3_model_comparison

August 7, 2025

Restarted cpu (Python 3.13.1)

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

Predicting Thermal Conductivity of Inorganic Materials via Machine Learning

Project Objective: To develop a robust machine learning pipeline for predicting the thermal conductivity of inorganic materials, leveraging composition-based features. This project demonstrates an end-to-end workflow, from data integration and feature engineering to model comparison and interpretation.

Key Stages of the Workflow: 1. **Data Integration:** Consolidate and clean thermal conductivity data from benchmark datasets. 2. **Feature Engineering:** Generate a rich set of 145 physics-informed features (e.g., elemental properties, stoichiometry, structural characteristics) from material compositions using `matminer`. 3. **Model Comparison:** Systematically train and evaluate a suite of regression models, including XGBoost, Random Forest, Gradient Boosting, and Support Vector Regression (SVR), on both standardized and power-transformed feature sets. 4. **Performance Evaluation & Selection:** Assess model performance using R^2, MAE, and RMSE metrics to identify the optimal model and preprocessing strategy. 5. **In-Depth Analysis:** Utilize SHAP (SHapley Additive exPlanations) for model interpretation, uncovering the key drivers of thermal conductivity.

This notebook serves as the core of the analysis, focusing on model evaluation and selection.

Accelerating Materials Discovery with Machine Learning

This notebook demonstrates a data-driven approach to predict the thermal conductivity of inorganic compounds. By integrating domain knowledge from materials science with robust machine learning techniques, we build interpretable models that can accelerate the discovery of novel materials.

The Challenge: Efficient Thermal Management

Efficient thermal management is critical across numerous high-tech sectors, from electronics to renewable energy. For instance, cooling can account for up to 40% of a data center's energy consumption. Materials with high thermal conductivity are essential for dissipating heat, improving device reliability, and enabling higher power densities. However, the experimental discovery and characterization of such materials is often slow and resource-intensive.

> **Impact Example:** A mere 10% improvement in cooling efficiency in a 1 MW data center can translate to annual savings of $30,000–$50,000, highlighting the significant economic incentive for developing superior thermal materials.

Our Approach: A Materials Informatics Workflow

We employ a machine learning workflow to predict material properties from compositional data alone, bypassing the need for extensive experimentation. This approach offers several key advantages:

- **Efficiency:** Rapidly screens thousands of candidate materials, drastically reducing the time and cost associated with laboratory testing.
- **Scalability:** Once trained, the model can make predictions for new compositions almost instantaneously.
- **Insight:** By analyzing model internals (e.g., with SHAP), we can uncover the underlying relationships between a material's composition and its thermal properties, guiding future research.
- **Generalizability:** This workflow is highly adaptable and can be repurposed to predict other key material properties, such as elastic moduli, Seebeck coefficient, or electronic bandgap.

Key Scientific Precedent: Our work builds upon established research in the field of materials informatics, which has demonstrated the power of machine learning in materials science. * *A general-purpose machine learning framework for predicting materials properties.* Ward et al., 2016. doi:10.1038/npjcompumats.2016.28 * *"Machine learning for molecular and materials science."* Butler et al., 2018. doi:10.1038/s41586-018-0337-2

Data Loading, Cleaning, and Feature Gathering

[ ]:
```
# --- Professionalized Imports and Setup ---
import os, sys
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,
    save_plot, style_df,
    prepare_data_for_modeling,
    log_and_print,
    load_or_process_dataframe

)
from modeling import (
    split_data,
```

```
    scale_features,
    train_xgboost,
    train_random_forest,
    train_gradient_boosting,
    train_svr,
    evaluate_model
)
from viz import (
    plot_model_comparison,
    plot_feature_importance,
    plot_parity_logscale,
    plot_residuals,
    plot_parity_grid,
    add_subplot_border
)
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from IPython.display import display
import json
from sklearn.preprocessing import PowerTransformer

# --- Setup environment and paths ---
setup_environment()
USE_CLUSTER_FEATURES = False
PLOTS_DIR = os.path.join(PROJECT_ROOT, 'plots', '3_model_comparison')
os.makedirs(PLOTS_DIR, exist_ok=True)
PARITY_GRID_PATH = os.path.join(PLOTS_DIR, 'adv_model_parity_grid.pdf')
MODEL_COMP_PATH = os.path.join(PLOTS_DIR, 'adv_model_comparison.pdf')
XGB_SHAP_PATH = os.path.join(PLOTS_DIR, 'adv_model_xgb_shap.pdf')

# --- Create plots directory ---
os.makedirs(PLOTS_DIR, exist_ok=True)

# --- Load featurized data using robust utility ---
CACHE_PATH = os.path.join(PROJECT_ROOT, 'data', 'processed', 'featurized.
 ↪parquet')
df = load_or_process_dataframe(cache_path=CACHE_PATH, project_root=PROJECT_ROOT)
log_and_print(f"Featurized dataframe shape: {df.shape}")
style_df(df.head())
```

```
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)
```

[ ]: <pandas.io.formats.style.Styler at 0x2cac76a9be0>

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

# Conditionally add cluster features based on the flag
if USE_CLUSTER_FEATURES:
    log_and_print("Generating cluster features...")
    from sklearn.preprocessing import StandardScaler
    from sklearn.cluster import KMeans
    X_numeric = X.select_dtypes(include=np.number)
    scaler_for_clustering = StandardScaler()
    X_scaled_for_clustering = scaler_for_clustering.fit_transform(X_numeric)
    kmeans = KMeans(n_clusters=9, random_state=42, n_init=10)
    cluster_labels = kmeans.fit_predict(X_scaled_for_clustering)
    X['cluster_label'] = cluster_labels
    X_final = pd.get_dummies(X, columns=['cluster_label'], prefix='cluster')
    log_and_print("Successfully added cluster labels as a one-hot encoded␣
 ↪feature.")
    display(style_df(X_final.head()))
else:
    X_final = X
    log_and_print("Skipping cluster feature generation. Using original feature␣
 ↪set.")
    display(style_df(X_final.head()))
```

```
Imputing NaN values in columns: ['mp_density', 'mp_volume', 'mp_band_gap',
'mp_energy_above_hull', 'jarvis_band_gap', 'jarvis_formation_energy', 'density']
Skipping cluster feature generation. Using original feature set.
```

<pandas.io.formats.style.Styler at 0x2cac7780050>

[ ]:
```
print("Data loaded and featurized successfully.")
X_final.head()
```

```
Data loaded and featurized successfully.
```

[ ]:

	temperature	MagpieData_minimum_Number	MagpieData_maximum_Number	\
0	300.0	32.0	52.0	
1	400.0	32.0	52.0	
2	700.0	32.0	52.0	
3	1000.0	32.0	52.0	
4	300.0	47.0	81.0	

	MagpieData_range_Number	MagpieData_mean_Number	MagpieData_avg_dev_Number	\
0	20.0	44.217391	9.030246	
1	20.0	44.217391	9.030246	
2	20.0	44.217391	9.030246	
3	20.0	44.217391	9.030246	

```
4                    34.0                50.933333                    4.720000

   MagpieData_mode_Number  MagpieData_minimum_MendeleevNumber  \
0                    52.0                                65.0
1                    52.0                                65.0
2                    52.0                                65.0
3                    52.0                                65.0
4                    47.0                                65.0

   MagpieData_maximum_MendeleevNumber  MagpieData_range_MendeleevNumber  …  \
0                                90.0                              25.0  …
1                                90.0                              25.0  …
2                                90.0                              25.0  …
3                                90.0                              25.0  …
4                                90.0                              25.0  …

   frac_p_valence_electrons  frac_d_valence_electrons  \
0                  0.197368                  0.672515
1                  0.197368                  0.672515
2                  0.197368                  0.672515
3                  0.197368                  0.672515
4                  0.101942                  0.728155

   frac_f_valence_electrons  mp_density   mp_volume  mp_band_gap  \
0                  0.000000    5.111686  222.541507     1.925878
1                  0.000000    5.111686  222.541507     1.925878
2                  0.000000    5.111686  222.541507     1.925878
3                  0.000000    5.111686  222.541507     1.925878
4                  0.067961    5.111686  222.541507     1.925878

   mp_energy_above_hull  jarvis_band_gap  jarvis_formation_energy   density
0              0.005818         1.554392                -1.060942  5.111686
1              0.005818         1.554392                -1.060942  5.111686
2              0.005818         1.554392                -1.060942  5.111686
3              0.005818         1.554392                -1.060942  5.111686
4              0.005818         1.554392                -1.060942  5.111686

[5 rows x 154 columns]
```

[ ]:
```
# ------------------------------------------------------------------------
# Assemble the combined dataset
# ------------------------------------------------------------------------
```

[ ]:
```
# ## Data Preparation and Preprocessing
#
# With the featurized dataset loaded, we proceed with preparing the data for␣
  ↪modeling. This involves:
```

```
# 1. **Separating Features and Target:** Isolating the feature matrix (`X`) from the target variable (`y`), which is `thermal_conductivity`.
# 2. **Log Transformation of Target:** Applying a natural logarithm transformation to the target variable (`y`). This is a standard practice when dealing with right-skewed data, as it helps to stabilize variance and linearize relationships, often leading to improved model performance.
# 3. **Data Splitting:** Partitioning the data into training and testing sets to ensure an unbiased evaluation of model performance.
# 4. **Feature Scaling:** Applying `StandardScaler` to the feature sets. This scales each feature to have a mean of 0 and a standard deviation of 1, which is crucial for distance-based algorithms (like SVR) and helps gradient-based methods (like XGBoost) converge faster.
# 5. **Feature Transformation (Experimental):** Applying a `PowerTransformer` (Yeo-Johnson method) to a copy of the scaled data. This transformation attempts to make the feature distributions more Gaussian. We will train models on both the scaled and the power-transformed datasets to empirically determine which representation yields better results.
```

[ ]:
```
# Remove feature selection logic and use the full feature set
X_train, X_test, y_train_log, y_test_log, y_train, y_test = split_data(X_final, y)
X_train_scaled, X_test_scaled, scaler = scale_features(X_train, X_test)
log_and_print("Data split and scaled using the full feature set.")
log_and_print(f"X_train_scaled shape: {X_train_scaled.shape}")
log_and_print(f"X_test_scaled shape: {X_test_scaled.shape}")

# --- Optional: Apply PowerTransformer ---
pt = PowerTransformer(method='yeo-johnson')
X_train_transformed = pt.fit_transform(X_train_scaled)
X_test_transformed = pt.transform(X_test_scaled)
log_and_print("\nApplied PowerTransformer to the scaled feature sets.")

# Convert transformed arrays back to DataFrames
X_train_transformed = pd.DataFrame(X_train_transformed, columns=X_train.columns, index=X_train.index)
X_test_transformed = pd.DataFrame(X_test_transformed, columns=X_test.columns, index=X_test.index)
```

```
Data split and scaled using the full feature set.
X_train_scaled shape: (605, 154)
X_test_scaled shape: (152, 154)

Applied PowerTransformer to the scaled feature sets.
```

[ ]:
```
## Model Training and Evaluation
#
```

```
# We now systematically train and evaluate our suite of regression models. To␣
 ↪identify the most effective preprocessing strategy, each model is trained on␣
 ↪two different versions of the feature set:
# 1.  **Standard Scaled Features:** Features scaled to have zero mean and unit␣
 ↪variance.
# 2.  **Power Transformed Features:** Scaled features that have also been␣
 ↪passed through a Yeo-Johnson power transformation to approximate a Gaussian␣
 ↪distribution.
#
# This comparative approach allows us to rigorously assess the impact of␣
 ↪feature distribution on model performance and select the combination of␣
 ↪model and preprocessing that yields the most accurate predictions. The␣
 ↪following models are evaluated:
# - **XGBoost:** A highly efficient and powerful gradient boosting␣
 ↪implementation.
# - **Random Forest:** An ensemble of decision trees, known for its robustness.
# - **Gradient Boosting:** Another powerful ensemble method based on boosting.
# - **SVR (Support Vector Regressor):** A kernel-based method effective in␣
 ↪high-dimensional spaces.
#
# All models are trained to predict the log-transformed thermal conductivity.

# Define model trainers and their names
model_trainers = {
    "XGBoost": train_xgboost,
    "Random Forest": train_random_forest,
    "Gradient Boosting": train_gradient_boosting,
    "SVR": train_svr,
}

# Define datasets
datasets = {
    "": (X_train_scaled, X_test_scaled),
    "_transformed": (X_train_transformed, X_test_transformed)
}

all_models = {}
all_results = {}

# Loop through models and datasets to train and evaluate
for model_name, trainer_func in model_trainers.items():
    for suffix, (X_train_data, X_test_data) in datasets.items():
        full_model_name = f"{model_name}{suffix}"
        log_and_print(f"--- Training {full_model_name} ---")

        model = trainer_func(X_train_data, y_train_log)
```

```
        all_models[full_model_name] = model

        results = evaluate_model(model, X_test_data, y_test_log, y_test)
        all_results[full_model_name] = results

        log_and_print(f"{full_model_name} training complete.")

print("\n--- All models trained and evaluated ---")
```

```
--- Training XGBoost ---
XGBoost training complete.
--- Training XGBoost_transformed ---
XGBoost_transformed training complete.
--- Training Random Forest ---
Random Forest training complete.
--- Training Random Forest_transformed ---
Random Forest_transformed training complete.
--- Training Gradient Boosting ---
Gradient Boosting training complete.
--- Training Gradient Boosting_transformed ---
Gradient Boosting_transformed training complete.
--- Training SVR ---
SVR training complete.
--- Training SVR_transformed ---
SVR_transformed training complete.

--- All models trained and evaluated ---
```

[ ]:
```
# Display results for all models on the log-transformed scale
for model_name, result in all_results.items():
    print(f"\n--- {model_name} Evaluation (Log-Transformed Scale) ---")
    print(f"MAE: {result['log']['mae']:.4f}, RMSE: {result['log']['rmse']:.4f},␣
 ↪R²: {result['log']['r2']:.4f}")
```

```
--- XGBoost Evaluation (Log-Transformed Scale) ---
MAE: 0.1856, RMSE: 0.3992, R²: 0.8612

--- XGBoost_transformed Evaluation (Log-Transformed Scale) ---
MAE: 0.1856, RMSE: 0.3995, R²: 0.8610

--- Random Forest Evaluation (Log-Transformed Scale) ---
MAE: 0.2111, RMSE: 0.4497, R²: 0.8239

--- Random Forest_transformed Evaluation (Log-Transformed Scale) ---
MAE: 0.2133, RMSE: 0.4515, R²: 0.8224

--- Gradient Boosting Evaluation (Log-Transformed Scale) ---
```

```
MAE: 0.2305, RMSE: 0.4290, R²: 0.8397

--- Gradient Boosting_transformed Evaluation (Log-Transformed Scale) ---
MAE: 0.2316, RMSE: 0.4309, R²: 0.8383

--- SVR Evaluation (Log-Transformed Scale) ---
MAE: 0.2988, RMSE: 0.6228, R²: 0.6622

--- SVR_transformed Evaluation (Log-Transformed Scale) ---
MAE: 0.2895, RMSE: 0.5828, R²: 0.7042
```

[ ]:
```
# ### Model Performance Metrics
```

[ ]:
```
# Create a DataFrame to display the results
results_summary = {model_name: result_data['log'] for model_name, result_data␣
 ↪in all_results.items()}
results_df = pd.DataFrame(results_summary).T

# Display the table
log_and_print("--- Model Performance Comparison (Log-Transformed Scale) ---")
display(style_df(results_df[['r2', 'mae', 'rmse']]))

# Save the comparison table as a CSV file
results_df.to_csv(os.path.join(PLOTS_DIR, 'model_comparison.csv'), index=True)
log_and_print("Comparison table saved as model_comparison.csv")
```

```
--- Model Performance Comparison (Log-Transformed Scale) ---

<pandas.io.formats.style.Styler at 0x2cac7924410>

Comparison table saved as model_comparison.csv
```

Model Selection: XGBoost on Scaled Features

The Decision Based on a comprehensive comparison of performance metrics, the **XGBoost model trained on standard scaled features (without power transformation)** is selected as the optimal model for this project.

Rationale for Selection

1. **Superior Predictive Accuracy:** The `XGBoost` model achieved the lowest Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) among all candidates, indicating its predictions are, on average, the most accurate.
2. **Robustness of Tree-Based Models:** A key insight from this experiment is that the tree-based models (XGBoost, Random Forest, Gradient Boosting) showed **no significant performance gain** from the `PowerTransformer`. Their non-linear, tree-based structure makes them inherently robust to the underlying distribution of features.
3. **Benefit for SVR:** In contrast, the `SVR` model's performance **did improve** with the transformed features. This is expected, as distance-based algorithms like SVR are sensitive to

feature scale and distribution. However, even the improved SVR was not competitive with the top tree-based models.

4. **The Value of Simplicity:** Since the best-performing model (`XGBoost`) does not benefit from the extra `PowerTransformer` step, we choose the simpler preprocessing pipeline. This reduces model complexity, improves training and inference speed, and enhances interpretability.

Conclusion This rigorous comparison validates our choice of the non-transformed **XGBoost** model. It provides the best balance of high accuracy, simplicity, and robustness. This model will be carried forward for more detailed feature selection and hyperparameter tuning in the subsequent notebooks.

Next Notebook: `4_modeling_and_feature_selection.py`