Cryptocurrency Liquidity Prediction: A Low-Level Design

Introduction

This Low-Level Design (LLD) document offers a comprehensive overview of the **Cryptocurrency Liquidity Prediction** project's implementation. It systematically breaks down all critical components including the directory structure, key scripts, and classes that support data processing and machine learning workflows. The document further elaborates on the Flask API setup, deployment configurations, and environment management. Its primary purpose is to provide developers and technical stakeholders with clear, component-wise guidance to streamline the development and maintenance of the predictive system, ensuring robustness and scalability throughout the project lifecycle.

Directory & File Structure

The project is organized into a clear and logical directory structure to facilitate easy navigation and maintenance. The **DATASET/** folder contains raw and processed data files such as data_with_liquidity.csv and cleaned_data.csv. The **EDA REPORT/** directory holds exploratory data analysis notebooks, including EDA_Analysis-checkpoint.ipynb, which assists in understanding data patterns and distributions.

The **NORMALIZATION_&_PREDICTION/** folder contains notebooks responsible for feature engineering, normalization, and prediction logic, notably featureengineering.ipynb and Normalization_Prediction.ipynb. For the web interface, **static/** hosts CSS styling files, while **templates/** includes HTML templates like index.html to render frontend pages.

Key standalone files include app.py for the Flask API, requirements.txt listing Python dependencies, and serialized model artifacts such as liquidity_scaler.pkl and tuned_liquidity_model.pkl. Development is supported with a virtual environment located in venv/, encapsulating all necessary packages.

Component-wise Implementation

Data Handling & Preprocessing

The initial step involves loading the cryptocurrency dataset using pandas.read_csv() to import data from DATASET/data_with_liquidity.csv. This dataset includes key attributes such as coin, symbol, date, liquidity, 24h_volume, mkt_cap, 1h, and price.

Preprocessing removes non-predictive columns coin, symbol, and date to reduce noise. Missing values, if any, are handled by imputation or removal to ensure data integrity. An important engineering step adds a binary feature liquidity_level derived as:

```
df['liquidity_level'] = (df['liquidity'] > 0.5).astype(int)
```

This label enables classification into high or low liquidity, facilitating threshold-based prediction.

Feature Engineering & Selection

Feature selection focuses on four predictive variables deemed most relevant: 24h_volume, mkt_cap, 1h, and price. These features are extracted and optionally transformed or combined to improve model performance. New interaction terms or scaling transformations may be applied within the featureengineering.ipynb notebook. The resulting clean and reduced dataset is saved as cleaned_data.csv for downstream tasks, preserving a tidy and consistent input format.

Data Normalization

To standardize feature magnitudes, StandardScaler from sklearn.preprocessing is employed. The scaler is first fit on the training split to compute mean and variance parameters. Then, it transforms both training and test data sets to zero-centered unit variance form, ensuring stable and unbiased model fitting. The fitted scaler object is serialized using pickle for reuse during prediction:

```
import pickle
with open('liquidity_scaler.pkl', 'wb') as f:
    pickle.dump(scaler, f)
```

Model Training & Hyperparameter Tuning

The core regression model implemented is RandomForestRegressor. The cleaned, normalized data is split into train/test subsets using train_test_split. An initial baseline model trains on the training set to establish a performance benchmark.

Improvement follows hyperparameter tuning conducted by GridSearchCV, exploring critical parameters such as n_estimators (number of trees), max_depth (maximum tree depth), and min_samples_split (minimum samples to split a node). The tuning objective is maximizing the coefficient of determination (r2_score), which assesses predictive accuracy. The best-performing model from the grid search is serialized analogous to the scaler:

```
with open('tuned_liquidity_model.pkl', 'wb') as f:
    pickle.dump(best_model, f)
```

Prediction Pipeline

The prediction stage encapsulates the loading and invocation of the scaler and trained model via a dedicated function predict_liquidity(volume, mkt_cap, h1, price). The feature vector is first converted into a NumPy array and scaled with the stored scaler. The model generates a continuous liquidity score, which is then thresholded at 0.5 to yield a qualitative "High" or "Low" liquidity classification. A typical implementation snippet looks like this:

```
def predict_liquidity(volume, mkt_cap, h1, price):
    features = np.array([[volume, mkt_cap, h1, price]])
    scaled = scaler.transform(features)
    liquidity = model.predict(scaled)[0]
    level = "High" if liquidity > 0.5 else "Low"
    return liquidity, level
```

Flask Web/API Layer

The Flask-based web application (app.py) provides the user interface and prediction service. Two primary routes are defined:

- / – Renders the home page using the index.html template, which presents a form for inputting feature values.
- /predict – Accepts POST requests to receive inputs, invokes the predict_liquidity function, and returns the prediction result.

The frontend is styled via CSS located in the static/style.css folder. Backend logic ensures input validation and error handling to maintain smooth user interactions and robust predictions.

Configuration & Dependencies

All project dependencies are enumerated in requirements.txt, including:

- pandas for data manipulation
- numpy for numerical operations
- scikit-learn for machine learning algorithms and preprocessing
- flask to serve the web application
- pickle for serializing model and scaler objects (builtin Python library)

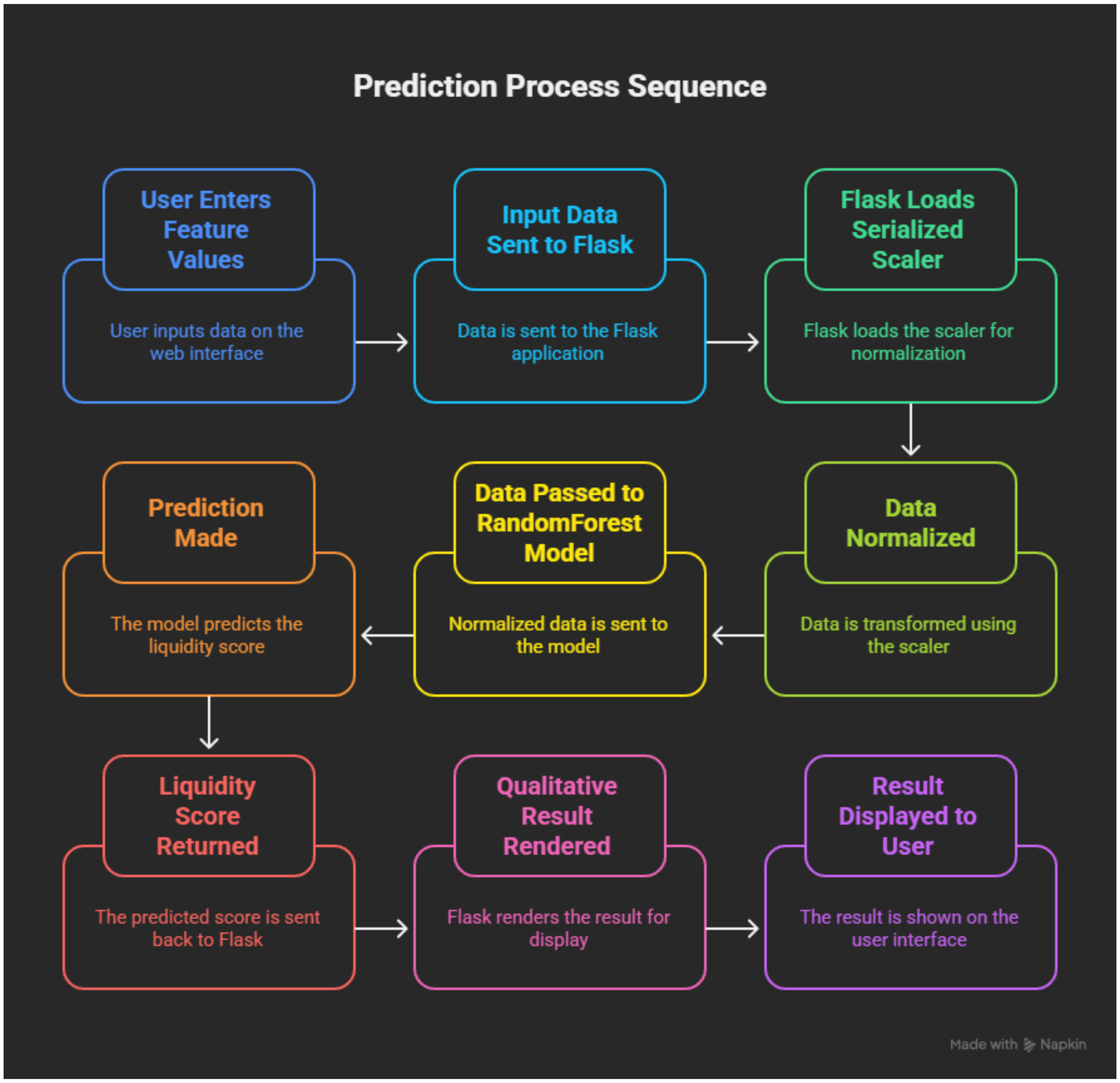
Development is recommended inside a virtual environment (venv/) to isolate dependencies and maintain reproducibility across environments. The application targets Python 3.x for compatibility.

Sequence Diagrams

The included Mermaid sequence diagrams visually represent the core interaction flows within the Cryptocurrency Liquidity Prediction system, clarifying key communication and data processing steps.

Prediction Sequence Diagram

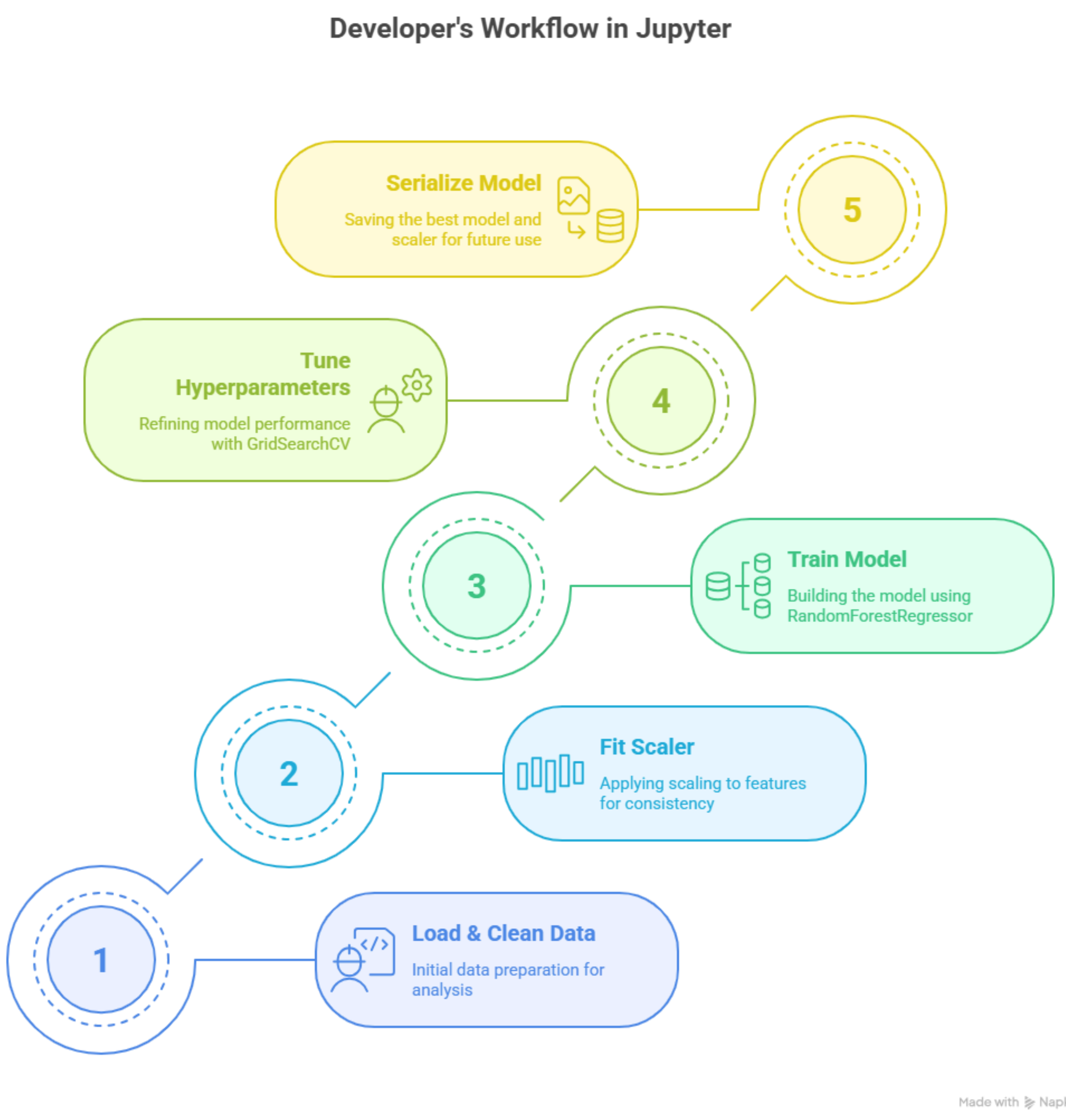
This diagram illustrates the end-to-end process triggered when a user submits input via the web interface. The flow begins with the user entering feature values on the frontend, which forwards the input data to the Flask application through a POST request. The Flask app then loads the serialized scaler to normalize these inputs before passing the transformed features to the trained RandomForest model for prediction. Finally, the predicted liquidity score is returned to the Flask app, which renders the qualitative result ("High" or "Low") back to the user interface for display.



Prediction Process Sequence
User Enters Feature Values
User inputs data on the web interface
Input Data Sent to Flask
Data is sent to the Flask application
Flask Loads Serialized Scaler
Flask loads the scaler for normalization
Data Normalized
Data is transformed using the scaler
Data Passed to RandomForest Model
Normalized data is sent to the model
Prediction Made
The model predicts the liquidity score
Liquidity Score Returned
The predicted score is sent back to Flask
Qualitative Result Rendered
Flask renders the result for display
Result Displayed to User
The result is shown on the user interface
Made with Napkin

Training Sequence Diagram

The training sequence outlines the developer's workflow in the Jupyter notebook environment, starting with loading and cleaning raw data. Subsequently, the scaler is fit and applied to the feature set, followed by model training using the RandomForestRegressor. Hyperparameter tuning via GridSearchCV refines model performance, after which the best model and fitted scaler are serialized and saved. These diagrams collectively provide a high-level visual summary of data flow and component interactions crucial for development, debugging, and maintenance.


Developer's Workflow in Jupyter
5
Serialize Model
Saving the best model and scaler for future use
4
Tune Hyperparameters
Refining model performance with GridSearchCV
3
Train Model
Building the model using RandomForestRegressor
2
Fit Scaler
Applying scaling to features for consistency
1
Load & Clean Data
Initial data preparation for analysis
Made with Napkin

Error Handling & Logging

Robust error handling is integral to ensuring reliable operation of the prediction system. Both frontend and backend implement **input validation** to confirm user inputs are numeric and within expected ranges, preventing invalid data from processing.

The backend encloses scaler and model loading within try-except blocks to catch file access or deserialization errors gracefully. When exceptions occur, clear and informative messages are returned to guide users or developers in troubleshooting.

Optionally, the application can incorporate logging mechanisms to record prediction requests and errors. This facilitates ongoing monitoring, diagnostics, and aids in maintaining operational robustness and observability throughout deployment.

Future-Proofing

To enhance maintainability and scalability, the Flask application and model loading logic are modularized for seamless upgrades and feature additions. The project is prepared to support RESTful API endpoints with JSON input/output, enabling flexible integrations. Environment variables or configuration files manage paths and secrets securely, promoting robust deployment practices.