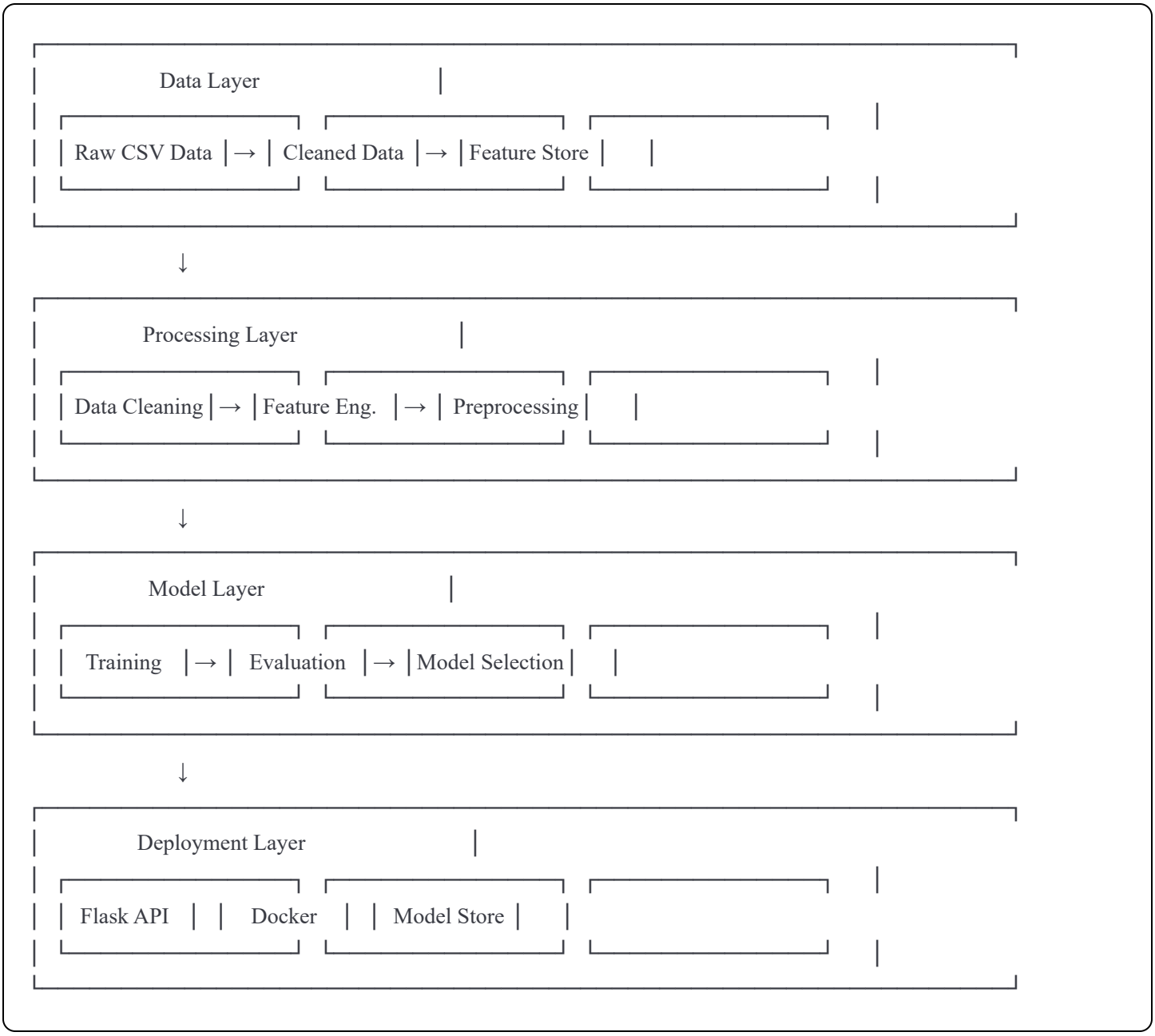
Heart Disease Prediction - Design Document

1. System Architecture

1.1 High-Level Architecture



1.2 Component Description

Data Layer

- **Raw Data Storage**: Original CSV file preservation
- **Cleaned Data**: Processed, validated data
- **Feature Store**: Engineered features ready for training

Processing Layer

- **Data Cleaning Module**: Missing value imputation, outlier detection
- **Feature Engineering**: Creating derived features, encoding
- **Preprocessing Pipeline**: Scaling, transformation, balancing

Model Layer

- **Training Engine**: Multiple algorithm training with cross-validation
- **Evaluation Module**: Performance metrics calculation
- **Model Selection**: Best model identification and optimization

Deployment Layer

- **REST API**: Flask-based prediction service
- **Containerization**: Docker for portability
- **Model Persistence**: Serialized model storage

2. Data Processing Design

2.1 Data Cleaning Strategy

Missing Value Handling

```
python

Strategy by Feature Type:

Numerical Features:
├── Age, BMI, Blood Pressure → Median imputation
├── Cholesterol, Triglycerides → Mean imputation (grouped by Age)
└── CRP, Homocysteine → KNN imputation (k=5)

Categorical Features:
├── Exercise Habits, Stress Level → Mode imputation
├── Alcohol Consumption → Special category "Unknown"
└── Binary features → Most frequent value
```

Outlier Detection

- **Method**: IQR (Interquartile Range) method
- **Action**: Cap at 1.5 * IQR boundaries

- **Features**: Blood Pressure, Cholesterol, BMI, Triglycerides

Data Validation Rules

```
python

Validation Rules:
├── Age: 18 <= age <= 120
├── Blood Pressure: 80 <= bp <= 250
├── BMI: 10 <= bmi <= 60
├── Cholesterol: 100 <= chol <= 500
└── Sleep Hours: 0 <= sleep <= 24
```

2.2 Feature Engineering

Derived Features

1. **Age Groups**:
   - Young Adult (18-35)
   - Middle Age (36-55)
   - Senior (56-80)
2. **BMI Categories**:
   - Underweight (<18.5)
   - Normal (18.5-24.9)
   - Overweight (25-29.9)
   - Obese (≥30)
3. **Risk Score**: Composite risk based on multiple factors

```
python

  risk_score = (
    high_bp * 0.3 +
    diabetes * 0.25 +
    smoking * 0.2 +
    family_history * 0.15 +
    low_exercise * 0.1
  )
```

4. **Cholesterol Ratio**: HDL/LDL interaction
5. **Lifestyle Score**: Exercise + Sleep + Stress combination

Feature Encoding Strategy

```
python

Encoding Approach:

Binary Features (Yes/No, Male/Female):
└── Label Encoding (0, 1)

Ordinal Features (Low/Medium/High):
└── Ordinal Encoding (0, 1, 2)

Nominal Features (if any):
└── One-Hot Encoding

Target Variable:
└── Label Encoding (No=0, Yes=1)
```

2.3 Feature Scaling

- **Method**: StandardScaler (zero mean, unit variance)
- **Applied to**: All numerical features
- **Pipeline**: Fit on training set, transform on test set

3. Model Design

3.1 Baseline Models

Model Selection Rationale

1. Logistic Regression
   ├── Pros: Interpretable, fast, probability output
   ├── Use: Baseline, feature importance
   └── Hyperparameters: C (regularization), penalty

2. Random Forest
   ├── Pros: Handles non-linearity, feature importance
   ├── Use: Ensemble baseline, robust to outliers
   └── Hyperparameters: n_estimators, max_depth, min_samples_split

3. Gradient Boosting (XGBoost)
   ├── Pros: High performance, handles imbalance
   ├── Use: Primary candidate for best performance
   └── Hyperparameters: learning_rate, max_depth, n_estimators

4. Support Vector Machine (SVM)
   ├── Pros: Effective in high dimensions
   ├── Use: Comparison benchmark
   └── Hyperparameters: C, kernel, gamma

5. LightGBM
   ├── Pros: Fast training, efficient memory usage
   ├── Use: Alternative to XGBoost
   └── Hyperparameters: num_leaves, learning_rate

3.2 Handling Class Imbalance

Techniques to Implement

1. **SMOTE** (Synthetic Minority Over-sampling Technique)
   - Generate synthetic samples for minority class
   - Apply only on training data
2. **Class Weights**
   - Adjust loss function to penalize minority errors more
   - Parameter: `class_weight='balanced'`
3. **Stratified Sampling**
   - Maintain class distribution in train/test split
   - Use `stratify` parameter
4. **Ensemble Methods**

- Inherently handle imbalance better
- Combine with sampling techniques

3.3 Hyperparameter Tuning

Search Strategy

```
python

Approach: RandomizedSearchCV
├── Reason: Faster than GridSearch, good results
├── Cross-Validation: 5-fold Stratified K-Fold
├── Scoring: F1-Score (balance precision/recall)
└── Iterations: 50 random combinations

Parameter Grids:

Random Forest:
├── n_estimators: [100, 200, 300, 500]
├── max_depth: [10, 20, 30, None]
├── min_samples_split: [2, 5, 10]
├── min_samples_leaf: [1, 2, 4]
└── max_features: ['sqrt', 'log2']

XGBoost:
├── learning_rate: [0.01, 0.05, 0.1, 0.2]
├── max_depth: [3, 5, 7, 9]
├── n_estimators: [100, 200, 300]
├── subsample: [0.6, 0.8, 1.0]
├── colsample_bytree: [0.6, 0.8, 1.0]
└── scale_pos_weight: [1, 2, 4] (for imbalance)
```

3.4 Model Evaluation Design

Metrics Framework

```
python
```

```
Primary Metrics:
├── F1-Score: Harmonic mean of precision/recall
├── ROC-AUC: Area under ROC curve
└── Precision-Recall AUC: For imbalanced data

Secondary Metrics:
├── Accuracy: Overall correctness
├── Precision: Positive predictive value
├── Recall: Sensitivity
├── Specificity: True negative rate
└── Matthews Correlation Coefficient

Confusion Matrix Analysis:
├── True Positives (TP): Correctly predicted disease
├── True Negatives (TN): Correctly predicted healthy
├── False Positives (FP): Incorrectly predicted disease
└── False Negatives (FN): Missed disease cases
```

Cross-Validation Strategy

- **Method**: Stratified K-Fold (k=5)
- **Rationale**: Maintains class distribution in each fold
- **Reporting**: Mean ± Standard Deviation for each metric

4. Code Architecture

4.1 Project Structure

```
heart_disease_prediction/
│
├── data/
│   ├── raw/                # Original data
│   ├── processed/          # Cleaned data
│   └── features/           # Engineered features
│
├── notebooks/
│   ├── 01_EDA.ipynb        # Exploratory analysis
│   ├── 02_Data_Cleaning.ipynb # Preprocessing
│   ├── 03_Feature_Engineering.ipynb
│   ├── 04_Model_Training.ipynb
│   └── 05_Model_Evaluation.ipynb
│
```

```
├── src/
│   ├── __init__.py
│   ├── data/
│   │   ├── __init__.py
│   │   ├── loader.py          # Data loading
│   │   ├── cleaner.py         # Cleaning functions
│   │   └── preprocessor.py    # Preprocessing pipeline
│   │
│   ├── features/
│   │   ├── __init__.py
│   │   ├── engineering.py     # Feature creation
│   │   └── selection.py       # Feature selection
│   │
│   ├── models/
│   │   ├── __init__.py
│   │   ├── train.py           # Training logic
│   │   ├── evaluate.py        # Evaluation metrics
│   │   └── predict.py         # Prediction interface
│   │
│   ├── visualization/
│   │   ├── __init__.py
│   │   └── plots.py           # Plotting functions
│   │
│   └── utils/
│       ├── __init__.py
│       └── helpers.py         # Utility functions
│
├── models/
│   ├── trained/               # Saved models
│   ├── scalers/               # Preprocessing objects
│   └── encoders/              # Encoding mappings
│
├── deployment/
│   ├── app.py                 # Flask application
│   ├── Dockerfile             # Container definition
│   ├── requirements.txt       # Dependencies
│   └── config.py              # Configuration
│
├── tests/
│   ├── test_data.py           # Data tests
│   ├── test_models.py         # Model tests
│   └── test_api.py            # API tests
│
├── docs/
```

```
│   ├── PROJECT_SPECIFICATION.md
│   ├── DESIGN.md
│   ├── API_DOCUMENTATION.md
│   └── MODEL_REPORT.md
│
├── requirements.txt         # Python dependencies
├── setup.py                 # Package installation
├── .gitignore               # Git ignore rules
└── README.md                # Project overview
```

4.2 Module Design

Data Processing Module

python

```python
class DataCleaner:
    def __init__(self, config):
        self.config = config

    def handle_missing_values(self, df):
        """Impute missing values based on strategy"""
        pass

    def detect_outliers(self, df):
        """Identify and handle outliers"""
        pass

    def validate_data(self, df):
        """Ensure data meets validation rules"""
        pass
```

Feature Engineering Module

python

```python
class FeatureEngineer:
    def __init__(self):
        self.encoders = {}

    def create_derived_features(self, df):
        """Generate new features"""
        pass

    def encode_categorical(self, df):
        """Encode categorical variables"""
        pass

    def scale_features(self, df):
        """Standardize numerical features"""
        pass
```

Model Training Module

```python
class ModelTrainer:
    def __init__(self, model_config):
        self.config = model_config
        self.models = {}

    def train_model(self, X, y, model_type):
        """Train specific model type"""
        pass

    def tune_hyperparameters(self, X, y, model):
        """Optimize hyperparameters"""
        pass

    def cross_validate(self, X, y, model):
        """Perform cross-validation"""
        pass
```

5. Deployment Design

5.1 API Design

Endpoints

```
POST /predict
├── Input: JSON with patient features
├── Output: Prediction + probability
└── Example:
   Request: {"age": 45, "gender": "Male", ...}
   Response: {"prediction": "Yes", "probability": 0.73}

GET /health
├── Purpose: Health check
└── Output: {"status": "healthy", "model_loaded": true}

GET /feature_importance
├── Purpose: Model interpretation
└── Output: Feature importance scores

POST /batch_predict
├── Purpose: Bulk predictions
└── Input: List of patient records
```

API Security

- Input validation
- Rate limiting
- HTTPS only (in production)
- API key authentication (optional)

5.2 Docker Configuration

Container Specs

```
dockerfile

Base Image: python:3.9-slim
Dependencies: requirements.txt
Exposed Port: 5000
Health Check: /health endpoint
Volume Mount: /models for persistence
```

5.3 Model Versioning

- Filename format: `model_v{version}_{date}.pkl`
- Metadata: Training date, metrics, features used

- Rollback capability: Keep previous versions

6. Testing Strategy

6.1 Unit Tests

- Data validation functions
- Feature engineering correctness
- Model loading/saving
- API endpoint responses

6.2 Integration Tests

- End-to-end pipeline
- API with model predictions
- Docker container functionality

6.3 Model Tests

- Prediction format validation
- Performance on validation set
- Handling edge cases

7. Monitoring and Logging

7.1 Logging Strategy

```
python

Logging Levels:
├── DEBUG: Detailed diagnostic info
├── INFO: General information flow
├── WARNING: Something unexpected
└── ERROR: Serious problems

Log Content:
├── Data loading events
├── Model training progress
├── Prediction requests
├── Error stack traces
└── Performance metrics
```

7.2 Model Monitoring

- Track prediction distribution
- Monitor feature drift
- Log response times
- Alert on anomalies

8. Performance Optimization

8.1 Code Optimization

- Vectorized operations (NumPy)
- Efficient data structures
- Lazy loading for large datasets
- Parallel processing where applicable

8.2 Model Optimization

- Feature selection to reduce dimensionality
- Model compression techniques
- Batch prediction for efficiency
- Caching frequent predictions

Document Version: 1.0
Last Updated: January 2026
Author: ML Engineering Team