Understanding Optimization with Optuna

This document provides a detailed explanation of the optimization framework that uses Optuna to find optimal parameters for XGBoost models. The framework includes the `optimize_with_optuna` function, visualization tools, and data export capabilities.

What is Optuna?

Optuna is an open-source hyperparameter optimization framework specifically designed for machine learning. It automates the process of finding the best set of hyperparameters for your models by efficiently searching through the parameter space.

The `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]:
    """
    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)
    """
```

Parameters

1. **black_box_func**: An instance of `BlackBoxFunction` that wraps an XGBoost model
2. **n_trials**: Number of optimization trials to run (default: 100)
3. **timeout**: Optional timeout in seconds

Return Value

The function returns a tuple containing:

1. Best parameters found during optimization
2. Best value achieved with those parameters
3. The complete Optuna study object

Step-by-Step Explanation

1. Parameter Bounds Validation

```
if not black_box_func.parameter_bounds:
    raise ValueError("Parameter bounds must be set before optimization")
```

Before starting optimization, the function checks if parameter bounds have been set. These bounds define the search space for each parameter.

Example parameter bounds:

```
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]
}
```

2. Defining the Objective Function

```
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)
```

This nested function defines what Optuna will optimize. For each trial:

1. Optuna suggests values for each parameter within the defined bounds
2. These values are collected into a `params` dictionary
3. The black box function is called with these parameters

Example data flow for a single trial:

```
# Optuna suggests these values for trial #1:
params = {
    "Ore": 175.3,
    "WaterMill": 15.7,
    "WaterZumpf": 210.2,
    "PressureHC": 82.1,
    "DensityHC": 1.72,
    "MotorAmp": 42.5,
    "Shisti": 0.12,
    "Daiki": 0.35
}

# The black box function is called with these parameters
result = black_box_func(**params)  # Returns something like 85.7 (the predicted
PSI80 value)
```

3. Creating and Running the Study

```
# 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)
```

Here's what happens:

1. The optimization direction is set based on whether we want to maximize or minimize the objective
2. An Optuna study is created with this direction
3. The study runs the optimization for the specified number of trials or until timeout

Example study creation:

```
# If black_box_func.maximize is True:
direction = "maximize"
study = optuna.create_study(direction="maximize")
# The study will try to find parameters that give the highest possible output
```

4. Retrieving Results

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

return best_params, best_value, study
```

After optimization completes:

1. The best parameters found during all trials are retrieved
2. The best value achieved with those parameters is retrieved
3. Both are returned along with the complete study object

Example results:

```
# After 50 trials, these might be the results:
best_params = {
    "Ore": 192.7,
    "WaterMill": 18.3,
    "WaterZumpf": 225.6,
    "PressureHC": 85.9,
    "DensityHC": 1.85,
    "MotorAmp": 45.2,
    "Shisti": 0.15,
    "Daiki": 0.42
}
best_value = 92.3  # The highest PSI80 value achieved
```

Behind the Scenes: How Optuna Works

Optuna uses a technique called Tree-structured Parzen Estimator (TPE) to intelligently search the parameter space. Here's what happens during optimization:

1. **Initial exploration**: Optuna starts by randomly sampling the parameter space to build an initial understanding

   ```
   # Early trials might look like:
   Trial #1: {"Ore": 175.3, "WaterMill": 15.7, ...} → Result: 82.5
   Trial #2: {"Ore": 160.1, "WaterMill": 12.3, ...} → Result: 78.9
   Trial #3: {"Ore": 190.5, "WaterMill": 19.2, ...} → Result: 88.7
   ```

2. **Learning from history**: As trials accumulate, Optuna builds probability models of which parameter values lead to good results

3. **Adaptive sampling**: Later trials focus more on promising regions of the parameter space

```
# Later trials might cluster around promising values:
Trial #20: {"Ore": 191.2, "WaterMill": 18.9, ...} → Result: 90.1
Trial #21: {"Ore": 192.5, "WaterMill": 18.5, ...} → Result: 91.8
Trial #22: {"Ore": 192.7, "WaterMill": 18.3, ...} → Result: 92.3
```

4. **Pruning**: Optuna can also stop unpromising trials early to save computation time

Complete Data Flow Example

Let's trace through a complete example:

1. **Setup**:

```
# Create black box function with model
black_box = BlackBoxFunction(model_id="xgboost_PSI80_mill8", maximize=True)

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

2. **Call optimize_with_optuna**:

```
best_params, best_value, study = optimize_with_optuna(
    black_box_func=black_box,
    n_trials=50
)
```

3. **Inside optimize_with_optuna**:

    - The objective function is defined
    - A study is created with direction="maximize"
    - For each trial (50 total):

```
# Trial #1
params = {
    "Ore": 175.3,         # Suggested by Optuna
    "WaterMill": 15.7,    # Suggested by Optuna
    # ... other parameters
}

# Call black_box with these parameters
result = black_box(**params)

# Inside black_box.__call__:
input_data = {
    "Ore": 175.3,
    "WaterMill": 15.7,
    # ... other parameters
}
prediction = xgb_model.predict(input_data)[0]  # e.g., 82.5
return prediction  # Since maximize=True
```

4. **After all trials**:

```
# Study contains all trial information
best_params = {
    "Ore": 192.7,
    "WaterMill": 18.3,
    # ... other parameters
}
best_value = 92.3

# Return these values
return best_params, best_value, study
```

Understanding the Black Box Function

The `BlackBoxFunction` class wraps an XGBoost model and provides a callable interface for Optuna to optimize. Here's how it works:

```
def __call__(self, **features) -> float:
    """
    Predict the target value based on the provided features.

    Args:
        features: Feature values as keyword arguments

    Returns:
        float: The predicted value
    """
    if not self.xgb_model:
        raise ValueError("Model not loaded")

    try:
        # Create a dictionary with all features
        input_data = {feature: features.get(feature, 0.0) for feature in
self.features}

        # Make prediction
        prediction = self.xgb_model.predict(input_data)[0]

        # Return prediction (negated if minimizing)
        return prediction if self.maximize else -prediction

    except Exception as e:
        logger.error(f"Error in prediction: {str(e)}")
        # Return a very bad value in case of error
        return -1e9 if self.maximize else 1e9
```

When Optuna calls this function with a set of parameters:

1. It creates a dictionary with all required features
2. It makes a prediction using the XGBoost model
3. It returns the prediction (or its negative if minimizing)

Key Concepts to Understand

1. **Black Box Function**: A function that takes input parameters and returns a value to optimize. You don't need to know its internal workings.

2. **Parameter Space**: The range of possible values for each parameter, defined by the parameter bounds.

3. **Trial**: A single evaluation of the objective function with a specific set of parameters.

4. **Study**: The complete optimization process, containing all trials and their results.

5. **Direction**: Whether to maximize or minimize the objective function.

6. **Objective Function**: The function that Optuna tries to optimize by suggesting different parameter values.

Practical Applications

This optimization approach is particularly useful for:

1. **Process optimization**: Finding the optimal mill settings to maximize PSI80
2. **Resource allocation**: Determining the best combination of inputs for efficient operation
3. **What-if analysis**: Exploring how different parameter combinations affect the output

Visualization and Analysis

After optimization, you can use Optuna's visualization tools to analyze the results:

```
# Plot optimization history
optuna.visualization.matplotlib.plot_optimization_history(study)

# Plot parameter importances
optuna.visualization.matplotlib.plot_param_importances(study)

# Plot parallel coordinate plot
optuna.visualization.matplotlib.plot_parallel_coordinate(study)
```

These visualizations help you understand:

- How the optimization progressed over time
- Which parameters had the most impact on the result
- How different parameter combinations relate to the objective value

Exporting Trial Data for Analysis

You can export all trial data to a CSV file for further analysis using the `export_study_to_csv` function:

```
def export_study_to_csv(study: optuna.study.Study, save_path: str) -> pd.DataFrame:
    """
    Export the Optuna study trials to a DataFrame and save as CSV.

    Args:
        study: The completed Optuna study
        save_path: Path to save the CSV file

    Returns:
        DataFrame containing the study trials data
    """
    # Create a list to hold all trial data
    trials_data = []

    # Extract trial information
    for i, trial in enumerate(study.trials):
        # Get basic trial info
        trial_dict = {
            "trial_number": i,
            "value": trial.value,  # The objective value
            "state": trial.state,  # COMPLETE, PRUNED, etc.
            "datetime_start": trial.datetime_start,
            "datetime_complete": trial.datetime_complete,
            "duration": (trial.datetime_complete -
trial.datetime_start).total_seconds() if trial.datetime_complete else None
        }

        # Add parameters used in this trial
        for param_name, param_value in trial.params.items():
            trial_dict[f"param_{param_name}"] = param_value

        # Add intermediate values if any (for trials that use pruning)
        for step, intermediate_value in trial.intermediate_values.items():
            trial_dict[f"intermediate_{step}"] = intermediate_value

        trials_data.append(trial_dict)

    # Convert to DataFrame
    trials_df = pd.DataFrame(trials_data)

    # Save to CSV
    trials_df.to_csv(save_path, index=False)

    return trials_df
```

The CSV file contains important information about each trial:

- **trial_number**: Sequential ID of each trial
- **value**: The objective value achieved with the parameters
- **state**: Trial state (COMPLETE, PRUNED, etc.)
- **datetime_start/datetime_complete**: Timestamps for the start and end of each trial
- **duration**: Time taken for each trial in seconds
- **param_X columns**: The parameter values tried for each input feature
- **intermediate_X columns**: Any intermediate values recorded during the trial

File Organization and Logging

The optimization framework is organized with a clear structure for logs and results:

Directory Structure

```
app/optimization/
├── logs/                              # Directory for log files
│   └── optuna_optimization_[model_id].log  # Log file for each model
├── optimization_results/              # Directory for optimization outputs
│   ├── optimization_history.png       # Plot of optimization progress
│   ├── parameter_importances.png      # Plot of parameter importance
│   ├── parallel_coordinate.png        # Parallel coordinates visualization
│   ├── optimization_results.json      # Best parameters and values
│   └── optuna_trials.csv              # Detailed data for all trials
└── test_optuna_opt.py                 # Main script for optimization
```

Logging Configuration

The optimization process logs both to console and to a dedicated log file:

```
def setup_logger(log_file=None):
    """Set up logger with file and console handlers"""
    logger = logging.getLogger(__name__)
    logger.setLevel(logging.INFO)
    logger.handlers = []  # Clear existing handlers

    # Create formatter
    formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %
(message)s')

    # Create console handler
    console_handler = logging.StreamHandler()
    console_handler.setFormatter(formatter)
    logger.addHandler(console_handler)

    # Create file handler if log_file is specified
    if log_file:
        file_handler = logging.FileHandler(log_file)
        file_handler.setFormatter(formatter)
        logger.addHandler(file_handler)

    return logger
```

The log file includes:

- Model loading and initialization
- Parameter boundaries
- Optimization progress
- Best parameters and values found
- Trial statistics

Conclusion

The `optimize_with_optuna` function provides a powerful way to find optimal parameter settings for your XGBoost model without having to manually test different combinations. Optuna intelligently explores the parameter space to find the best possible configuration within the given constraints.

With the addition of dedicated logging, organized file outputs, and trial data export to CSV, the framework now provides comprehensive tools for both optimization and analysis of the optimization process.