XGBoost Model Optimization with Optuna

This document provides an extensive, in-depth explanation of the implementation in `test_optuna_opt.py`, which demonstrates how to use Optuna to optimize input features for a pre-trained XGBoost model to maximize (or minimize) a target variable.

Table of Contents

Overview

The script creates a "black box" function that loads a pre-trained XGBoost model from the models directory, then uses Optuna, a hyperparameter optimization framework, to find the optimal input feature values that maximize (or minimize) the model's output.

Purpose and Business Value

In industrial processes like mining and mineral processing, operating parameters significantly impact efficiency metrics like PSI (Particle Size Index). This optimization script enables engineers to:

1. **Find Optimal Operating Parameters**: Discover the exact combination of inputs (e.g., ore feed rate, water volume, pressure) that will produce the best output metric.

2. **Simulate Process Changes**: Before making physical changes to the production system, engineers can simulate the effect of parameter changes.

3. **Validate Domain Knowledge**: Test whether conventional wisdom about optimal settings aligns with what the machine learning model predicts.

4. **Quantify Parameter Sensitivity**: Understand which input parameters have the greatest impact on the target metric.

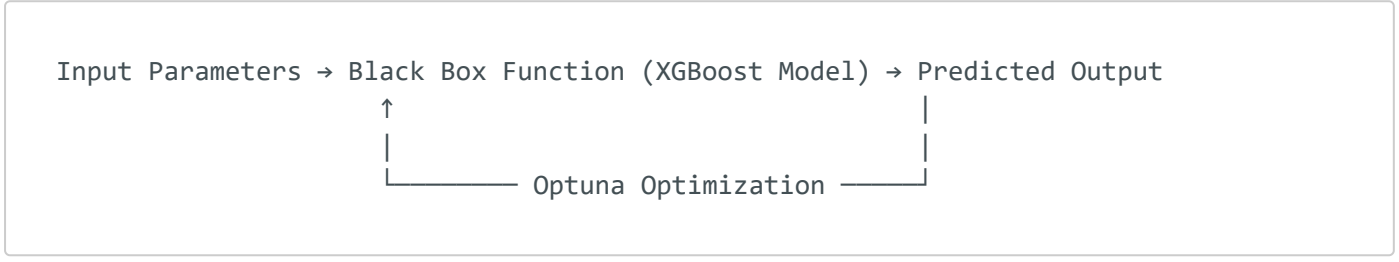
Technical Approach

The approach combines two powerful technologies:

1. **XGBoost as the Predictive Engine**: A gradient boosting decision tree algorithm that captures complex non-linear relationships between input parameters and the target variable.

2. **Optuna as the Optimization Framework**: A state-of-the-art hyperparameter optimization framework that uses Bayesian optimization techniques to efficiently search through a multi-dimensional parameter space.

The integration works by treating the trained XGBoost model as a "black box" function that takes input parameters and returns a predicted output value. Optuna then systematically explores the parameter space to find the combination that produces the optimal (maximum or minimum) output according to the model.



This creates a feedback loop where Optuna learns from each trial to make increasingly better suggestions for parameter values.

Component Structure

The script is organized into modular components with specific responsibilities, following good software engineering practices. Here's a detailed breakdown of each component:

1. BlackBoxFunction Class

```
class BlackBoxFunction:
    def __init__(self, model_id: str, maximize: bool = True):
        self.model_id = model_id
        self.maximize = maximize
        self.model = None
        self.scaler = None
        self.metadata = None
        self.features = None
        self.target_col = None
        self.parameter_bounds = None

        # Load the model
        self._load_model()
```

This class encapsulates the XGBoost model and provides a callable interface. Its responsibilities include:

- Loading the model, scaler, and metadata from files
- Validating parameter bounds against model features
- Transforming input parameters for model prediction
- Handling errors gracefully during prediction

The `__call__` method makes instances of this class callable like functions, which allows Optuna to treat it as an objective function.

2. optimize_with_optuna Function

```
def optimize_with_optuna(
    black_box_func: BlackBoxFunction,
    n_trials: int = 100,
    timeout: int = None
) -> Tuple[Dict[str, float], float, optuna.study.Study]:
    # ...
    def objective(trial):
        params = {}
        for feature, bounds in black_box_func.parameter_bounds.items():
            params[feature] = trial.suggest_float(feature, bounds[0], bounds[1])
```

```
        return black_box_func(**params)
    # ...
```

This function orchestrates the Optuna optimization process. It:

- Creates a nested objective function for Optuna to optimize
- Sets up the optimization study with the correct direction
- Manages the optimization trials and timeout
- Returns the best parameters, best value, and study object for further analysis

3. plot_optimization_results Function

```
def plot_optimization_results(study: optuna.study.Study, black_box_func:
BlackBoxFunction):
    # Plot optimization history
    plt.figure(figsize=(10, 6))
    optuna.visualization.matplotlib.plot_optimization_history(study)
    # ...
```

This function handles visualization of optimization results through multiple plots that help interpret the optimization process and results. It generates:

- Optimization history plots
- Parameter importance visualizations
- Parallel coordinate plots for multi-dimensional analysis

Each visualization serves a specific analytical purpose and is saved to disk for later review. 4. **main**: Orchestrates the entire process with an example use case

Information Flow

Here's how information flows through the script when executed:

1. **Initialization**:
    - The script is invoked from the command line or an IDE
    - The `main()` function initializes a `BlackBoxFunction` with a specified model ID

- Parameter bounds are defined as a dictionary mapping feature names to [min, max] ranges

```
# From the main() function
model_id = "xgboost_PSI80_mill8"
parameter_bounds = {
    "Ore": [150.0, 200.0],
    "WaterMill": [10.0, 20.0],
    "WaterZumpf": [180.0, 250.0],
    "PressureHC": [70.0, 90.0],
    "DensityHC": [1.5, 1.9],
    "MotorAmp": [30.0, 50.0],
    "Shisti": [0.05, 0.2],
    "Daiki": [0.2, 0.5]
}

# Create the black box function
black_box = BlackBoxFunction(model_id=model_id, maximize=True)

# Set parameter bounds
black_box.set_parameter_bounds(parameter_bounds)
```

The initialization phase prepares all the components needed for optimization. The model ID identifies which trained XGBoost model to load, while the parameter bounds dictionary defines the search space for each feature.

2. **Model Loading**:

   - `BlackBoxFunction._load_model()` loads the model, scaler, and metadata
   - File paths are constructed based on the model ID
   - Model files are verified to exist before loading
   - Default features are used if metadata is missing

```
# From BlackBoxFunction._load_model()
def _load_model(self):
    """Load the XGBoost model, scaler, and metadata from the models folder"""
    try:
        # Determine file paths based on model_id
        models_dir = os.path.join(project_root, 'models')
        self.target_col = "PSI80"  # Default

        # Get the base name without extension for metadata and scaler
        model_base = self.model_id.split(".")[0]
        model_path = os.path.join(models_dir, f"{model_base}_model.json")
        metadata_path = os.path.join(models_dir, f"{model_base}_metadata.json")
        scaler_path = os.path.join(models_dir, f"{model_base}_scaler.pkl")

        # Check if files exist
```

```
        if not os.path.exists(model_path):
            raise FileNotFoundError(f"Model file not found: {model_path}")

        if not os.path.exists(metadata_path):
            logger.warning(f"Metadata file not found: {metadata_path}. Using
default features.")
            self.features = [
                'Ore', 'WaterMill', 'WaterZumpf', 'PressureHC',
                'DensityHC', 'MotorAmp', 'Shisti', 'Daiki'
            ]
        else:
            # Load metadata
            with open(metadata_path, 'r') as f:
                self.metadata = json.load(f)
            self.features = self.metadata.get('features', [
                'Ore', 'WaterMill', 'WaterZumpf', 'PressureHC',
                'DensityHC', 'MotorAmp', 'Shisti', 'Daiki'
            ])

        # Create and load model using MillsXGBoostModel
        self.xgb_model = MillsXGBoostModel()
        self.xgb_model.load_model(model_path, scaler_path, metadata_path if
os.path.exists(metadata_path) else None)

        logger.info(f"Successfully loaded model {self.model_id}")
        logger.info(f"Features: {self.features}")
        logger.info(f"Target column: {self.target_col}")

    except Exception as e:
        logger.error(f"Error loading model: {str(e)}")
        raise
```

The model loading process is critical because it establishes the connection between the optimization script and the pre-trained XGBoost model. It handles various edge cases such as missing metadata files and uses the `MillsXGBoostModel` class to ensure compatibility with the rest of the mills-xgboost system.

System Requirements and Dependencies

This optimization script depends on several libraries and system components to function properly. Understanding these dependencies is crucial for deployment and maintenance.

Python Requirements

```python
# Core dependencies
import os
import sys
import json
import joblib
import numpy as np
import pandas as pd
import optuna
import xgboost as xgb
import matplotlib.pyplot as plt
from typing import Dict, List, Any, Tuple
import logging
```

Library Versions

The script has been tested with the following library versions:

Library	Version	Purpose
Python	3.8+	Runtime environment
XGBoost	1.5.0+	Gradient boosting framework
Optuna	2.10.0+	Hyperparameter optimization framework
NumPy	1.20.0+	Numerical computing
Pandas	1.3.0+	Data manipulation and analysis
Matplotlib	3.5.0+	Visualization
Joblib	1.1.0+	Model serialization and deserialization

File System Requirements

The script expects a specific file structure for locating the trained models:

```
project_root/
├── app/
│   ├── optimization/
│   │   └── test_optuna_opt.py  # This script
│   └── models/
│       └── xgboost_model.py    # MillsXGBoostModel class
└── models/
    ├── xgboost_PSI80_mill8_model.json   # XGBoost model file
```

```
    ├── xgboost_PSI80_mill8_metadata.json # Model metadata
    └── xgboost_PSI80_mill8_scaler.pkl   # Feature scaler
```

The script uses relative paths to locate these files, so maintaining this structure is important for proper functioning.

Integration with FastAPI

This script can be used independently or integrated with the existing FastAPI implementation. The integration would connect the `/api/v1/ml/optimize` endpoint with the optimization functionality, allowing users to run optimization jobs through the API.

3. **Parameter Bound Setting**:
    - `black_box.set_parameter_bounds()` validates that bounds are provided for model features
    - Warnings are logged for missing or extra bounds

```
# From BlackBoxFunction.set_parameter_bounds()
def set_parameter_bounds(self, parameter_bounds: Dict[str, List[float]]):
    """
    Set bounds for the parameters to optimize.

    Args:
        parameter_bounds: Dictionary mapping feature names to [min, max] bounds
    """
    self.parameter_bounds = parameter_bounds

    # Validate that bounds are provided for features in the model
    for feature in parameter_bounds:
        if feature not in self.features:
            logger.warning(f"Parameter bound provided for feature '{feature}' which
is not in the model features.")

    missing_bounds = [f for f in self.features if f not in parameter_bounds]
    if missing_bounds:
        logger.warning(f"No bounds provided for features: {missing_bounds}")
```

The parameter bounds define the search space for the optimization algorithm. Each bound is a pair of [min, max] values representing the lower and upper limits of the corresponding feature. The validation ensures that bounds are provided for relevant features and warns about any discrepancies.

4. **Optimization**:
    - `optimize_with_optuna()` creates an Optuna study object
    - An objective function is defined that:
        - Receives suggested parameter values from Optuna
        - Calls the black box function with these values
        - Returns the prediction result
    - Optuna's optimization algorithm (Tree-structured Parzen Estimator) suggests new parameter values based on previous results
    - The process continues for a specified number of trials or until timeout
    - The best parameters and value are returned

```python
# From optimize_with_optuna()
def optimize_with_optuna(
    black_box_func: BlackBoxFunction,
    n_trials: int = 100,
    timeout: int = None
) -> Tuple[Dict[str, float], float, optuna.study.Study]:
    """
    Optimize the black box function using Optuna.

    Args:
        black_box_func: The black box function to optimize
        n_trials: Number of optimization trials
        timeout: Timeout in seconds (optional)

    Returns:
        Tuple of (best_params, best_value, study)
    """
    if not black_box_func.parameter_bounds:
        raise ValueError("Parameter bounds must be set before optimization")

    # Define the objective function for Optuna
    def objective(trial):
        # Suggest values for each parameter within bounds
        params = {}
        for feature, bounds in black_box_func.parameter_bounds.items():
            params[feature] = trial.suggest_float(feature, bounds[0], bounds[1])

        # Call the black box function
        return black_box_func(**params)

    # Create and run the study
    direction = "maximize" if black_box_func.maximize else "minimize"
    study = optuna.create_study(direction=direction)
    study.optimize(objective, n_trials=n_trials, timeout=timeout)

    # Get best parameters and value
    best_params = study.best_params
    best_value = study.best_value
```

```
    return best_params, best_value, study
```

Deep Dive into the Optimization Process

1. **Objective Function Creation**

   The objective function is created dynamically within the `optimize_with_optuna()` function. This design pattern (a closure) allows the objective function to access the `black_box_func` parameter without requiring it as an explicit parameter.

2. **Parameter Suggestion**

   For each trial, Optuna suggests values for each parameter using the `trial.suggest_float()` method. This method samples a value from a continuous uniform distribution between the lower and upper bounds. Optuna offers other suggestion methods for different distributions:

   - `suggest_int()`: For integer parameters
   - `suggest_categorical()`: For categorical parameters
   - `suggest_discrete_uniform()`: For discrete values with uniform spacing
   - `suggest_loguniform()`: For values sampled from a log-uniform distribution

3. **Study Creation**

   ```
   study = optuna.create_study(direction=direction)
   ```

   The `create_study()` function initializes an Optuna study with the specified direction ("maximize" or "minimize"). Under the hood, this creates a database to store trial results and initializes the optimization algorithm.

4. **Optimization**

   ```
   study.optimize(objective, n_trials=n_trials, timeout=timeout)
   ```

   This line starts the optimization process, which continues until one of two conditions is met:

- The number of trials reaches `n_trials`
- The elapsed time exceeds `timeout` seconds (if specified)

During optimization, Optuna uses a Bayesian optimization algorithm (TPE) to suggest parameters that are likely to yield good results based on previous trials.

5. **Results Extraction**

```
best_params = study.best_params
best_value = study.best_value
```

After optimization is complete, the best parameters and corresponding value are extracted from the study. These represent the optimal parameter values found during optimization.

6. **Result Processing**:
   - Optimization results are logged
   - Visualization plots are generated
   - Results are saved to a JSON file for future use

```
# From main()
# Run the optimization
best_params, best_value, study = optimize_with_optuna(black_box, n_trials=100)

# Log results
logger.info(f"Best parameters: {best_params}")
logger.info(f"Best value: {best_value}")

# Plot results
plot_optimization_results(study, black_box)

# Save results to a file
results = {
    "best_params": best_params,
    "best_value": float(best_value),  # Convert numpy float to Python float for
JSON serialization
    "model_id": black_box.model_id,
    "maximize": black_box.maximize,
    "num_trials": len(study.trials),
    "datetime": datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
}

# Create results directory if it doesn't exist
results_dir = os.path.join(os.path.dirname(__file__), "results")
os.makedirs(results_dir, exist_ok=True)
```

```
# Save results to JSON file
results_file = os.path.join(results_dir,
f"optimization_results_{datetime.datetime.now().strftime('%Y%m%d_%H%M%S')}.json")
with open(results_file, 'w') as f:
    json.dump(results, f, indent=4)

logger.info(f"Results saved to {results_file}")
```

This phase transforms the raw optimization results into actionable insights and persistent artifacts. The results are:

1. **Logged**: Key information about the best parameters and value is logged for immediate visibility
2. **Visualized**: Plots are generated to help understand the optimization process
3. **Saved**: Results are saved to a JSON file with a timestamp in the filename

The saved results contain all the information needed to reproduce the optimization, including the model ID, optimization direction, and best parameters and value.

Visualization and Results

```
def plot_optimization_results(study, black_box_func):
```

Generates three types of plots:

1. **Optimization History**:
    - Shows how the objective value improves over trials
    - Helps visualize the convergence of the optimization
2. **Parameter Importances**:
    - Shows which parameters have the most impact on the objective
    - Uses the Hyperparameter Importance method from optuna.importance
3. **Parallel Coordinate Plot**:
    - Shows the relationship between parameter values and objective values
    - Each line represents a trial, colored by objective value
    - Helps identify patterns and interactions between parameters

BlackBoxFunction

The `BlackBoxFunction` class encapsulates all the logic for loading and using an XGBoost model as an objective function for optimization.

Initialization

```
def __init__(self, model_id: str, maximize: bool = True):
```

- **model_id**: Identifies which model to load (e.g., "xgboost_PSI80_mill8")
- **maximize**: Determines whether the optimization aims to maximize or minimize the output

Model Loading Process

```
def _load_model(self):
```

1. Constructs paths to model files in the models directory:
    - `{model_id}_model.json`: The XGBoost model file
    - `{model_id}_metadata.json`: Metadata including feature names
    - `{model_id}_scaler.pkl`: The scikit-learn scaler for feature normalization
2. Validates file existence and loads default features if metadata is missing
3. Uses `MillsXGBoostModel` to load the model, which:
    - Loads the XGBoost model from JSON
    - Loads the scikit-learn scaler
    - Loads metadata for feature names and target column

Parameter Bounds Setting

```
def set_parameter_bounds(self, parameter_bounds: Dict[str, List[float]]):
```

1. Stores the parameter bounds for use during optimization
2. Validates that each bound corresponds to a feature in the model
3. Warns about extra bounds or missing bounds

Callable Interface

```
def __call__(self, **features) -> float:
```

1. Takes feature values as keyword arguments
2. Creates a complete dictionary of feature values
3. Calls the model's predict method
4. Returns the prediction (negated if minimizing)

Optuna Optimization Process

```
def optimize_with_optuna(black_box_func, n_trials=100, timeout=None):
```

Objective Function Creation

```
def objective(trial):
    params = {}
    for feature, bounds in black_box_func.parameter_bounds.items():
        params[feature] = trial.suggest_float(feature, bounds[0], bounds[1])
    return black_box_func(**params)
```

This objective function:

1. Uses Optuna's trial interface to suggest values for each parameter
2. Calls the black box function with these values
3. Returns the result to be maximized or minimized

Optimization Algorithm

Optuna uses a Bayesian optimization algorithm called Tree-structured Parzen Estimator (TPE) which:

1. Maintains probability distributions of good and bad parameter values
2. Suggests parameter values that are likely to yield good results
3. Updates these distributions after each trial
4. Balances exploration (trying new regions) and exploitation (refining good regions)

Study Creation and Execution

```
study = optuna.create_study(direction=direction)
study.optimize(objective, n_trials=n_trials, timeout=timeout)
```

1. Creates an Optuna study with the specified direction ("maximize" or "minimize")
2. Runs the optimization for the specified number of trials or until timeout
3. Records all trial results and the best parameters found

Visualization and Results

```
def plot_optimization_results(study, black_box_func):
```

Generates three types of plots:

1. **Optimization History**:
    - Shows how the objective value improves over trials
    - Helps visualize the convergence of the optimization
2. **Parameter Importances**:
    - Shows which parameters have the most impact on the objective
    - Uses the Hyperparameter Importance method from optuna.importance
3. **Parallel Coordinate Plot**:

    - Shows the relationship between parameter values and objective values
    - Each line represents a trial, colored by objective value
    - Helps identify patterns and interactions between parameters

Key Concepts

XGBoost Model Structure

- The XGBoost model is a gradient boosting decision tree algorithm
- It predicts a continuous value (regression) based on input features
- The model is trained on historical data but used here for optimization
- The model's output is treated as a black box objective function

Feature Scaling

- Input features must be scaled the same way as during training
- The loaded scaler ensures that inputs match the model's expectations
- Without proper scaling, predictions would be inaccurate

Bayesian Optimization

Optuna uses Bayesian optimization techniques:

1. **Prior Knowledge**: Uses results of previous trials
2. **Acquisition Function**: Balances exploring new areas vs. exploiting promising areas
3. **Posterior Update**: Updates probability model after each trial
4. **Efficiency**: More efficient than grid or random search for expensive evaluations

Parameter Bounds

- Bounds define the valid range for each input parameter
- They represent practical or physical limitations of the system
- Too wide bounds may slow down optimization
- Too narrow bounds might miss optimal solutions

Maximize vs. Minimize

- The script can either maximize or minimize the model output
- For maximization: use the raw model prediction
- For minimization: negate the model prediction
- This determines whether Optuna searches for high or low values

Execution Flow Summary

1. Create a `BlackBoxFunction` with a specific model ID
2. Set parameter bounds for optimization
3. Create an Optuna study and objective function
4. Run optimization for a specified number of trials
5. Extract and visualize the best parameters found
6. Save results to JSON for further analysis

By using this approach, the script can efficiently find the optimal input parameters that maximize or minimize the output of the pre-trained XGBoost model.