5_hyperparameter_tuning

August 25, 2025

[ ]:
```
#!/usr/bin/env python
# coding: utf-8
```

5. Final Model Optimization: Hyperparameter Tuning

Author: Angela Davis **Date:** July 2, 2025

Workflow Overview

This notebook represents the final, crucial step in our modeling pipeline: optimizing the performance of our chosen model architecture. We take the best-performing model (XGBoost) and the optimal feature set identified in the previous notebook and fine-tune its hyperparameters to maximize its predictive accuracy.

The workflow is as follows: 1. **Load Final Feature Set:** Load the featurized dataset and the curated list of selected features from notebook 4, ensuring a consistent data foundation. 2. **Prepare Data:** Apply the same training/testing split and scaling logic used previously to prepare the data for modeling. 3. **Define Hyperparameter Search Space:** Define a comprehensive search space for the key hyperparameters of the XGBoost model. 4. **Execute Randomized Search:** Employ `RandomizedSearchCV` with cross-validation to efficiently search for the optimal parameter combination within the defined space. 5. **Train and Evaluate Final Model:** Train a new XGBoost model using the best hyperparameters found during the search and evaluate its performance on the held-out test set. 6. **Benchmark Against Baseline:** Compare the tuned model's performance against the untuned, default-parameter model to quantify the value added by tuning. 7. **Save Production-Ready Model:** Serialize the final, tuned XGBoost model, making it ready for deployment in a production environment or for use in prediction scripts.

[ ]:
```
# --- Professionalized Imports and Setup ---
import os, sys
import pandas as pd
import numpy as np
import json
import joblib
from xgboost import XGBRegressor
import shap
import matplotlib.pyplot as plt
from IPython.display import display
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint, uniform
```

```
# --- Define Project Root for Robust Pathing ---
try:
    # Assumes the script is in the 'notebooks' directory
    PROJECT_ROOT = os.path.abspath(os.path.join(os.path.dirname(__file__), '..
 ↪'))
except NameError:
    # Fallback for interactive environments (Jupyter, VSCode)
    PROJECT_ROOT = os.path.abspath(os.path.join(os.getcwd()))

# Add the 'src' directory to the Python path
SRC_PATH = os.path.join(PROJECT_ROOT, 'src')
if SRC_PATH not in sys.path:
    sys.path.insert(0, SRC_PATH)

from utils import (
    setup_environment,
    load_or_process_dataframe,
    save_plot,
    style_df,
    prepare_data_for_modeling,
    log_and_print
)
from modeling import split_data, scale_features, evaluate_model
from viz import plot_parity_logscale

# --- Setup environment and paths ---
setup_environment()
CACHE_PATH = os.path.join(PROJECT_ROOT, 'data', 'processed', 'featurized.
 ↪parquet')
SELECTED_FEATURES_PATH = os.path.join(PROJECT_ROOT, 'data', 'processed',␣
 ↪'selected_features_xgb.json')
FINAL_MODEL_PATH = os.path.join(PROJECT_ROOT, 'models', 'tuned_xgboost_model.
 ↪joblib')
PLOTS_DIR = os.path.join(PROJECT_ROOT, 'plots', '5_hyperparameter_tuning')
os.makedirs(PLOTS_DIR, exist_ok=True)

PARITY_PLOT_PATH = os.path.join(PLOTS_DIR, 'tuned_xgb_model_parity_plot.pdf')
SHAP_PLOT_PATH = os.path.join(PLOTS_DIR, 'tuned_xgb_model_shap_summary.pdf')
COMPARISON_TABLE_PATH = os.path.join(PLOTS_DIR, 'final_model_comparison.csv')

# --- Load featurized data using robust utility ---
df = load_or_process_dataframe(cache_path=CACHE_PATH, project_root=PROJECT_ROOT)
log_and_print(f"Featurized dataframe shape: {df.shape}")
```

```
Loaded cached DataFrame from c:\Users\angel\Thermal-Conductivity-
ML\data\processed\featurized.parquet (parquet)
Loaded cached DataFrame from c:\Users\angel\Thermal-Conductivity-
```

```
ML\data\processed\featurized.parquet
Featurized dataframe shape: (757, 177)
```

1. Load and Prepare Final Data

We begin by loading the complete featurized dataset and, most importantly, the final list of selected features that was curated and validated in the previous notebook. This ensures that we are tuning our model on the exact feature set that was proven to be most effective. All subsequent steps—splitting and scaling—are identical to the previous notebooks to maintain consistency.

[ ]:
```
# Prepare data for modeling
X, y = prepare_data_for_modeling(df, target_col='thermal_conductivity')
```

```
Imputing NaN values in columns: ['mp_density', 'mp_volume', 'mp_band_gap',
'mp_energy_above_hull', 'jarvis_band_gap', 'jarvis_formation_energy', 'density']
```

[ ]:
```
# Load the final list of selected features from the previous notebook.
# We no longer need the conditional logic for cluster features, as that
# hypothesis was tested and rejected in notebook 4.
with open(SELECTED_FEATURES_PATH, 'r') as f:
    selected_features = json.load(f)

# Ensure all selected features are present in the dataframe
available_features = [f for f in selected_features if f in X.columns]
if len(available_features) != len(selected_features):
    log_and_print("Warning: Some selected features were not found in the␣
 ↪dataframe.")

X_selected = X[available_features]

log_and_print(f"Loaded and applied {len(available_features)} selected features.
 ↪")
```

```
Warning: Some selected features were not found in the dataframe.
Loaded and applied 78 selected features.
```

[ ]:
```
# Split and scale the data
X_train, X_test, y_train_log, y_test_log, y_train, y_test =␣
 ↪split_data(X_selected, y)
X_train_scaled, X_test_scaled, scaler = scale_features(X_train, X_test)

log_and_print(f"Data is ready for hyperparameter tuning. X_train_scaled shape:␣
 ↪{X_train_scaled.shape}")
```

```
Data is ready for hyperparameter tuning. X_train_scaled shape: (605, 78)
```

2. Hyperparameter Tuning with RandomizedSearchCV

We define a parameter grid and use `RandomizedSearchCV` to efficiently explore the most promising hyperparameter combinations for the XGBoost model.

[ ]:
```
# Define the parameter grid for XGBoost
xgb_param_dist = {
    'n_estimators': randint(100, 1000),
    'learning_rate': uniform(0.01, 0.3),
    'max_depth': randint(3, 15),
    'subsample': uniform(0.6, 0.4), # Note: subsample + colsample_bytree should␣
 ↪not be > 1
    'colsample_bytree': uniform(0.6, 0.4),
    'gamma': uniform(0, 0.5),
    'reg_alpha': uniform(0, 1),
    'reg_lambda': uniform(1, 2)
}

# Initialize the XGBoost regressor
xgb_reg = XGBRegressor(random_state=42, n_jobs=-1)

# Initialize RandomizedSearchCV for XGBoost
xgb_random_search = RandomizedSearchCV(
    estimator=xgb_reg,
    param_distributions=xgb_param_dist,
    n_iter=100,
    cv=10,
    scoring='neg_mean_squared_error',
    verbose=1,
    n_jobs=-1,
    random_state=42
)
```

[ ]:
```
# Fit the random search model
log_and_print("Starting RandomizedSearchCV for XGBoost...")
xgb_random_search.fit(X_train_scaled, y_train_log)
log_and_print("XGBoost search complete.")

# Get the best XGBoost estimator
best_xgb_tuned = xgb_random_search.best_estimator_

# Evaluate the tuned XGBoost model
tuned_xgb_results = evaluate_model(best_xgb_tuned, X_test_scaled, y_test_log,␣
 ↪y_test)
tuned_xgb_log_df = pd.DataFrame([{**tuned_xgb_results['log'], 'Model': 'Tuned␣
 ↪XGBoost'}])

log_and_print("\nTuned XGBoost Model Performance (log scale):")
display(style_df(tuned_xgb_log_df[['Model', 'r2', 'mae', 'rmse']]))
```

```
Starting RandomizedSearchCV for XGBoost…
Fitting 10 folds for each of 100 candidates, totalling 1000 fits
XGBoost search complete.
```

```
Tuned XGBoost Model Performance (log scale):
```

```
<pandas.io.formats.style.Styler at 0x2cc25172ad0>
```

3. Compare with Baseline and Finalize

To provide a clear measure of the value gained from hyperparameter tuning, we'll first train a baseline XGBoost model using its default parameters on the exact same data split. We will then compare it against our tuned XGBoost model.

3.1. Train and Evaluate Baseline XGBoost Model

[ ]:
```
# --- PRE-TUNING MODEL: Selected features, default XGBoost ---
pre_tuning_xgb = XGBRegressor(random_state=42, n_jobs=-1)
pre_tuning_xgb.fit(X_train_scaled, y_train_log)
pre_tuning_results = evaluate_model(pre_tuning_xgb, X_test_scaled, y_test_log,␣
 ↪y_test)
pre_tuning_log_df = pd.DataFrame([{**pre_tuning_results['log'], 'Model':␣
 ↪'Pre-Tuning XGB'}])

log_and_print("Pre-Tuning Model Performance (log scale):")
display(style_df(pre_tuning_log_df[['Model', 'r2', 'mae', 'rmse']]))
```

```
Pre-Tuning Model Performance (log scale):
```

```
<pandas.io.formats.style.Styler at 0x2cc25173750>
```

3.2. Performance Comparison

[ ]:
```
# --- UPDATE COMPARISON TABLES ---
comparison_log_df = pd.concat([pre_tuning_log_df, tuned_xgb_log_df]).
 ↪set_index('Model')

log_and_print("\nModel Comparison (log scale):")
display(style_df(comparison_log_df[['r2', 'mae', 'rmse']]))
```

```
Model Comparison (log scale):
```

```
<pandas.io.formats.style.Styler at 0x2cc2521c2d0>
```

5. Final Visualizations and Model Saving

We'll now generate a parity plot and a SHAP summary plot for our final, tuned model to visually inspect its performance and interpret its predictions. Finally, we save the model for deployment.

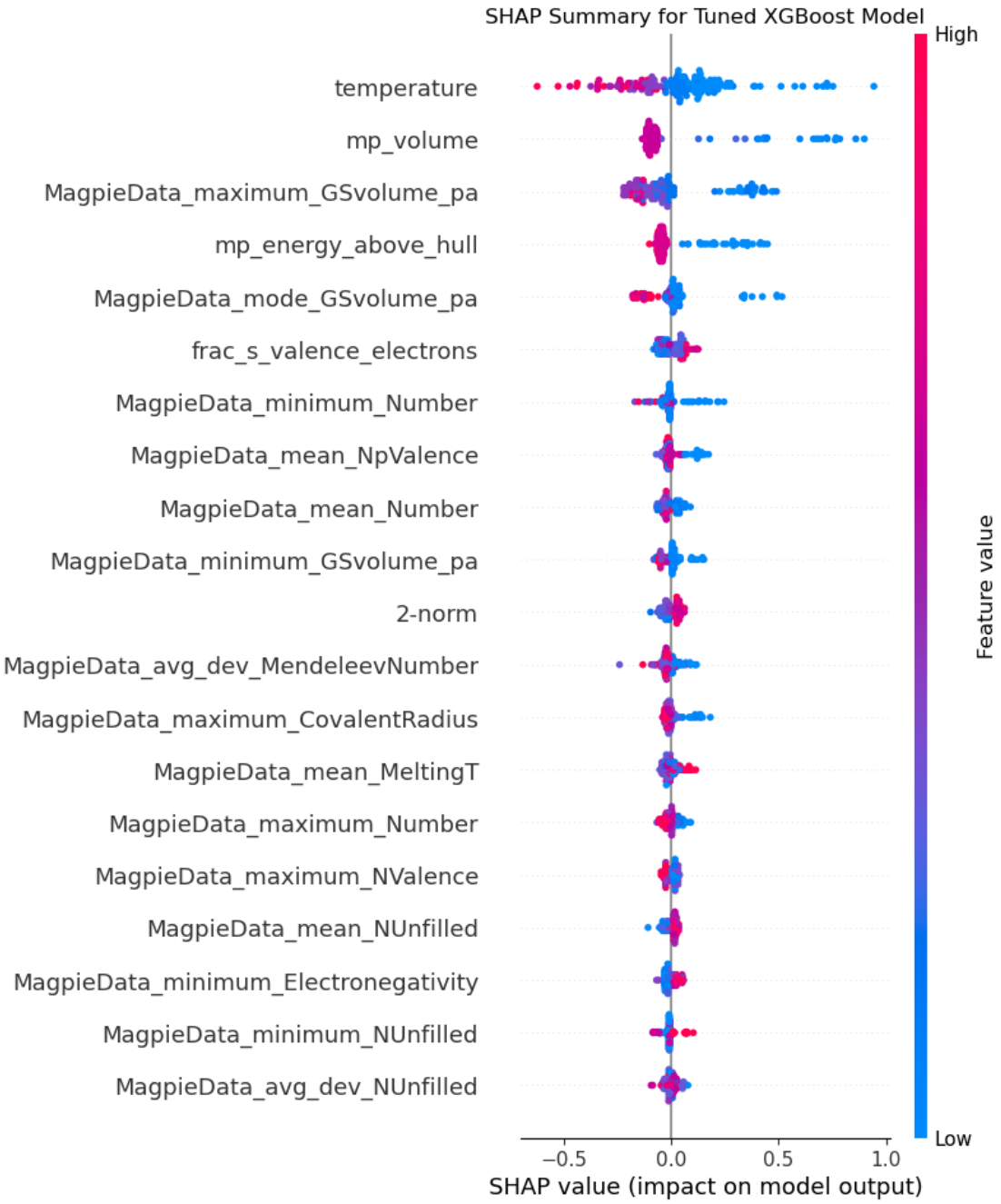
[ ]:
```
# --- PARITY PLOT FOR TUNED MODEL ---
log_and_print("\nGenerating final plots for the winning model (Tuned XGBoost)...
 ↪")
fig_parity = plot_parity_logscale(best_xgb_tuned, X_test_scaled, y_test_log,␣
 ↪"Tuned XGBoost")
save_plot(fig_parity, PARITY_PLOT_PATH)
```

```
fig_parity.show()

# Generate and save the SHAP summary plot
explainer = shap.TreeExplainer(best_xgb_tuned)
shap_values = explainer.shap_values(X_test_scaled)
shap.summary_plot(shap_values, X_test_scaled, show=False,␣
 ↪feature_names=X_selected.columns)
plt.title("SHAP Summary for Tuned XGBoost Model")
save_plot(plt.gcf(), SHAP_PLOT_PATH)
plt.show()
log_and_print(f"Parity plot saved to {PARITY_PLOT_PATH}")
log_and_print(f"SHAP summary plot saved to {SHAP_PLOT_PATH}")
```

```
Generating final plots for the winning model (Tuned XGBoost)…
Plot saved to c:\Users\angel\Thermal-Conductivity-
ML\plots\5_hyperparameter_tuning\tuned_xgb_model_parity_plot.pdf

Plot saved to c:\Users\angel\Thermal-Conductivity-
ML\plots\5_hyperparameter_tuning\tuned_xgb_model_shap_summary.pdf
```




```
Parity plot saved to c:\Users\angel\Thermal-Conductivity-
ML\plots\5_hyperparameter_tuning\tuned_xgb_model_parity_plot.pdf
SHAP summary plot saved to c:\Users\angel\Thermal-Conductivity-
ML\plots\5_hyperparameter_tuning\tuned_xgb_model_shap_summary.pdf
```

[ ]:
```
# Save the scaler object for use in the prediction script
SCALER_PATH = os.path.join(PROJECT_ROOT, 'models', 'scaler.joblib')
```

```
joblib.dump(scaler, SCALER_PATH)
log_and_print(f"Scaler saved to: {SCALER_PATH}")

# Save the final tuned model
joblib.dump(best_xgb_tuned, FINAL_MODEL_PATH)
log_and_print(f"\nWinning model (Tuned XGBoost) saved to: {FINAL_MODEL_PATH}")
```

```
Scaler saved to: c:\Users\angel\Thermal-Conductivity-ML\models\scaler.joblib

Winning model (Tuned XGBoost) saved to: c:\Users\angel\Thermal-Conductivity-
ML\models\tuned_xgboost_model.joblib
```

6. Conclusion and Next Steps

The hyperparameter tuning process successfully improved the XGBoost model's performance, leading to a higher R^2 score and lower error metrics compared to the baseline model on the curated feature set. The final model, saved as `tuned_xgboost_model.joblib`, represents the culmination of our workflow and is now ready for use in our prediction pipeline.

The SHAP analysis confirms that the model's predictions are driven by physically meaningful properties, such as atomic volume, temperature, and electronic structure, which increases our confidence in its real-world applicability.

This project demonstrates a complete, end-to-end machine learning workflow, from initial data exploration and cleaning to rigorous, data-driven model selection and final optimization. Key decisions, such as rejecting power-transformed features and cluster labels, were made based on evidence, resulting in a final model that is both accurate and appropriately complex.

Potential Next Steps:

1. **Deployment:** Integrate the saved `tuned_xgboost_model.joblib` into a web service or API for real-time predictions, using the `scripts/predict_from_csv.py` script as a template.
2. **Deeper Error Analysis:** Investigate the largest prediction errors to identify specific material classes or regions of the feature space where the model struggles, which could guide future feature engineering efforts.
3. **Continuous Improvement:** As more data becomes available, retrain and re-tune the model to further enhance its predictive accuracy and expand its applicability.
4. **Alternative Architectures:** While XGBoost performed well, exploring other advanced models like Graph Neural Networks (GNNs), which are well-suited for materials science, could yield further improvements.