Cryptocurrency Liquidity Prediction Pipeline: Architecture and Data Flow

Pipeline Overview

The cryptocurrency liquidity prediction pipeline is designed as a modular and sequential architecture, ensuring a structured flow of data from raw input to final liquidity prediction. This approach promotes clarity, maintainability, and scalability across all stages involved in processing and modeling.

The pipeline consists of seven main stages:

1. **Data Ingestion:** Loading raw cryptocurrency market data for analysis.
2. **Data Preprocessing & Cleaning:** Removing irrelevant information and handling missing values to ensure data quality.
3. **Feature Engineering & Selection:** Extracting and selecting meaningful variables that influence liquidity prediction.
4. **Data Normalization:** Standardizing features to allow fair comparison and improve model performance.
5. **Model Training & Hyperparameter Tuning:** Building and optimizing the predictive model using machine learning algorithms.
6. **Model Serialization:** Saving trained models and scalers for later reuse without retraining.
7. **Prediction and Inference:** Applying the model on new data through an API or interface to generate liquidity forecasts.

Pipeline Flow Diagram

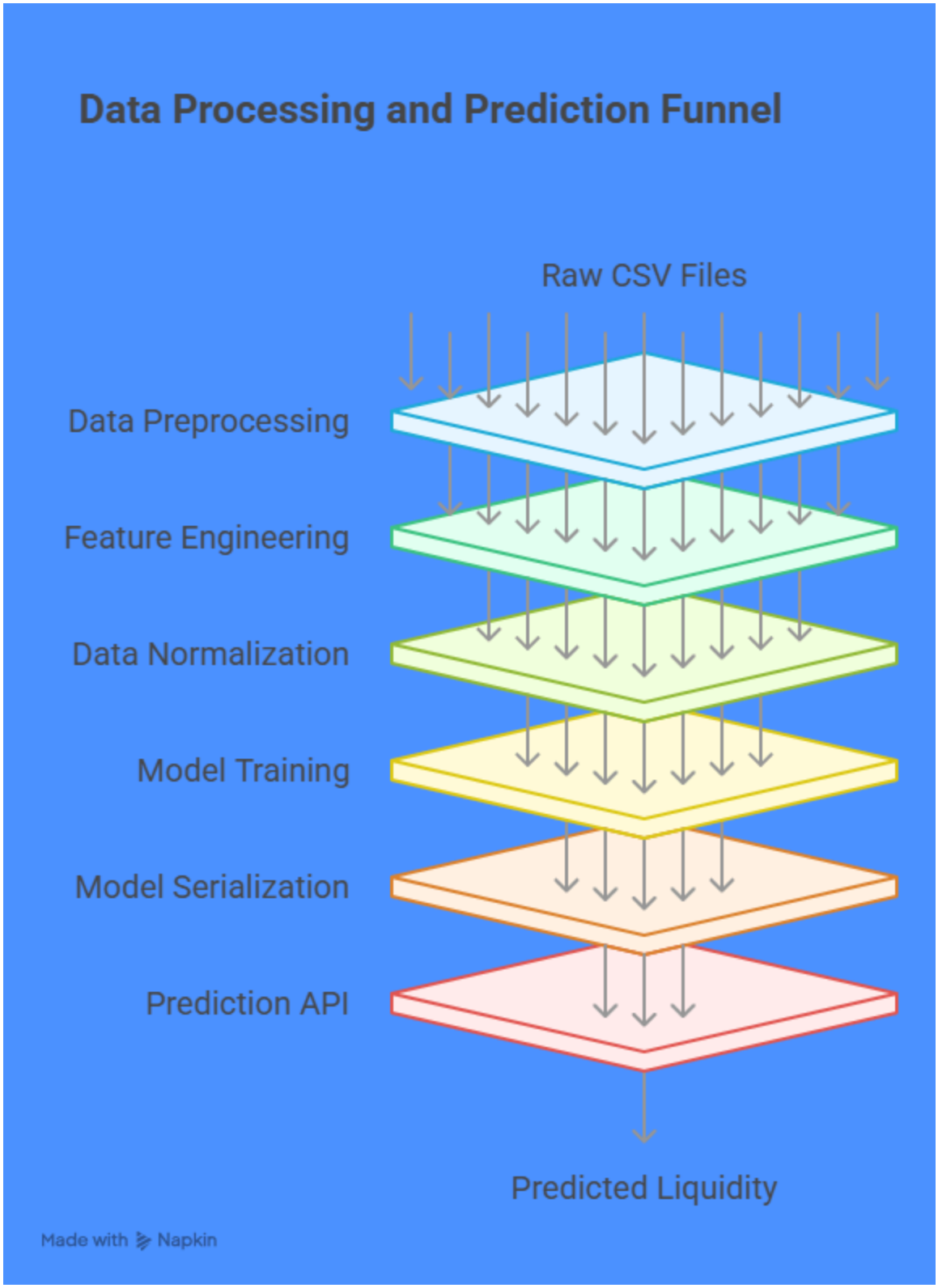
The following Mermaid flowchart visually illustrates the end-to-end data process within the pipeline. It begins with **Raw CSV Files** containing historical cryptocurrency data, which are ingested into the system.

The data then passes through successive transformations:

- **Data Preprocessing & Cleaning:** Removes irrelevant columns and handles missing values.
- **Feature Engineering & Selection:** Extracts meaningful predictors for liquidity.
- **Data Normalization:** Standard scales features for unbiased modeling.
- **Model Training:** A RandomForestRegressor learns liquidity trends.
- **Model & Scaler Serialization:** Saves artifacts as .pkl files for reuse.
- **Prediction API/UI:** A Flask application accepts user or new data inputs, applies scaling and the model, and returns liquidity predictions labeled as *"High"* or *"Low"*.

This diagram clearly represents the smooth flow of data and component roles, emphasizing modularity and reusability throughout the pipeline.

```
flowchart TD
    A[Raw CSV Files (DATASET/)] --> B[Data Preprocessing & Cleaning]
    B --> C[Feature Engineering & Selection]
    C --> D[Data Normalization (StandardScaler)]
    D --> E[Model Training (RandomForestRegressor)]
    E --> F[Model & Scaler Serialization (.pkl files)]
    F --> G[Prediction API/UI (Flask)]
    H[User/New Data] --> G
    G --> I[Predicted Liquidity ("High"/"Low")]
```


Data Processing and Prediction Funnel
Raw CSV Files
Data Preprocessing
Feature Engineering
Data Normalization
Model Training
Model Serialization
Prediction API
Predicted Liquidity
Made with Napkin

Step-by-Step Data Flow

Data Ingestion

The initial step involves loading raw cryptocurrency data from a CSV file located at DATASET/data_with_liquidity.csv. This is performed using the pandas.read_csv() method, which efficiently reads and parses the dataset into a DataFrame format suitable for downstream processing. The ingested data contains historical market statistics and associated liquidity labels.

Data Preprocessing & Cleaning

Once loaded, the dataset undergoes preprocessing to enhance quality and relevance. Irrelevant columns such as coin, symbol, and date are dropped to focus modeling on numeric features. Missing or null values in the dataset are handled via imputation or row removal to prevent biased model outcomes. Additionally, a new binary target variable, liquidity_level, is derived by applying a threshold to liquidity metrics to classify instances into *High* or *Low* liquidity.

Feature Engineering & Selection

Core features selected for model input include 24h_volume, mkt_cap, 1h price, and price itself. Optional derived features can be created to capture interaction effects or temporal trends. Following selection, the refined dataset is saved as cleaned_data.csv for transparency and reproducibility in subsequent steps.

Data Normalization

To mitigate scale disparities among features, StandardScaler from the sklearn.preprocessing module standardizes the data. This transformation centers features around zero mean and unit variance, ensuring that no single feature disproportionately influences the model.

Model Training & Hyperparameter Tuning

The normalized data is used to train a RandomForestRegressor, a robust and interpretable ensemble method suitable for regression tasks. Hyperparameters are optimized via GridSearchCV, which systematically evaluates candidate parameter combinations to maximize predictive performance. Model evaluation metrics, including the coefficient of determination (R^2), quantify fit quality and help prevent overfitting.

Model & Scaler Serialization

After training, both the fitted model and scaler objects are serialized using Python's pickle module. These artifacts are saved respectively as tuned_liquidity_model.pkl and liquidity_scaler.pkl, enabling consistent transformation and prediction on new data without requiring retraining.

Prediction and Inference

For inference, user or new input data is received via a web interface or API endpoint. The backend first loads the serialized scaler and model, applies the scaler transformation to the input features, then uses the trained model to predict liquidity levels. The output provides both a probability score and a binary classification label of either *"High"* or *"Low"* liquidity, which is then returned to the user interface for presentation.

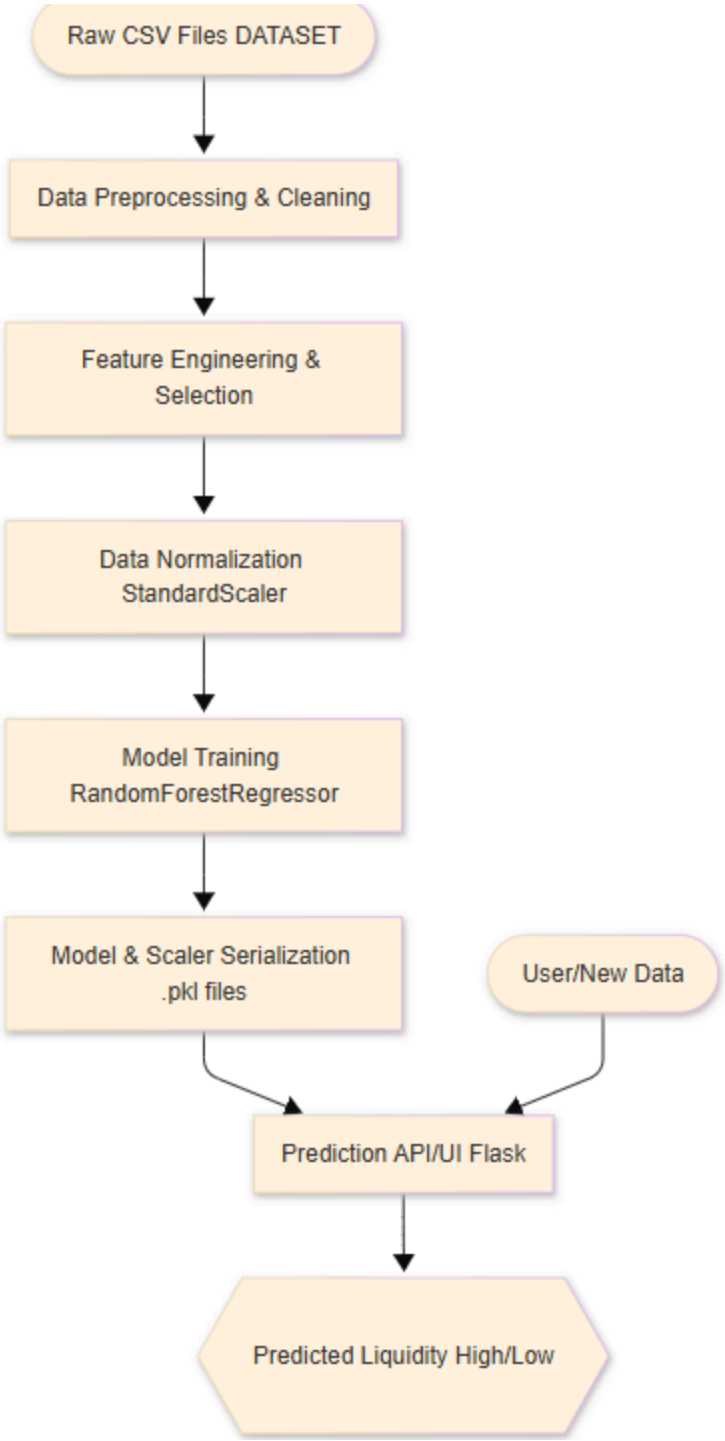
Component Interaction

The pipeline's components interact seamlessly to form a cohesive system. Notebooks execute critical tasks including data cleaning, exploratory data analysis, feature engineering, normalization, and model training. Once trained, the model and scaler are serialized into .pkl files for easy reuse, preventing redundant computations. The Flask application (app.py) acts as the interface layer, managing user input validation, applying the saved scaler for preprocessing, invoking the trained model for prediction, and displaying results. This modular design ensures clear separation of concerns, promoting maintainability and reliable, repeatable liquidity predictions.

Example Data Flow (End-to-End)

A team member uploads new market data to the DATASET/ directory. They then run the processing notebook, which cleans the data, engineers features, normalizes inputs, and trains the model. This produces key outputs: cleaned_data.csv, liquidity_scaler.pkl, and tuned_liquidity_model.pkl.

Next, a user accesses the Flask app, inputs cryptocurrency data, which is scaled and passed to the model. The app returns a real-time liquidity prediction, demonstrating a seamless and repeatable pipeline from raw data to final inference.


Raw CSV Files DATASET
Data Preprocessing & Cleaning
Feature Engineering & Selection
Data Normalization StandardScaler
Model Training RandomForestRegressor
Model & Scaler Serialization .pkl files
User/New Data
Prediction API/UI Flask
Predicted Liquidity High/Low