in dataframes:
        if col_name in df.columns:
            df[col_name] = pd.to_datetime(df[col_name], format='%Y-%m-%d %H:%M:%S.%f', errors='coerce')
```

Datetime fields are parsed with a specific format to ensure accurate temporal analysis, with invalid entries coerced to handle errors.

0.7 Feature Engineering

The pipeline constructs a feature dataset from `ride_matches`, adding:

- **Distance**: Euclidean distance (`distance_km`) calculated as:

$$\text{distance\_km} = \sqrt{(\text{dropoff\_lat} - \text{pickup\_lat})^2 + (\text{dropoff\_lng} - \text{pickup\_lng})^2} \times 111 \tag{1}$$

- **Temporal Features**: `request_hour` and `request_day` extracted from `matched_at`.
- **Acceptance Time**: `current_accept_time` merged from `ride_acceptance_delay`, with missing values filled by the median.

0.8 Hourly Statistics Computation

```python
def compute_hourly_stats(df_features, df_requests, df_acceptance, df_completion):
    stats = []
    base_date = (df_features['matched_at'].iloc[0].floor('D')
```

```
                if not df_features['matched_at'].isna().all()
                else pd.to_datetime('2025-01-01'))
    for idx, row in df_features.iterrows():
        ride_id = row['ride_id']
        hour = row['request_hour']
        lag_hour = max(0, hour - 1)
        lead_hour = min(23, hour + 1)
        prev_week_start = base_date - timedelta(weeks=1) + timedelta
           (hours=lag_hour)
        prev_week_end = base_date - timedelta(weeks=1) + timedelta(
           hours=lead_hour + 1)
        mask_requests = (
            (df_requests['request_time'] >= prev_week_start) &
            (df_requests['request_time'] < prev_week_end) &
            (df_requests['request_time'].dt.hour.between(lag_hour,
               lead_hour))
        )
        request_count = df_requests[mask_requests].shape[0]
        hourly_counts = df_requests[mask_requests].groupby(
           df_requests['request_time'].dt.hour).size()
        avg_request_count = hourly_counts.mean() if not
           hourly_counts.empty else 0
        stdev_request_count = hourly_counts.std() if len(
           hourly_counts) > 1 else 0
        # ... (similar for acceptance and completion)
        z_request_count = (request_count - avg_request_count) /
           stdev_request_count if stdev_request_count > 0 else 0
        z_accept_time = (row['current_accept_time'] -
           avg_accept_time) / stdev_accept_time if stdev_accept_time
            > 0 else 0
        stats.append({
            'ride_id': ride_id,
            'request_count': request_count,
            'avg_request_count': avg_request_count,
            'stdev_request_count': stdev_request_count,
            'avg_accept_time': avg_accept_time,
            'stdev_accept_time': stdev_accept_time,
            'avg_ride_duration': avg_ride_duration,
            'stdev_ride_duration': stdev_ride_duration,
            'z_request_count': z_request_count,
            'z_accept_time': z_accept_time
        })
    return pd.DataFrame(stats)
```

The `compute_hourly_stats` function calculates historical statistics for the previous week's data within a 3-hour window (current hour $\pm$ 1), including:

- Request counts, averages, and standard deviations.
- Average and standard deviation of acceptance times and ride durations.

- Z-scores for request counts and acceptance times, calculated as:

$$z = \frac{\text{current} - \text{mean}}{\text{std\_dev}} \tag{2}$$

Rows with zero statistics are filtered out to ensure meaningful data.

0.9 Merging and Filtering

Computed statistics are merged with the feature dataset:

```
df_merged = df_features.merge(stats_df, on='ride_id', how='left')
df_merged = df_merged[df_merged[stat_columns].ne(0).any(axis=1)]
```

This ensures only rows with non-zero statistics are used for modeling.

0.10 Model Fitting

Four regression models are trained to predict ride prices (ZAR), replacing the previous duration-based approach:

- **Linear Regression**: Fits a linear model, minimizing squared errors. Sensitive to outliers but interpretable.
- **Ridge Regression**: Uses L2 regularization to mitigate overfitting, with cross-validated alpha selection.
- **Lasso Regression**: Applies L1 regularization for feature selection, potentially zeroing coefficients, with cross-validated alpha.
- **ElasticNet Regression**: Combines L1 and L2 penalties for balanced regularization, with cross-validated parameters.

Features used:

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

Models are evaluated using Mean Squared Error (MSE) and R^2 scores, with coefficients interpreted for pricing impact (e.g., price increase per kilometer).

0.11 Generating Pricing Coefficients

```
feature_to_pricing = {
    'distance_km': 'baseRate',
    'request_count': 'coeffRequests',
    'avg_accept_time': 'coeffAcceptTime',
```

```
    'avg_ride_duration': 'coeffRideDuration',
    'z_request_count': 'stdDevFactor',
    'z_accept_time': 'stdDevFactor'
}
for name, coefs in model_results.items():
    included = [abs(c) for f, c in coefs.items() if f != 'distance_km']
    total = sum(included) or 1
    std_dev_features = ['z_request_count', 'z_accept_time']
    std_dev_sum = sum(abs(coefs[f]) for f in std_dev_features) / len(std_dev_features) if std_dev_features else 0
    for f in feature_cols:
        key = feature_to_pricing.get(f)
        if key == 'baseRate':
            print(f"DECLARE @{key} FLOAT = {linear_model.intercept_:.2f};  -- Base price from intercept")
        elif key == 'stdDevFactor':
            continue
        else:
            coef_val = coefs[f]
            print(f"DECLARE @{key} FLOAT = {coef_val:.4f};")
    if std_dev_sum > 0:
        print(f"DECLARE @stdDevFactor FLOAT = {std_dev_sum:.4f};")
```

Features are mapped to pricing parameters, with non-distance coefficients normalized:

$$\text{norm\_coef} = \frac{|\text{coef}|}{\text{total\_sum}} \tag{3}$$

The `stdDevFactor` is computed as the mean of normalized z-score coefficients. Output is formatted for SQL stored procedures, e.g.:

```
DECLARE @coeffRequests FLOAT = 0.2300;
DECLARE @stdDevFactor FLOAT = 0.1400;
```

0.12 Data Generation

The pipeline uses surge-based data from `SurgePricingDataGeneration.py`, addressing the limitations of `generate_ride_data.py`'s uniform distribution. Key features:

- **Demand Modeling**: Simulates demand peaks at 8 AM and 5 PM using Gaussian-like functions:

$$\text{demand} = 10 + 30\exp\left(-\frac{(\text{hours}-8)^2}{10}\right) + 40\exp\left(-\frac{(\text{hours}-17)^2}{12}\right) + \text{random noise} \tag{4}$$

- **Surge Pricing**: Calculates prices based on a demand-to-supply ratio, clipped between 1x and 3x, applied to a base ride price.

- **Variable Rides**: Scales ride volume by demand, with surge-dependent acceptance delays.

Data is generated for a single day, centered in Sandton, Gauteng, South Africa.

0.13 Conclusion

The updated pipeline leverages surge-based data, dynamic feature engineering, and price prediction to generate coefficients for a dynamic pricing algorithm. By addressing the uniform distribution limitation, it produces realistic, data-driven pricing coefficients suitable for integration into a stored procedure like `CalculateDynamicPrice`.