Low-Level Design (LLD)

Low-Level Design (LLD)

1 Introduction

This document provides the detailed implementation specifications for the modules and components outlined in the High-Level Design.

1.2 Prediction API Module (`app.py`)

This module is the core of the online prediction service.

- **Endpoint Definition:**
  - **URL:** `/predict`
  - **HTTP Method:** `POST`
- **Request Body (JSON Format):** The API expects a JSON object containing key-value pairs for 21 mushroom features.

  JSON

  ```
  {
      "cap-shape": "x",
      "cap-surface": "s",
      "cap-color": "n",
      "bruises": "t",
      "odor": "p",
      "gill-attachment": "f",
      "gill-spacing": "c",
      "gill-size": "n",
      "gill-color": "k",
      "stalk-shape": "e",
      "stalk-root": "e",
      "stalk-surface-above-ring": "s",
      "stalk-surface-below-ring": "s",
      "stalk-color-above-ring": "w",
      "stalk-color-below-ring": "w",
      "veil-color": "w",
      "ring-number": "o",
      "ring-type": "p",
      "spore-print-color": "k",
      "population": "s",
      "habitat": "u"
  }
  ```

- **Response Body (JSON Format):** The API returns a JSON object with the prediction and the probabilities.

  JSON

  ```
  {
    "prediction": "Poisonous",
    "probability": {
      "edible": 0.0157,
      "poisonous": 0.9843
    }
  ```

Low-Level Design (LLD)

- **Logic Flow of `predict()` function:**

  1. The Flask application loads the `mushroom_model.joblib` and `model_columns.joblib` files into memory on startup.
  2. When a `POST` request is received at `/predict`, the incoming JSON data is parsed.
  3. The JSON object is converted into a single-row Pandas DataFrame.
  4. The DataFrame is one-hot encoded using `pd.get_dummies()`.
  5. The encoded DataFrame's columns are aligned with the original training columns using `reindex()`, ensuring consistency and filling any missing columns with `0`.
  6. The prepared DataFrame is passed to the loaded model's `.predict()` and `.predict_proba()` methods.
  7. The results are formatted into the final JSON response structure and returned to the client.

1.3 Dockerfile Specification

The `Dockerfile` defines the steps to build the application container image.

Dockerfile

```
# Use an official Python runtime as a parent image
FROM python:3.9-slim

# Set the working directory in the container to /app
WORKDIR /app

# Copy the requirements file into the container
COPY requirements.txt .

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of the application's code into the container
COPY . .

# Run the application using Gunicorn on port 8080, as required by Cloud Run
CMD ["gunicorn", "-b", "0.0.0.0:8080", "app:app"]
```