Car Price Prediction System

Module Documentation

Sandeep Singh Naruka

April 6, 2025

Contents

Module 1: Config Directory

1.1 config.py

This module defines global configuration constants and paths used across the project for machine learning model inference and data preprocessing. It also handles dynamic path resolution to ensure compatibility across environments.

- `CATEGORICAL_COLUMNS`: List of all categorical features in the dataset.
- `NUMERICAL_COLUMNS`: List of all numerical features in the dataset.
- `ENCODING_PATHS`: Dictionary mapping each categorical column to its respective encoding dictionary JSON path.
- `NOMINAL_COLUMNS`: List of nominal categorical columns (unordered categories).
- `ORDINAL_COLUMNS`: List of ordinal categorical columns (ordered categories).
- `TARGET_COLUMN`: The column name representing the target variable (car price).
- `Q_MATRIX`: Absolute path to the Q-matrix (used for feature transformations or quality scores).
- `X_SCALER`, `Y_SCALER`: Absolute paths to pre-fitted scalers used for transforming input and output data.
- `RELEVENT_COLUMNS_IDX`: Path to the relevant feature indices used during model training.
- `MODEL`: Path to the serialized model weights in JSON format.

Module 2: backend

2.1 Overview

The `backend` directory serves as the core of the application's business logic. It contains authentication mechanisms, caching utilities, API routes, data schemas, asynchronous task definitions, and services related to the car price prediction system.

2.2 Submodules

This module includes the following submodules:

- **auth/**: Handles JWT-based authentication.
- **cache/**: Manages Redis caching utilities.
- **routes/**: Contains all route handlers for APIs (upload, task handling, etc.).
- **schemas/**: Defines Pydantic schemas for data validation.
- **services/**: (Empty for now) Intended to hold business logic layers (e.g., service classes).
- **tasks/**: Manages Celery-based asynchronous task modules.

2.3 Dependencies

- `fastapi`, `pydantic`, `redis`, `jwt`, `celery`
- Project-specific modules: `db.py`, `redis_client.py`, `extensions.py`, etc.

2.4 Descriptions of Submodules

2.4.1 auth/jwt_utils.py

- Provides JWT token creation, verification, and decoding.
- Secures routes using bearer token authentication.

2.4.2 cache/redis_utils.py

- Offers Redis-based caching for rate limiting and data retrieval.
- Common for caching expensive operations or enforcing limits.

2.4.3 routes/task_routes.py

- Routes for task management: creation, deletion, listing.
- Performs role-based checks and logs tasks.

2.4.4 routes/upload.py

- Handles CSV file upload and processing.
- Uses Pydantic for input validation and integrates Redis caching.

2.4.5 schemas/task_schemas.py

- Defines Pydantic models for task creation and retrieval.
- Provides serialization/deserialization layer for API requests/responses.

2.4.6 services/

- Placeholder for service-layer abstractions.
- Currently unused but aligns with clean architecture practices.

2.4.7 tasks/__init__.py

- Contains Celery periodic/dynamic tasks.
- Transfers active task logs to persistent storage on schedule.

2.5 Security and Performance Notes

- JWT-based authentication is used to secure endpoints.
- Redis caching is used for rate limiting and improving speed.
- Celery allows for scalable background job processing.

2.6 Module: `jwt_utils.py`

Path: `backend/auth/jwt_utils.py`

Summary:

Provides JWT-based authentication utilities for securing Flask routes and retrieving user context.

Functions:

- `token_required(f)`: Decorator that checks JWT from the request header, decodes it, and stores the user in Flask's `g`.
- `get_current_user()`: Fetches the current user from DB using `get_jwt_identity()`.

Note: Uses raw JWT decoding and a hardcoded secret key. In production, store keys securely.

2.7 Module: task_routes.py

Path: backend/routes/task_routes.py

Summary:

Defines authenticated API endpoints to create and update tasks using JWT and Pydantic validation.

Blueprints:

- routes: Default blueprint (unused in current code).
- task_bp: API blueprint with prefix /api.

Endpoints:

- PUT /api/task/<id>: Updates a task. Only allowed by the task creator or an admin. Uses JWT to identify the user.
- POST /api/task: Creates a task. Requires JWT and validates input using TaskCreateSchema.

Security:

- Uses @token_required to enforce authentication.
- Role-based update control enforced via comparison of user.id and task.user_id.

2.8 Module: upload.py

This module defines the CSV upload endpoint for task data ingestion.

- **Route:** /upload-csv
  Allows bulk task creation through a CSV upload. Each row is validated using TaskCSVSchema. Valid tasks are inserted into the TaskManager table. Invalid rows are reported back in the response.
- **Key Dependencies:**
  - pandas for reading and processing the CSV file.
  - TaskCSVSchema for row-level validation.
  - SQLAlchemy SessionLocal for database operations.
- **Temporary Assumption:** All tasks are assigned to user ID 1 (typically an admin or manager).

2.9 Module: backend/schemas/task_schemas.py

This module contains request validation schemas using `Pydantic` for task operations.

- `TaskCSVSchema`
  Used for validating CSV-imported tasks. Requires a `title`; `description` is optional.
- `TaskCreateSchema`
  Used when creating a task via API. Validates that the title has at least 3 characters, description at least 5 characters, and status is one of: `pending`, `in_progress`, `completed`.

Note: Pydantic provides automatic type checking and error reporting for incoming data.

2.10 Module: backend/tasks/__init__.py

This module initializes the Celery worker and defines background tasks.

- `celery`
  Celery instance configured with Redis as broker and backend.
- `transfer_active_tasks()`
  A scheduled Celery task that logs all "active" tasks from the `TaskManager` table to the `TaskLogger` table, once per day. It ensures no duplicates are created for the same task on the same day.

Purpose: Enables asynchronous, daily task tracking for auditing.

2.11 app.py - Flask App Entry Point

- Sets up the Flask application and connects to PostgreSQL.
- Registers blueprints for modular routes.
- Implements car price prediction from user input.
- Handles task logging, updates, and CSV uploads.
- Uses Redis for caching, and Flask-Limiter for rate-limiting.

2.12 Celery Configuration(celery*worker.py*)

- Initializes the Celery app for distributed task processing.
- Configures Redis as both the message broker and result backend.
- Enables UTC timezone and schedules a daily recurring task.
- The scheduled task `transfer_active_tasks` runs every day at midnight.

2.13 Database Models

- **Car**: Stores raw vehicle data used for predictions and analysis.
  - `id`: Primary key.
  - `year`, `manufacturer`, `model`, `condition`, etc.: Vehicle features.
- **RoleEnum**: Enumeration for role-based access control (RBAC).
  - Values: `admin`, `manager`, `user`.
- **User**: Represents an application user with a unique username and a role.
  - `id`, `username`, `role`.
  - One-to-many relationship with `TaskManager`.
- **TaskManager**: Tracks user-created tasks.
  - Fields include `id`, `title`, `description`, `status`, timestamps.
  - Linked to `User` via `user_id`.
  - One-to-many relationship with `TaskLogger`.
- **TaskLogger**: Audit log table recording task status updates.
  - Fields: `id`, `task_id`, `old_status`, `new_status`, `changed_by`, `changed_at`.
  - Linked to `TaskManager`.

Database Connection Setup

- **Environment Variables**:
  - Loaded via `dotenv` to configure database access.
  - Supports both Docker-style `DATABASE_URL` or individual components (`DB_USER`, `DB_PASSWORD`, etc.).
- **Connection String Construction**:
  - If `DATABASE_URL` is defined, it is used directly.
  - Otherwise, a connection string is constructed using the individual environment variables.
- **create_engine_with_retry** (function):
  - Implements retry logic with:
    - `retries=5`, `delay=2s` between retries.
    - Connection pooling: `pool_size=10`, `max_overflow=20`.
    - Connection recycling: `pool_recycle=1800` (30 minutes).

  - Raises an exception after repeated failures.
  - Prints connection success/failure info.

- **engine**:
  - SQLAlchemy Engine object, initialized using the retry-enabled function.

- **SessionLocal**:
  - A configured SQLAlchemy session maker bound to the engine.
  - Used throughout the app to interact with the database.

- **Base**:
  - Declarative base class used for defining ORM models.

- **Main Test Block**:
  - If the script is run directly, it attempts a test query: `SELECT version();`.
  - Verifies connection and prints PostgreSQL version.

2.14 Database Seeding Script

The following Python script `seed_data.py` is used to populate the database with sample data for development and testing. It performs the following actions:

- Establishes a database session using `SessionLocal`.
- Creates three users with predefined roles: `admin`, `manager`, and `user`.
- Commits the users to the database.
- Creates a sample task titled `"Fix Bug"` with description `"Fix issue #231"` and assigns it to the manager.
- Commits the task to the database.
- Prints a confirmation message and closes the session.

The function `seed_data()` encapsulates this process and is executed when the script is run as the main module.

2.15 Summary

The `backend` module is the central logic layer of the application. It manages APIs, tasks, authentication, and data validation. Its modular structure makes the app scalable, maintainable, and secure.

Module 3: Data Transformation Directory

3.1 data_transformation.py

This module provides functionality to transform raw user input into a preprocessed NumPy array that is compatible with the machine learning model used for car price prediction.

Key Functionality

- Reads user input (as a dictionary) and converts it to a Pandas DataFrame.
- Encodes categorical columns using predefined encoding dictionaries stored in JSON files.
- Applies one-hot encoding to nominal categorical variables.
- Normalizes the year column relative to the first car invention (1886).
- Keeps only relevant columns as determined during model training.
- Applies PCA using a Q-matrix to reduce dimensionality.
- Scales the processed data using pre-fitted scalers.

Function: `process_input(input_data)`

- **Parameters:** `input_data` — A Python dictionary containing raw user input values.
- **Returns:** A NumPy array ready for input into the ML model.
- **Pipeline:**
  1. Converts input dictionary to DataFrame.
  2. Applies encoding to categorical variables using `utils.encode()`.
  3. Performs one-hot encoding using `utils.convert_to_ohe_input()`.
  4. Drops original nominal columns.
  5. Applies normalization and PCA via `utils.apply_pca_by_Q_mat()`.
  6. Scales final features using `utils.scale()`.

Module 4: Models Directory

4.1 model.py

This module performs prediction using a pretrained linear regression model. It loads model weights from a JSON file and applies the weights to transformed input data to generate a car price prediction.

Function: `predict(data, file_path)`

- **Parameters:**
  - `data` — A NumPy array containing preprocessed input features.
  - `file_path` — Placeholder parameter (currently unused).
- **Returns:** The predicted car price in real-world scale after inverse transformation.
- **Workflow:**
  1. Load trained model weights from a JSON file.
  2. Compute the linear prediction $y = \theta_0 + X \cdot \theta$.
  3. Use the stored target scaler to convert predictions back to original price scale.

Module 5: Utils Directory

5.1 utils.py

This module provides a collection of helper functions for data encoding, transformation, and scaling used in the preprocessing and prediction pipeline.

Function: `encode(data, column_name, json_file_path)`

- Replaces categorical column values with their numeric encodings using a JSON mapping file.
- **Inputs:**
  - `data`: pandas DataFrame containing the column.
  - `column_name`: Name of the categorical column.
  - `json_file_path`: Path to JSON encoding dictionary.
- **Output:** Modifies the DataFrame in place.

Function: `convert_to_ohe_input(value, column_name, json_file_path)`

- Converts a single categorical input into a one-hot encoded row.
- **Special Case:** For the `model` column, assumes 774 categories.
- **Output:** pandas DataFrame with one-hot encoded representation.

Function: `keep_relevant_columns(data, file_path)`

- Filters out columns with Pearson correlation below a threshold, using index values stored in a NumPy file.
- **Output:** Filtered pandas DataFrame.

Function: `apply_pca_by_Q_mat(file_path, data)`

- Applies Principal Component Analysis using a precomputed orthogonal matrix (Q).
- **Output:** NumPy array with reduced dimensions.

Function: `scale(file_path, data)`

- Loads a pre-trained scaler and transforms the data accordingly.
- **Output:** Scaled NumPy array.

Function: `un_scale(file_path, data)`

- Applies inverse transformation to restore original values from scaled data using a pre-trained scaler.
- **Output:** Unscaled data in original units.

Module 6: Other Important Files

This section documents auxiliary files essential for setting up, configuring, and running the project.

docker-compose.yml

Defines and manages multi-container Docker applications. It configures services like the Flask backend, PostgreSQL, Redis, and sets up networking, environment variables, and volume bindings.

Dockerfile

Specifies the build instructions for creating a custom Docker image for the Flask backend. It installs dependencies, copies application code, and defines the startup command.

lib-install.sh

A shell script used during Docker image creation to install necessary system-level packages (like build tools, Python libraries, and system dependencies) required by the app.

requirements.txt

Lists all Python packages and their versions required to run the project. Used with pip to install the environment consistently across development, testing, and production.

auto-dir-setup.sh

Script for setting up project folder structure and placeholder files automatically. It helps maintain project organization and can be run at initial setup.

kaggle-data-download

A utility or script (could be Python or Bash) to automate downloading datasets directly from Kaggle using the Kaggle API. Ensures reproducibility by handling data acquisition.

src.ipynb

A Jupyter notebook created initially on Google Colab containing the full workflow: from data cleaning and preprocessing to model training and evaluation. Acts as a single-point reference for experimental development before integration into the production pipeline.