Low Level Document

Mushroom Prediction

Written By: Agam Patel

Document Version: 1.0

Last Date of Revision: 04/07/2025

# Document Control

## Change Record

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 0.1 | 01/07/2025 | Agam Patel | Initial draft and architecture included |
| 0.2 | 02/07/2025 | Agam Patel | Module descriptions, flow diagram added |
| 1.0 | 04/07/2025 | Agam Patel | Final version with unit tests appended |

# Table of Contents

| 1. Introduction | 4 |
| --- | --- |
| 1. What is Low-Level Design? | 4 |
| 1. Scope | 4 |
| 1. Architecture | 5 |
| 1. Architecture Description | 5 |
| 1. Data Description | 5 |
| 1. Data Ingestion | 6 |
| 1. Data Validation | 7 |
| 1. Data Transformation | 7 |
| 1. Model Training | 8 |
| 1. Model Evaluation | 9 |
| 1. Prediction Pipeline | 9 |
| 1. Deployment (MLOps) | 10 |
| 1. Unit Test Cases | 10 |
| 15. Unit Test Cases | 11 |

# 

# 1. Introduction

The Mushroom Classification System is designed to predict whether a mushroom is edible or poisonous based on its physical characteristics. It is implemented using a modular machine learning pipeline with built-in MLOps support. This Low-Level Design (LLD) document provides an in-depth, technical breakdown of each component, class, function, data flow, and dependencies used in the implementation. It complements the High-Level Design (HLD) by presenting implementation-ready details for developers and reviewers.

# 2. What is Low-Level Design (LLD)?

Low-Level Design (LLD) focuses on the detailed internal workings of each module defined in the High-Level Design. It includes the structure of classes, method definitions, input-output mappings, configurations, and file organization. While HLD gives a macro-level architecture, LLD serves as the micro-level blueprint that enables developers to begin implementation or testing immediately.

# 3. Scope

This LLD document covers the internal logic and structure of the following components:

* Data ingestion and preprocessing using Pandas and YAML-based configuration
* Schema validation and sanity checks
* Label encoding and train-test data splitting
* Model training using three classifiers (Logistic Regression, Random Forest, XGBoost)
* Model evaluation and saving of metrics
* MLflow-based experiment tracking
* DVC versioning
* Prediction pipeline implementation using saved encoders and model
* Modular code structure organized under src/mushroomPrediction/

It also includes the class diagrams, function signatures, configuration files, and unit test outlines necessary for complete understanding and development.

# 4. Architecture

The architecture of the Mushroom Classification System is modular and configuration-driven. It is organized into separate components to handle each stage of the machine learning pipeline, supported by configuration and entity classes. All paths and parameters are handled via external YAML configuration files, ensuring flexibility and maintainability.

# 5. Architecture Description

* **main.py**: The central script responsible for executing each stage sequentially via pipeline functions.
* **src/mushroomPrediction/components/**: Contains modular Python scripts that handle core pipeline stages like data ingestion, validation, transformation, model training, evaluation, and prediction.
* **src/mushroomPrediction/config/**: Contains the configuration.py module, which loads parameters from YAML files and prepares config objects.
* **src/mushroomPrediction/entity/**: Contains dataclasses that define structured configurations (DataIngestionConfig, ModelTrainingConfig, etc.).
* **artifacts/**: Auto-generated folder that stores raw and transformed datasets, model files, encoders, evaluation reports, and logs.
* **config/**: Contains config.yaml and params.yaml, which hold configurable paths and model parameters.
* **MLflow and DVC Integration**: MLflow is used for experiment tracking and model registry. DVC is used for dataset and stage versioning.



# 6. Data Description

The system uses a publicly available dataset from the UCI Machine Learning Repository, accessed via Kaggle under the name mushroom-classification. It contains 8,124 instances of mushrooms described by 22 categorical features. The target label (class) indicates whether a mushroom is edible or poisonous.

* **Input Format**:  
   A CSV file named mushrooms.csv with no missing values, containing all categorical features. Each row represents one mushroom sample.
* **Key Columns**:  
  + Input features: cap-shape, cap-surface, cap-color, odor, gill-size, etc.
  + Target: class — encoded as 0 for edible and 1 for poisonous.
* **Raw Data Path**:  
   artifacts/raw/mushrooms.csv
* **Transformed Data Paths**:  
  + artifacts/transformed/X\_train.csv
  + artifacts/transformed/y\_train.csv
  + artifacts/transformed/X\_test.csv
  + artifacts/transformed/y\_test.csv
* **Encoders**:  
  + artifacts/encoders/ordinal\_encoder.pkl (for feature columns)
  + artifacts/encoders/label\_encoder\_y.pkl (for target column)

# 7. Data Ingestion Module

The Data Ingestion module is responsible for fetching the raw mushroom dataset from an external source and preparing it for further processing. In this project, the dataset is downloaded using the KaggleHub API, which enables programmatic access to publicly available Kaggle datasets. The dataset used is the "Mushroom Classification" dataset hosted under the user uciml.

The ingestion logic is encapsulated in a class named DataIngestion, located in the dataIngestionComponent.py file within the src/mushroomPrediction/components directory. When initialized, the class receives a configuration object of type DataIngestionConfig, which provides essential inputs such as the dataset link, file name to extract, and the local directory where the file should be saved.

The module contains logic to create the necessary directory structure if it does not already exist. It then downloads the dataset and saves the specified CSV file to the target location. If the file already exists, the system avoids redundant downloads unless explicitly triggered.

The output of this module is a raw CSV file, typically named mushrooms.csv, saved in the artifacts/raw/ directory. This file serves as the foundation for all subsequent stages in the machine learning pipeline.

# 8. Data Validation Module

The Data Validation module ensures that the input dataset is clean, well-structured, and suitable for further processing and model training. This module acts as a gatekeeper by checking the structural integrity and statistical soundness of the dataset before it enters the transformation pipeline.

The module is implemented in a class named DataValidation, located in the dataValidationComponent.py file within the src/mushroomPrediction/components directory. When initialized, this class accepts a configuration object of type DataValidationConfig, which provides the expected schema (defined in schema.yaml) and the path to the raw dataset.

The core logic of the module performs several key checks. Firstly, it ensures that all expected columns are present in the dataset. Secondly, it verifies that there are no missing values or completely null columns. The validation step also checks for class imbalance, ensuring that the distribution between edible and poisonous mushrooms is reasonable and doesn't heavily skew toward one class, which could affect model performance.

If the dataset passes all validation checks, a flag is set and logged to confirm success. If any validation fails, the pipeline is halted, and a structured exception is raised, identifying the exact issue.

This module plays a crucial role in preventing corrupted, incomplete, or biased data from contaminating the downstream machine learning workflow.

# 9. Data Transformation Module

The Data Transformation module is responsible for preparing the dataset for machine learning by converting categorical features into numerical form and splitting the data into training and testing sets. This step ensures that the data is in the right format for model consumption and that the target column is correctly encoded.

The module is implemented in a class named DataTransformation, located in the dataTransformationComponent.py file under the src/mushroomPrediction/components directory. When initialized, the class receives a configuration object of type DataTransformationConfig, which includes the paths for raw data input, processed data output, and encoder storage.

The transformation process begins by reading the validated raw dataset. All input features, which are originally in categorical form, are encoded using the OrdinalEncoder from scikit-learn. This encoder transforms each categorical value into an integer, while handling unseen or unknown values during inference. The target column, class, is separately encoded using LabelEncoder, converting edible (e) and poisonous (p) labels into numerical values (typically 0 and 1). Both encoders are saved as .pkl files in the artifacts/encoders directory for later reuse during prediction.

Once encoding is complete, the module splits the dataset into training and testing subsets using an 80-20 stratified split to maintain class balance. These transformed datasets are then saved in the artifacts/transformed directory.

The Data Transformation module plays a foundational role in ensuring that the models receive clean, numerical input and that the same transformations can be applied consistently at prediction time.

# 10. Model Training Module

The Model Training module is responsible for training multiple machine learning classifiers on the transformed dataset and identifying the model that performs best based on evaluation metrics. This module introduces flexibility, experimentation, and reproducibility to the core training process.

The training logic is encapsulated in a class named ModelTrainer, located in the modelTrainerComponent.py file within the src/mushroomPrediction/components directory. When instantiated, this class accepts a configuration object of type ModelTrainingConfig, which provides paths to the training datasets, model output directory, and parameter settings for each classifier.

This module supports training of three distinct models: Logistic Regression, Random Forest, and XGBoost. Each model is trained on the encoded training data and evaluated using the testing data. For each classifier, the module computes accuracy, precision, recall, and F1-score. Hyperparameters can be tuned either manually or through simple grid-based logic within the codebase. All metrics and training details are logged to MLflow to enable experiment tracking and comparison.

Once the evaluation scores are obtained, the module selects the best-performing model based on F1-score (or another preferred metric), and saves it in the artifacts/model/ directory using Joblib. This saved model is later reused by the prediction pipeline.

The Model Training module ensures that model selection is systematic, repeatable, and recorded, providing a robust foundation for inference and deployment.

# 11. Model Evaluation Module

The Model Evaluation module is designed to assess the performance of trained models on both the training and testing datasets. It calculates essential metrics that indicate how well the model generalizes to unseen data. This evaluation ensures that the selected model is neither underfitting nor overfitting and is appropriate for deployment.

This logic resides in the modelEvaluationComponent.py file and typically works in tandem with the Model Training module. It loads the trained model along with the test dataset and computes various performance metrics, including accuracy, precision, recall, and F1-score. These metrics are calculated using scikit-learn’s evaluation functions.

In addition to displaying these scores, the module saves the evaluation results in a structured report format (e.g., reports.json) within the artifacts/evaluation/ directory. The same metrics are also logged to MLflow, allowing comparisons between multiple models and training runs.

This module provides critical feedback to ensure that only well-performing models proceed to deployment and enables transparent monitoring of model quality over time.

# 12. Prediction Pipeline

The Prediction Pipeline is responsible for making real-time predictions based on user input. It represents the deployment-facing part of the system, where the trained model is applied to new data to infer whether a given mushroom is edible or poisonous.

This pipeline is implemented in the predictionComponent.py file under the src/mushroomPrediction/components directory. When executed, it loads the saved model and encoders from the artifacts directory. The user input, provided as a dictionary of categorical values (e.g., cap-shape: bell, odor: almond), is first converted into a DataFrame and then encoded using the previously saved OrdinalEncoder.

Once transformed, the input is passed to the trained model for prediction. The model returns the class label in encoded form (typically 0 or 1), which is then translated back to the original human-readable output: "edible" or "poisonous". Proper error handling is in place to catch and report issues like missing fields or unseen categorical values.

The prediction module ensures a consistent and reliable inference mechanism, making it suitable for integration into web applications or APIs.

# 13. Deployment and MLOps

Deployment in the Mushroom Classification System follows the principles of MLOps, which ensure reliable delivery, versioning, and monitoring of machine learning models in production. The project integrates key MLOps tools such as MLflow and DVC to manage the model lifecycle and ensure reproducibility.

MLflow is used extensively during the training and evaluation phases to track experiments, model versions, and associated metrics. Each model is logged with its parameters, performance scores, and artifacts, enabling easy comparison between different runs. This makes it simple to select and serve the best-performing model when needed.

DVC (Data Version Control) is used to manage data and pipeline versioning. By linking datasets, intermediate artifacts, and stages through configuration and lock files, DVC ensures that the pipeline can be reproduced exactly at any point in the future. This is particularly useful in collaborative or production environments, where traceability is crucial.

The trained model and encoders are saved in the artifacts/ directory, which can be containerized using Docker for scalable deployment. An optional interface such as a REST API (using FastAPI) or a web app (using Streamlit) can be layered on top of the prediction pipeline to provide interactive access.

This MLOps setup supports not only automated training and prediction but also experimentation, reproducibility, and long-term maintainability.

# 14. Unit Testing

Unit testing is implemented to ensure the stability and correctness of individual modules. Each critical function or component is accompanied by test cases that verify input-output behavior, handle edge cases, and assert the integrity of results.

For instance, the data validation module is tested to ensure it correctly identifies schema mismatches and missing values. The transformation module is tested for proper label encoding and train-test splitting. Model training functions are verified to ensure models are trained and saved without errors. Similarly, the prediction pipeline is tested with both valid and invalid inputs to confirm accurate responses and graceful failure handling.

Tests are implemented using Python’s built-in unittest or pytest framework. They are organized in a separate tests/ directory, with test scripts corresponding to each module. Logging and assertions are used within test cases to confirm expected outcomes.

Automating unit tests ensures confidence in the codebase, supports continuous integration, and catches regressions early during development.

# 15. Unit Test Cases

| Test Case | Pre-requisite | Expected Results |
| --- | --- | --- |
| Validate raw CSV matches expected schema | Raw dataset 'mushrooms.csv' present in artifacts/ | All required columns exist and match schema definition |
| Check for missing/null values in dataset | Raw data loaded using pandas | No column or row contains null values |
| Encode features and target correctly | Dataset passed to transformation module | All features are encoded numerically, target is 0/1 |
| Split dataset into training and test sets | Encoded data available | Training and test datasets are saved successfully |
| Train all three models and select the best | Training data is valid | Best model based on F1-score is saved |
| Evaluate model performance and save metrics | Trained model and test data available | Accuracy, precision, recall, F1 are saved and logged |
| Generate prediction from user input | Encoders and model are loaded | Output is either 'edible' or 'poisonous' |
| Handle invalid or unseen input values gracefully | Model and encoders are loaded | Graceful error message or fallback applied |
| Log model details to MLflow | MLflow server or local tracking enabled | Metrics and models logged to MLflow |
| Track pipeline stage changes with DVC | DVC initialized and configured | DVC tracks changes in data, model, and pipeline files |