CardioDetect: Complete Technical Deep Dive

A Comprehensive Technical Manual for Project Defense & Understanding

---

TABLE OF CONTENTS

PART I: THE DATA FOUNDATION

> **"Data is the fuel that powers machine learning. Without good data, even the best algorithms fail."**

Before we can build any machine learning model, we need data—and not just any data, but high-quality, clinically validated data that represents the patterns we want our model to learn. This section explains every dataset we use, why we chose it, where it came from, and how we prepared it for training.

Chapter 1: Understanding Our Datasets

1.1 The Fundamental Challenge of Medical ML

Building a machine learning system for cardiovascular disease prediction presents a unique challenge: we need to answer two fundamentally different clinical questions, and no single dataset can answer both.

Question 1: "Does this patient have heart disease RIGHT NOW?"

This is a **detection** problem. To answer it, we need data that includes the results of diagnostic tests—specifically, stress tests that show how the heart behaves under exertion. These tests reveal problems that might not be visible when the patient is at rest. Features like maximum heart rate achieved during exercise, whether the patient experienced chest pain during the test, and changes in the heart's electrical activity (ST depression) are critical for this task.

Question 2: "What is this patient's risk of developing heart disease in the next 10 years?"

This is a **prediction** problem. To answer it, we need longitudinal data—data collected from patients who were healthy at first and then followed for many years to see who developed heart disease and who didn't. This type of data tells us which risk factors (like high blood pressure, smoking, or high cholesterol) actually predict future disease.

Because these two questions require fundamentally different types of data, we use two primary datasets: the **UCI Heart Disease Dataset** for detection and the **Framingham Heart Study** for prediction.

1.2 The UCI Heart Disease Dataset: Detecting Current Disease

What Is This Dataset? The UCI Heart Disease Dataset is one of the most famous benchmark datasets in machine learning. It was collected in 1988 from four different medical centers: the Cleveland Clinic Foundation in Ohio, the Hungarian Institute of Cardiology in Budapest, the VA Medical Center in Long Beach, California, and University Hospital in Zurich, Switzerland.

The dataset is publicly available through the UCI Machine Learning Repository, maintained by the University of California, Irvine. Researchers Robert Detrano, Andras Janosi, Walter Steinbrunn, and Matthias Pfisterer collected this data specifically to study whether machine learning algorithms could help predict the presence of heart disease.

Source Information

Property	Value
Official Name	Heart Disease Dataset
Source	UCI ML Repository
Year Collected	1988
Total Samples	303 (Cleveland subset)
Target Variable	`target` → 0 = No disease, 1 = Disease present

Why We Chose This Dataset The key advantage of the UCI dataset is that it contains **stress test results**. During a cardiac stress test, a patient exercises (usually on a treadmill) while doctors monitor their heart. This test reveals information that simply cannot be obtained from a resting patient:

1. **Maximum Heart Rate Achieved (thalach):** A healthy heart can beat faster during exercise. If a patient's maximum heart rate is unusually low for their age, it may indicate heart disease.
2. **Exercise-Induced Angina (exang):** Some patients only experience chest pain during exercise, not at rest. This is a significant warning sign.
3. **ST Depression (oldpeak):** The ST segment is a specific part of the heart's electrical signal (visible on an ECG). During exercise, changes in this segment can indicate that parts of the heart muscle aren't getting enough blood.
4. **Slope of ST Segment (slope):** How the ST segment behaves during recovery from exercise provides additional diagnostic information.

Complete Feature Description (14 Features) **Feature 1: Age (age)** — Age is the single most important risk factor for heart disease. As we age, our arteries naturally become stiffer and less flexible, a process called arterial stiffening. Plaque also accumulates on artery walls over time. In the UCI dataset, patient ages range from 29 to 77 years.

Feature 2: Sex (sex) — Biological sex significantly affects heart disease risk. Men generally develop heart disease about 10 years earlier than women, partly because estrogen provides some protection to women before menopause. Coded as 0 for female and 1 for male.

Feature 3: Chest Pain Type (cp) — This is one of the most diagnostically significant features. Cardiologists classify chest pain into four categories: - **Type 0 - Typical Angina:** Classic heart-related chest pain with pressure/squeezing, triggered by exertion, relieved by rest. Strongly associated with heart disease. - **Type 1 - Atypical Angina:** Has some but not all features of typical angina. - **Type 2 - Non-Anginal Pain:** Doesn't follow angina pattern; often muscular or digestive. - **Type 3 - Asymptomatic:** No chest pain despite possible heart disease ("silent ischemia").

Feature 4: Resting Blood Pressure (trestbps) — Blood pressure measured at rest in mmHg. Normal is below 120, Stage 1 hypertension is 130-139, Stage 2 is 140+. Dataset range: 94-200 mmHg.

Feature 5: Serum Cholesterol (chol) — Total cholesterol in mg/dL. Cholesterol builds up in artery walls, forming plaques that restrict blood flow. Range: 126-564 mg/dL.

Feature 6: Fasting Blood Sugar (fbs) — Binary: whether blood sugar exceeds 120 mg/dL. Indicates diabetes/pre-diabetes, which accelerates heart disease.

Feature 7: Resting ECG Results (restecg) — Measures heart's electrical activity at rest: - Value 0: Normal - Value 1: ST-T wave abnormality (blood supply issues) - Value 2: Left ventricular hypertrophy (enlarged heart chamber)

Feature 8: Maximum Heart Rate Achieved (thalach) — Highest heart rate during stress test. A healthy heart beats faster during exercise. Range: 71-202 bpm.

Feature 9: Exercise-Induced Angina (exang) — Binary: whether exercise triggered chest pain. Strong indicator of coronary artery disease.

Feature 10: ST Depression (oldpeak) — How much the ST segment dropped during exercise vs rest (in mm). Higher values indicate more severe blood supply problems. Range: 0-6.2 mm.

Feature 11: Slope of Peak Exercise ST Segment (slope) — Shape of ST segment at peak exercise: - Value 0: Upsloping (less concerning) - Value 1: Flat (concerning) - Value 2: Downsloping (most concerning, strongly associated with disease)

Feature 12: Major Vessels Colored (ca) — Number of heart's major vessels showing blockages on fluoroscopy (0-3). More vessels affected = more severe disease.

Feature 13: Thalassemia (thal) — Thallium stress test results: - Value 1: Normal blood flow - Value 2: Fixed defect (permanent damage) - Value 3: Reversible defect (temporary blood flow reduction)

Feature 14: Target (target) — Ground truth label (0 = no disease, 1 = disease) determined by cardiac catheterization (angiography), the gold standard.

1.3 The Framingham Heart Study: Predicting Future Risk

Historical Significance The **Framingham Heart Study** is one of the most important medical studies in history. Started in 1948 in Framingham, Massachusetts, it has followed **three generations** of participants to understand cardiovascular disease.

This study **discovered**: - The link between **cholesterol** and heart disease - The dangers of **high blood pressure** - The cardiovascular effects of **smoking** - The concept of **"risk factors"** (term coined by this study)

Source Information

Property	Value
Official Name	Framingham Heart Study
Source	NIH/NHLBI
Started	1948 (still ongoing - 75+ years!)
Our Subset	~11,500 samples
Target Variable	`TenYearCHD` $\rightarrow$ 10-year CHD risk

Core Features (11 Features) **age** — Primary risk factor (range: 32-70) **sex** — 0=Female, 1=Male **totChol** — Total cholesterol (107-696 mg/dL) **sysBP** — Systolic blood pressure (83-295 mmHg) **diaBP** — Diastolic blood pressure (48-142 mmHg) **BMI** — Body Mass Index (15-56 kg/m^2) **heartRate** — Resting heart rate (44-143 bpm) **glucose** — Fasting glucose (40-394 mg/dL) **currentSmoker** — Binary smoking status **diabetes** — Binary diabetes status **BPMeds** — Binary: on blood pressure medication

Engineered Features (24 Additional) We created additional features using domain knowledge:

Feature	Formula	Medical Rationale
`pulse_pressure`	SBP - DBP	Arterial stiffness indicator
`map`	DBP + (SBP-DBP)/3	Mean arterial pressure - organ perfusion
`chol_ratio`	TotalChol / HDL	Bad vs good cholesterol balance
`age_bp_interaction`	Age × SBP	Age amplifies BP damage
`metabolic_score`	Composite	Multiple risk factors combined

Chapter 2: Data Preprocessing & Splitting

2.1 Data Quality Issues

Real medical data has many problems that must be addressed:

Missing Values: Patients miss tests, values aren't recorded. Example: glucose = NaN because patient didn't fast.

Impossible Values: Data entry errors create physiologically impossible values. Example: cholesterol = 0 (everyone has cholesterol), age = 150.

Inconsistent Units: Different labs use different units. Example: cholesterol in mmol/L vs mg/dL.

Different Label Formats: Datasets use different representations. Example: “yes”/“no” vs 1/0 vs “positive”/“negative”.

2.2 Our Preprocessing Steps

Step 1: Handle Impossible Values

```
def clean_impossible_values(df):
    # Cholesterol can't be 0 or >600
    df.loc[(df['chol'] == 0) | (df['chol'] > 600), 'chol'] = np.nan

    # Blood pressure bounds
    df.loc[(df['sysBP'] < 60) | (df['sysBP'] > 250), 'sysBP'] = np.nan

    return df
```

Step 2: Imputation with Median

We use **median** instead of mean because medical data often contains outliers:

```
Example: Blood Pressure Values
[120, 125, 130, 135, 300]  ← 300 is a data entry error

Mean = (120+125+130+135+300) / 5 = 162    Biased by outlier!
Median = 130 (middle value)               Robust to outliers!
```

Step 3: Standardization

Different features have vastly different scales. We normalize to mean=0, std=1:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
# Now all features have mean=0, std=1
```

2.3 Data Splitting: 70% / 15% / 15%

Why Split Data?

If we train and test on the same data, the model memorizes answers instead of learning patterns. This is **overfitting**—great performance on training data, terrible on new patients.

Our Split: - **Training (70%):** Model learns from this data - **Validation (15%):** Used to tune hyperparameters; detect overfitting - **Test (15%):** Final evaluation; never touched until the end

Stratified Splitting:

We ensure each split has the same proportion of disease cases:

```
Original: 75.3% healthy, 24.7% disease
Train:    75.3% healthy, 24.7% disease
Valid:    75.3% healthy, 24.7% disease
Test:     75.3% healthy, 24.7% disease
```

2.4 Handling Class Imbalance

Heart disease is relatively rare (~25% of our data). A model predicting "healthy" for everyone would achieve 75% accuracy—but miss every disease case!

Solution 1: Class Weights

```
XGBClassifier(scale_pos_weight=3.0)  # Penalize missed disease 3x more
```

Solution 2: Threshold Adjustment Lower the decision threshold from 0.50 to 0.25 to catch more true cases.

Solution 3: Stratified K-Fold Maintain class ratios during cross-validation.

PART II: MACHINE LEARNING PIPELINE

> **"The goal is to turn data into information, and information into insight."**

Chapter 3: Feature Engineering - Creating 34 Features from 12 Inputs

3.1 Why Feature Engineering Matters

Raw patient data contains 12 base measurements. But medical knowledge tells us that certain COMBINATIONS and TRANSFORMATIONS of these values are more predictive than the values alone.

For example, "Pulse Pressure" (systolic BP minus diastolic BP) is a better indicator of arterial stiffness than either BP value alone. We create these derived features to give our model access to medical domain knowledge.

3.2 Base Features (12)

These are the raw inputs we collect from patients:

Category	Features
Demographics	`age`, `sex`

Category	Features
Lifestyle	`smoking`, `bp_meds` (on medication)
Conditions	`hypertension`, `diabetes`
Vitals	`systolic_bp`, `diastolic_bp`, `heart_rate`
Labs	`total_cholesterol`, `fasting_glucose`, `bmi`

3.3 Derived Features (22)

File: `Milestone_2/experiments/train_honest.py`

Vital Calculations (2 features)

```
# Pulse Pressure: difference between systolic and diastolic
# High PP (>50) indicates stiff arteries
df['pulse_pressure'] = df['systolic_bp'] - df['diastolic_bp']

# Mean Arterial Pressure: average blood pressure during cardiac cycle
# Formula: DBP + (PP / 3) because heart spends 2/3 of time in diastole
df['map'] = df['diastolic_bp'] + (df['pulse_pressure'] / 3)
```

Medical Rationale: - **Pulse pressure > 50:** Indicates arterial stiffening (worse prognosis) - **MAP:** Critical for organ perfusion; MAP < 60 means organs aren't getting enough blood

Risk Flags (4 features)

```
# Binary flags based on clinical cutoffs
df['hypertension_flag'] = ((df['systolic_bp'] >= 140) | (df['diastolic_bp'] >= 90)).astype(i
df['high_cholesterol_flag'] = (df['total_cholesterol'] >= 240).astype(int)
df['high_glucose_flag'] = (df['fasting_glucose'] >= 126).astype(int)
df['obesity_flag'] = (df['bmi'] >= 30).astype(int)
```

Medical Rationale: - 140/90 mmHg is the hypertension threshold (ACC/AHA Stage 2) - 240 mg/dL cholesterol is "high" per ATP III guidelines - 126 mg/dL fasting glucose indicates diabetes (ADA criteria) - BMI 30 is clinical obesity (WHO definition)

Metabolic Syndrome Score (1 feature)

```
# Sum of metabolic risk flags (0-4)
# Metabolic syndrome = 3+ of these conditions together
df['metabolic_syndrome_score'] = (
    df['hypertension_flag'] +
    df['high_cholesterol_flag'] +
    df['high_glucose_flag'] +
    df['obesity_flag']
)
```

Medical Rationale: Metabolic syndrome (score 3) dramatically increases cardiovascular risk—more than the sum of individual factors. The clustering of these conditions indicates underlying insulin resistance.

Age Groups (5 features - one-hot encoded)

```
df['age_group_<40'] = (df['age'] < 40).astype(int)
df['age_group_40-49'] = ((df['age'] >= 40) & (df['age'] < 50)).astype(int)
df['age_group_50-59'] = ((df['age'] >= 50) & (df['age'] < 60)).astype(int)
df['age_group_60-69'] = ((df['age'] >= 60) & (df['age'] < 70)).astype(int)
df['age_group_70+'] = (df['age'] >= 70).astype(int)
```

Why Bin Age? Cardiovascular risk isn't linear with age. A 35→40 year transition is less significant than a 55→60 transition. Binning captures these non-linear age effects.

BMI Categories (4 features - one-hot encoded)

```
df['bmi_cat_Underweight'] = (df['bmi'] < 18.5).astype(int)
df['bmi_cat_Normal'] = ((df['bmi'] >= 18.5) & (df['bmi'] < 25)).astype(int)
df['bmi_cat_Overweight'] = ((df['bmi'] >= 25) & (df['bmi'] < 30)).astype(int)
df['bmi_cat_Obese'] = (df['bmi'] >= 30).astype(int)
```

Medical Rationale: WHO BMI categories capture health risk better than raw BMI. The relationship between BMI and mortality is U-shaped (both under and overweight increase risk).

Log Transforms (3 features)

```
# Log transform for right-skewed distributions
df['log_total_cholesterol'] = np.log1p(df['total_cholesterol'])
df['log_fasting_glucose'] = np.log1p(df['fasting_glucose'])
df['log_bmi'] = np.log1p(df['bmi'])
```

Why Log Transform? Cholesterol and glucose have long right tails (some people have very high values). Log transforms: 1. Reduce the influence of extreme outliers 2. Make distributions more normal 3. Help tree-based models make better splits

Interaction Features (3 features)

```
# Combinations that have synergistic effects
df['age_sbp_interaction'] = df['age'] * df['systolic_bp']
df['bmi_glucose_interaction'] = df['bmi'] * df['fasting_glucose']
df['age_smoking_interaction'] = df['age'] * df['smoking']
```

Medical Rationale: - **Age × SBP:** High BP is more dangerous in elderly patients - **BMI × Glucose:** Obesity accelerates diabetic complications - **Age × Smoking:** Cumulative damage from smoking increases with age

3.4 Feature Summary Table

Category	Features	Count
Demographics	age, sex	2
Lifestyle	smoking, bp_meds	2
Conditions	hypertension, diabetes	2
Vitals	systolic_bp, diastolic_bp, heart_rate	3
Labs	total_cholesterol, fasting_glucose, bmi	3
Subtotal: Base		**12**
Vital Calculations	pulse_pressure, map	2
Risk Flags	hypertension, cholesterol, glucose, obesity	4
Metabolic Score	metabolic_syndrome_score	1
Age Groups	5 bins (one-hot)	5
BMI Categories	4 bins (one-hot)	4
Log Transforms	log_cholesterol, log_glucose, log_bmi	3
Interactions	age×sbp, bmi×glucose, age×smoking	3
Subtotal: Derived		**22**
TOTAL		**34**

Chapter 4: XGBoost - Our Core Algorithm

4.1 What Is XGBoost?

XGBoost (Extreme Gradient Boosting) is an ensemble learning technique that combines multiple decision trees sequentially. Each new tree specifically tries to correct the errors of previous trees.

3.2 How Gradient Boosting Works

Imagine teaching a class:

1. **First lecture:** Students get 60% of questions right.
2. **Second lecture:** Focus on topics they got wrong. Now 75% correct.
3. **Third lecture:** Focus on remaining mistakes. Now 85% correct.
4. Continue until near-perfect performance.

XGBoost does the same with decision trees:

1. **Tree 1:** Makes initial predictions. Some are wrong.
2. **Tree 2:** Trained on the *errors* of Tree 1.
3. **Tree 3:** Trained on remaining errors.
4. **Final prediction:** Sum of all trees' contributions.

3.3 Why XGBoost Over Deep Learning?

Factor	XGBoost	Deep Learning
Best for	Tabular data	Images, text, audio
Data needed	Thousands	Millions
Training time	Minutes	Hours/Days
Hardware	CPU sufficient	Needs GPU
Interpretability	Feature importance built-in	Black box

Our data: ~**16,000 samples, 34 tabular features** → XGBoost is optimal.

Chapter 4: Dual Model Architecture

4.1 Two Clinical Questions, Two Models

We answer two fundamentally different questions, each requiring a specialized model:

```
                    CARDIODETECT DUAL MODEL ARCHITECTURE



         DETECTION MODEL                      PREDICTION MODEL

      Question: "Does this               Question: "What is the
      patient have heart                 10-year risk of developing
      disease RIGHT NOW?"                heart disease?"

      Dataset: UCI Heart Disease         Dataset: Framingham Study
      Samples: 303                       Samples: ~11,500
      Features: 14                       Features: 35 (11+24)

      Key Features:                      Key Features:
      • Stress test results              • BP, Cholesterol, BMI
      • Exercise-induced angina          • Smoking, Diabetes
      • ST depression                    • 10-year follow-up data

      Architecture: Voting               Architecture: XGBoost
      Ensemble (4 models)                with threshold tuning

      Accuracy: 91.45%                   Accuracy: 94.01%
```

4.2 Detection Model: Diagnosing Current Disease

Purpose: Determine if a patient currently has heart disease based on clinical indicators and stress test results.

Dataset: UCI Heart Disease (303 samples, 14 features including stress test data)

Architecture: Voting Classifier ensemble of 4 models

Why Ensemble for Detection? The UCI dataset is small (303 samples). Individual models on small datasets are prone to high variance—small changes in training data cause large changes in predictions. By combining 4 diverse models, we reduce this variance and get more stable, reliable predictions.

Performance: - Accuracy: 91.45% - Uses features that require clinical testing (ECG, stress test)

4.3 Prediction Model: Forecasting Future Risk

Purpose: Estimate a patient's 10-year risk of developing cardiovascular disease.

Dataset: Framingham Heart Study (~11,500 samples, 35 features)

Architecture: Single XGBoost model with optimized threshold (0.25)

Why Single Model for Prediction? The Framingham dataset is large enough (~11,500 samples) that a well-tuned XGBoost model achieves excellent performance without needing ensemble averaging. The additional complexity of an ensemble isn't justified when a single model performs at 94.01% accuracy.

Performance: - Accuracy: 94.01% - Sensitivity: 62.77% (with optimized threshold) - Uses features obtainable from routine checkups (no stress test required)

4.4 Singleton Pattern - Loading Both Models

File: `services/ml_service.py`, Lines 20-36

```python
class MLService:
    _instance = None  # Single shared instance

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance._load_pipeline()  # Load BOTH models ONCE
        return cls._instance

    def _load_pipeline(self):
        # Load detection model (ensemble)
        self.detection_model = joblib.load('models/detection_ensemble.pkl')
        # Load prediction model (XGBoost)
        self.prediction_model = joblib.load('models/prediction_xgb.pkl')
```

Result: First request loads both models (2-3s), subsequent requests use cached models (milliseconds).

Chapter 5: The Ensemble Models Explained

5.1 Why Four Different Algorithms?

Each algorithm in our ensemble has different strengths and makes different types of errors. By combining them, we get the best of all approaches:

```
                      VOTING CLASSIFIER ENSEMBLE


     Patient Data







  XGBoost    LightGBM     Random        Extra
                         Forest        Trees
   0.72       0.68         0.75          0.70       ← Probabilities





              Average: 0.7125


              Final Prediction: Disease (if   0.50)
```

5.2 Model 1: XGBoost (Extreme Gradient Boosting)

How It Works: XGBoost builds trees sequentially, where each new tree corrects the errors of previous trees. It uses gradient descent to minimize a loss function.

```
Tree 1: Initial predictions (many errors)
    ↓ Calculate residuals (errors)
Tree 2: Trained to predict residuals
    ↓ Add Tree 2's predictions to Tree 1's
Tree 3: Trained on remaining errors
    ...
Final: Sum of all 300 trees
```

Key Characteristics: - **Regularization:** L1 and L2 regularization prevent overfitting - **Handling Missing Values:** Automatically learns best direction for missing values - **Parallel Processing:** Can use multiple CPU cores

Hyperparameters We Used:

```
XGBClassifier(
    n_estimators=300,       # 300 trees
    max_depth=4,            # Each tree has max 4 levels
    learning_rate=0.05,     # Small steps for stability
    subsample=0.8,          # Use 80% of data per tree
    colsample_bytree=0.8,   # Use 80% of features per tree
)
```

Strengths: Best overall accuracy on tabular data, built-in feature importance
Weaknesses: Can overfit on very small datasets

5.3 Model 2: LightGBM (Light Gradient Boosting Machine)

How It Works: LightGBM is also gradient boosting, but with two key innovations that make it faster:

1. **Leaf-wise Growth:** Instead of growing trees level-by-level (like XGBoost), LightGBM grows the leaf that reduces loss the most. This reaches better accuracy with fewer leaves.
2. **Histogram-based Splitting:** Instead of testing every possible split point, it buckets continuous values into histograms, dramatically speeding up training.

```
XGBoost: Level-wise growth        LightGBM: Leaf-wise growth




(Balanced tree)                   (Deeper on one side)
```

Key Characteristics: - **Speed:** 2-3x faster training than XGBoost - **Memory Efficient:** Uses less RAM - **Categorical Feature Support:** Native handling without one-hot encoding

Hyperparameters We Used:

```
LGBMClassifier(
    n_estimators=300,
    max_depth=4,
    learning_rate=0.05,
    num_leaves=31,          # Max leaves per tree
    min_child_samples=20,   # Minimum samples per leaf
)
```

Strengths: Fastest gradient boosting, handles large datasets well **Weaknesses:** Can overfit with too many leaves on small data

---

5.4 Model 3: Random Forest

How It Works: Random Forest builds many decision trees **independently** (not sequentially) and averages their predictions. Each tree is trained on a random subset of data (bagging) and considers only a random subset of features at each split.

```
Original Data

        Bootstrap Sample 1    Tree 1    Prediction 1
          (Random 70% of data)

        Bootstrap Sample 2    Tree 2    Prediction 2
          (Different random 70%)

        Bootstrap Sample 3    Tree 3    Prediction 3
                 ...

Final Prediction = Average(Tree 1, Tree 2, ..., Tree N)
```

Key Differences from Gradient Boosting: - Trees are built **independently**, not sequentially - No learning rate—each tree contributes equally - More randomness at each step

Why Include It? Random Forest makes different types of errors than gradient boosting methods. It's less prone to overfitting on noisy data and provides diversity in our ensemble.

Hyperparameters We Used:

```
RandomForestClassifier(
    n_estimators=300,       # 300 trees
    max_depth=6,            # Slightly deeper than XGBoost
    min_samples_split=10,  # Minimum samples to split a node
    min_samples_leaf=4,    # Minimum samples in leaf
    max_features='sqrt',   # Use sqrt(n_features) at each split
)
```

Strengths: Robust to overfitting, fast parallel training, handles outliers well **Weaknesses:** Less accurate than gradient boosting on tabular data

---

5.5 Model 4: Extra Trees (Extremely Randomized Trees)

How It Works: Extra Trees is like Random Forest, but with even more randomness:

1. **No Bootstrapping:** Uses the entire dataset for each tree (not random samples)
2. **Random Splits:** Instead of finding the optimal split threshold, it chooses thresholds randomly

```
Random Forest:              Extra Trees:
For each split,             For each split,
find BEST threshold         pick RANDOM threshold

Feature X > 5.7 (optimal)   Feature X > 3.2 (random)
```

Why Even More Randomness? The additional randomness reduces variance even further. While each individual Extra Trees model might be less accurate, it provides unique perspectives that improve ensemble diversity.

Hyperparameters We Used:

```
ExtraTreesClassifier(
    n_estimators=300,
    max_depth=6,
    min_samples_split=10,
    min_samples_leaf=4,
    max_features='sqrt',
)
```

Strengths: Fastest training, maximum variance reduction, complements other models well **Weaknesses:** Individual accuracy lower than XGBoost

---

5.6 Soft Voting: Why We Average Probabilities

Hard Voting: Each model votes 0 or 1. Majority wins.

```
XGBoost: 1, LightGBM: 0, RF: 1, ET: 1 → Final: 1 (3-1 vote)
```

Problem: Loses confidence information. A 51% prediction counts the same as 99%.

Soft Voting (Our Choice): Average the probabilities, then apply threshold.

```
XGBoost: 0.72, LightGBM: 0.68, RF: 0.75, ET: 0.70
Average: 0.7125 → Final: 1 (above 0.50 threshold)
```

Advantage: Captures model confidence. A model that's 99% sure has more influence than one that's 51% sure.

5.7 Ensemble Performance Comparison

Model	Individual Accuracy	In Ensemble
XGBoost	88.2%	
LightGBM	87.5%	
Random Forest	86.1%	
Extra Trees	85.8%	
Voting Ensemble	**91.45%**	—

Key Insight: The ensemble outperforms every individual model by 3-5%. This is because different models make different errors, and averaging cancels out model-specific mistakes.

5.8 Advanced: Stacking Classifier

Beyond voting, we also experimented with **Stacking**—a more sophisticated ensemble technique:

```python
from sklearn.ensemble import StackingClassifier

stacking = StackingClassifier(
    estimators=[
        ('lgbm', LGBMClassifier(...)),
        ('xgb', XGBClassifier(...)),
        ('hgb', HistGradientBoostingClassifier(...))
    ],
    final_estimator=LogisticRegression(C=1.0, max_iter=1000),
    cv=5,
    passthrough=True,  # Also pass original features to meta-learner
    n_jobs=-1
)
```

How Stacking Works:

```
                    STACKING CLASSIFIER


   Original Features (34)





 LightGBM  XGBoost  HistGB   ← Level 0: Base Learners
   0.72     0.68    0.70







    + Original 34     ← "passthrough=True" adds original features
      features




   Logistic           ← Level 1: Meta-Learner
   Regression            (learns to combine base predictions)


     Final: 0.71
```

Why Stacking Outperforms Voting: - Voting just averages probabilities (equal weight) - Stacking LEARNS optimal weights through the meta-learner - `passthrough=True` lets the meta-learner also use original features for calibration

---

Chapter 6: Optuna - Automated Hyperparameter Tuning

6.1 Why Optuna?

Manually testing hyperparameter combinations is tedious. Optuna automates this with intelligent search algorithms.

File: `Milestone_2/experiments/train_detection_maximum.py`

6.2 How Optuna Works

```python
import optuna
```

```python
def lgbm_objective(trial):
    """Each trial tests a different hyperparameter combination."""
    params = {
        'n_estimators': trial.suggest_int('n_estimators', 100, 500),
        'max_depth': trial.suggest_int('max_depth', 3, 12),
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.3, log=True),
        'num_leaves': trial.suggest_int('num_leaves', 20, 150),
        'min_child_samples': trial.suggest_int('min_child_samples', 10, 100),
        'subsample': trial.suggest_float('subsample', 0.6, 1.0),
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.6, 1.0),
    }

    model = LGBMClassifier(**params)
    scores = cross_val_score(model, X_train, y_train, cv=5, scoring='accuracy')
    return scores.mean()  # Optuna maximizes this value

# Run 100 trials
study = optuna.create_study(direction='maximize')
study.optimize(lgbm_objective, n_trials=100)

# Get best hyperparameters
print(study.best_params)
```

6.3 Optuna's Smart Search

Unlike random search, Optuna uses **Tree-structured Parzen Estimator (TPE)** which:

1. **Learns from history:** Good trials inform future guesses
2. **Focuses on promising regions:** Spends more time on hyperparameters that improve results
3. **Handles correlations:** Understands that some hyperparameters interact

100 Optuna trials typically outperform 1000 random search trials.

6.4 Our Tuning Results

Model	Best CV Accuracy	Trials
LightGBM	91.2%	100
XGBoost	90.8%	100
HistGradientBoosting	90.5%	100
MLP Neural Network	89.1%	50

Chapter 7: Cross-Validation with Threshold Optimization

7.1 Why K-Fold Cross-Validation?

Splitting data once (train/test) is risky—results depend on which samples happen to be in test. K-Fold averages across multiple splits for reliable estimates.

7.2 Stratified K-Fold

"Stratified" ensures each fold has the same disease proportion as the original data:

```
Original Data: 75% healthy, 25% disease

Standard K-Fold (risky):
  Fold 1: 80% healthy, 20% disease  ← Unbalanced!
  Fold 2: 70% healthy, 30% disease  ← Unbalanced!

Stratified K-Fold (our choice):
  Fold 1: 75% healthy, 25% disease
  Fold 2: 75% healthy, 25% disease
```

7.3 Per-Fold Threshold Optimization

File: `Milestone_2/experiments/train_cv_ensemble.py`

```python
def cross_validate_with_threshold(model, X, y, cv=5):
    """CV with per-fold threshold optimization."""
    skf = StratifiedKFold(n_splits=cv, shuffle=True, random_state=42)

    for fold, (train_idx, val_idx) in enumerate(skf.split(X, y)):
        X_train_fold, X_val_fold = X[train_idx], X[val_idx]
        y_train_fold, y_val_fold = y[train_idx], y[val_idx]

        model.fit(X_train_fold, y_train_fold)
        y_proba = model.predict_proba(X_val_fold)[:, 1]

        # Find best threshold for THIS fold
        best_thresh, best_acc = 0.5, 0
        for thresh in np.arange(0.35, 0.65, 0.01):
            acc = accuracy_score(y_val_fold, (y_proba >= thresh).astype(int))
            if acc > best_acc:
                best_acc = acc
                best_thresh = thresh

        print(f"Fold {fold+1}: Best threshold = {best_thresh:.2f}, Accuracy = {best_acc:.2%
```

Why Per-Fold? Different data distributions may have different optimal thresholds. Averaging across folds gives a robust threshold estimate.

Chapter 8: Threshold Optimization

6.1 The Default Threshold Problem

Classification models output probabilities (e.g., "67% chance of disease"). To make a binary decision, we use a threshold:

Default: If probability 0.50 → "Disease"

Problem: In medicine, false negatives are catastrophic. Missing a sick patient could kill them. A false positive (healthy patient flagged for testing) is just an inconvenience.

6.2 Understanding the Stakes

False Negative (Type II Error): "You're healthy" when patient has disease. - Patient goes home, doesn't seek treatment - Condition worsens, potentially fatal - **CATASTROPHIC**

False Positive (Type I Error): "You might be at risk" when patient is healthy. - Patient gets additional testing - Tests come back negative, patient relieved - **Inconvenient but harmless**

6.3 Our Threshold Optimization

We tested multiple thresholds:

Threshold	Accuracy	Sensitivity (Recall)	Specificity	Decision
0.50	85.00%	42.10%	95.23%	Too many misses
0.40	82.50%	52.30%	91.45%	Better
0.30	79.20%	58.90%	87.32%	Good
0.25	**76.84%**	**62.77%**	83.15%	**Sweet spot**
0.20	73.10%	68.42%	78.90%	Too many false alarms

Why 0.25: We catch 63% of disease cases (vs 42% at default). The 8% accuracy drop is acceptable because we're catching 20% more sick patients.

Chapter 7: Hyperparameter Tuning

7.1 What Are Hyperparameters?

Learned parameters: What the model learns from data (tree splits, weights). **Hyperparameters:** Settings we choose BEFORE training (number of trees, tree depth).

7.2 Key XGBoost Hyperparameters

Parameter	What It Controls	Our Value
`n_estimators`	Number of trees	300
`max_depth`	How deep trees grow	4
`learning_rate`	Step size for learning	0.05
`subsample`	% of data per tree	0.8
`colsample_bytree`	% of features per tree	0.8
`scale_pos_weight`	Class imbalance correction	3.0

7.3 Our Tuning Process

Step 1: Define Search Space

```
param_distributions = {
    'n_estimators': [100, 200, 300, 400, 500],
    'max_depth': [3, 4, 5, 6],
    'learning_rate': [0.01, 0.05, 0.1],
    'subsample': [0.7, 0.8, 0.9],
}
```

Step 2: Randomized Search with 5-Fold CV

```
search = RandomizedSearchCV(
    XGBClassifier(),
    param_distributions,
    n_iter=100,      # Try 100 combinations
    cv=5,            # 5-fold cross-validation
    scoring='roc_auc' # Optimize for AUC, not accuracy
)
```

Why ROC-AUC? With 75% healthy patients, accuracy is misleading. AUC measures how well the model separates classes regardless of threshold.

Chapter 8: SHAP - Model Interpretability

8.1 The Black Box Problem

Doctors need to understand WHY a model made a prediction. "The model says you're at risk" isn't acceptable—they need to explain it to patients.

8.2 SHAP Explanations

SHAP (SHapley Additive exPlanations) explains individual predictions:

```
Base value (average): 35% risk
```

```
+ Age = 68:           +15%  (older = higher risk)
+ Systolic BP = 165:  +12%  (high BP = higher risk)
+ Smoker = Yes:       +8%
- HDL = 65:           -3%   (good cholesterol = lower risk)
= Final prediction:   75%
```

Clinical Value: - Explain predictions to patients - Verify model uses clinically sensible features - Identify which factors to target for intervention

---

PART III: INPUT PROCESSING

Chapter 7: OCR Pipeline

7.1 The Challenge

Patients bring medical reports that are: - Scanned PDFs (images of paper) - Phone photos (low quality, skewed) - Various formats and layouts

We need to extract structured data from these unstructured images.

7.2 Our Four-Stage Pipeline

File: `Milestone_2/pipeline/integrated_pipeline.py`

Stage 1: PDF/Image Handling - Digital PDFs: Extract text directly - Scanned PDFs: Convert to images, then OCR

Stage 2: Image Preprocessing

```python
def preprocess_image(self, image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    denoised = cv2.fastNlMeansDenoising(gray)
    binary = cv2.adaptiveThreshold(denoised, 255,
        cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)
    binary = self._deskew(binary)  # Fix rotation
    return binary
```

Stage 3: Text Extraction We use **Tesseract OCR** with multiple page segmentation modes for best results.

Stage 4: Field Extraction Regular expressions extract specific values:

```python
age_patterns = [
    r"age[:\s]+(\d+)\s*(?:years?|yrs?)?",
    r"(\d+)\s*(?:years?|yrs?)\s*old",
]
```

Chapter 8: API Validation (Serializers)

8.1 Why Validation Matters

Invalid input can crash our model or produce nonsensical results. We validate ALL input before processing.

8.2 Our Serializer

File: `predictions/serializers.py`

```python
class ManualInputSerializer(serializers.Serializer):
    age = serializers.IntegerField(min_value=18, max_value=120)
    systolic_bp = serializers.IntegerField(min_value=60, max_value=250)
    diastolic_bp = serializers.IntegerField(min_value=30, max_value=150)

    def validate(self, data):
        # Cross-field validation
        if data['systolic_bp'] <= data['diastolic_bp']:
            raise ValidationError("Systolic must be > diastolic")
        return data
```

Invalid requests get clear error messages:

```json
{
    "errors": {
        "age": ["Age must be at least 18 years"],
        "systolic_bp": ["Systolic must be > diastolic"]
    }
}
```

PART IV: OUTPUT PROCESSING

Chapter 9: Risk Categorization

9.1 Clinical Risk Bands

File: `Milestone_2/pipeline/integrated_pipeline.py`

```python
def categorize_risk(self, probability: float):
    if probability < 0.15:
        return "  LOW", "Routine monitoring"
    elif probability < 0.40:
        return "  MODERATE", "Lifestyle modifications advised"
    else:
        return "  HIGH", "Medical consultation recommended"
```

9.2 Why These Thresholds?

Based on **ACC/AHA Guidelines** and **Number Needed to Treat (NNT)** analysis:

Risk Level	10-Year Risk	Clinical Action
LOW	<15%	Lifestyle maintenance only
MODERATE	15-40%	Consider statin therapy
HIGH	40%	Aggressive intervention

Chapter 10: Clinical Advisor

10.1 From Prediction to Action

Knowing "65% risk" is useful, but what should the doctor DO? The Clinical Advisor translates predictions into specific, guideline-based recommendations.

File: `Milestone_2/pipeline/clinical_advisor.py`

10.2 Guidelines Implemented

Guideline	What It Covers
ACC/AHA 2017	Blood pressure management
ACC/AHA 2018	Cholesterol & statin therapy
ACC/AHA 2019	Primary prevention
WHO 2020	Physical activity

10.3 Blood Pressure Classification

```
BP_CATEGORIES = {
    "Normal": {"sbp": (0, 120), "action": "Maintain lifestyle"},
    "Elevated": {"sbp": (120, 130), "action": "Lifestyle modifications"},
    "Stage 1 HTN": {"sbp": (130, 140), "action": "Consider medication if risk  10%"},
    "Stage 2 HTN": {"sbp": (140, 180), "action": "Medication + lifestyle"},
    "HTN Crisis": {"sbp": (180, 999), "action": "IMMEDIATE evaluation"},
}
```

10.4 Lifestyle Recommendations

Personalized advice based on patient data: - **Smokers:** "Quit smoking - reduces risk 50% within 1 year" - **Exercise:** "150 minutes moderate activity per week" - **Diet:** "Limit sodium to 2,300mg/day, emphasize vegetables" - **Obesity (BMI 30):** "5-10% weight loss can reduce BP by 5-20 mmHg"

PART V: WEB APPLICATION

Chapter 11: Backend Architecture (Django REST Framework)

11.1 Project Structure

```
Milestone_3/
  cardiodetect/        # Project settings
     settings.py      # Configuration
     urls.py          # URL routing
     middleware.py    # Security middleware
  accounts/            # User authentication
     models.py        # User model
     views.py         # Auth endpoints
     serializers.py   # Validation
  predictions/         # Core ML features
     models.py        # Prediction storage
     views.py         # API endpoints
     serializers.py   # Input validation
  services/
      ml_service.py    # ML model loading
```

11.2 API Endpoints

Endpoint	Method	Purpose
`/api/auth/register/`	POST	Create account
`/api/auth/login/`	POST	Get JWT tokens
`/api/predictions/manual/`	POST	Submit health data
`/api/predictions/ocr/`	POST	Upload medical report
`/api/predictions/history/`	GET	View past predictions

Chapter 12: Frontend Architecture (Next.js & React)

12.1 Technology Stack

Technology	Purpose
Next.js 14	React framework with server-side rendering
TypeScript	Type safety
Tailwind CSS	Styling
Context API	Global state management

12.2 Authentication Context

File: `frontend/src/context/AuthContext.tsx`

```
interface AuthContextType {
    user: User | null;
    isAuthenticated: boolean;
    login: (email: string, password: string) => Promise<void>;
    logout: () => void;
}

export function AuthProvider({ children }) {
    const [user, setUser] = useState(null);

    const login = async (email, password) => {
        const response = await api.post('/auth/login/', { email, password });
        setUser(response.data.user);
        localStorage.setItem('tokens', JSON.stringify(response.data.tokens));
    };

    return (
        <AuthContext.Provider value={{ user, isAuthenticated: !!user, login, logout }}>
            {children}
        </AuthContext.Provider>
    );
}
```

12.3 Protected Routes

```
function ProtectedRoute({ children }) {
    const { isAuthenticated, isLoading } = useAuth();

    if (isLoading) return <LoadingSpinner />;
    if (!isAuthenticated) return <Navigate to="/login" />;
    return children;
}
```

PART VI: SECURITY

Chapter 13: JWT Authentication

13.1 What Is JWT?

JSON Web Token (JWT) is an open standard (RFC 7519) for securely transmitting information. Unlike session-based auth, JWT is **stateless**—the token contains all necessary information.

13.2 JWT Structure

```
eyJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjoiYWJjMTIzIn0.SIGNATURE
    Header          Payload
```

Header: Algorithm & type **Payload:** User data (user_id, role, expiration) **Signature:** Verification using secret key

13.3 Our Configuration

File: `cardiodetect/settings.py`, Lines 178-184

```
SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(hours=24),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=7),
    'ROTATE_REFRESH_TOKENS': True,
}
```

Security Note: JWTs are encoded, not encrypted. Never put passwords in the payload!

Chapter 14: Password Hashing (PBKDF2)

14.1 Why Hash Passwords?

If hackers steal the database, they get hashes—not passwords. Hashing is **one-way**; you cannot reverse it.

14.2 PBKDF2 Process

```
"MyPassword123"
    ↓ + Random Salt
    ↓ × 600,000 iterations
"pbkdf2_sha256$600000$salt$hash..."
```

Why 600,000 iterations? Makes brute-force attacks take years.

File: `accounts/models.py`, Line 24

```
user.set_password(password)  # Django handles PBKDF2 automatically
```

Chapter 15: Rate Limiting

15.1 The Attack

Without rate limiting, attackers try millions of passwords:

```
POST /login { password: "123456" }
POST /login { password: "password" }
... (millions per second)
```

15.2 Our Protection

File: `cardiodetect/middleware.py`, Lines 12-77

```
self.limits = {
    '/api/auth/login/': (20, 300),        # 20 per 5 minutes
    '/api/auth/register/': (20, 3600),    # 20 per hour
    '/api/auth/password-reset/': (10, 3600),
}
```

Exceeded? → HTTP 429 "Too Many Requests"

Chapter 16: Security Headers

16.1 Headers We Set

File: `cardiodetect/middleware.py`, Lines 79-107

Header	Value	Attack Prevented
`X-Content-Type-Options`	`nosniff`	MIME sniffing
`X-Frame-Options`	`DENY`	Clickjacking
`X-XSS-Protection`	`1; mode=block`	Cross-site scripting

APPENDIX: QUICK REFERENCE

Key File Locations

Component	File	Lines
JWT Config	`cardiodetect/settings.py`	178-184
Password Hashing	`accounts/models.py`	24
Rate Limiting	`cardiodetect/middleware.py`	12-77
ML Singleton	`services/ml_service.py`	20-36
Prediction API	`predictions/views.py`	65-162
OCR Pipeline	`Milestone_2/pipeline/integrated_pipeline.py`	70-220
Clinical Advisor	`Milestone_2/pipeline/clinical_advisor.py`	83-400

Key Metrics

Metric	Value
Detection Accuracy	91.45%
Prediction Accuracy	94.01%
Optimized Threshold	0.25
Sensitivity (Recall)	62.77%
Total Features	34
Training Samples	~16,000

Model Hyperparameters

Parameter	Value
n_estimators	300
max_depth	4
learning_rate	0.05
subsample	0.8
scale_pos_weight	3.0

END OF DOCUMENT

This document provides comprehensive understanding of every major component in CardioDetect for project defense and presentation.