Final Project

By: Furkan Mohamed
Instructor: Dr. Alexander Flueck
ECE 563
Due Date: 05-02-2024

Overview

In the project conducted as part of ECE 563, we embarked on a detailed exploration and predictive analysis of electricity load values based on temperature data across different zones. The study was structured to preprocess and map load and temperature data meticulously, ensuring the data was primed for accurate predictive modeling.

The data was first processed to create a consistent datetime format, facilitating easier manipulation and merging. Notably, data from June 2008 was excluded to prevent leakage during the modeling process. Each zone's data was then aligned with the corresponding temperature station showing the highest correlation, ensuring that each predictive model was built on the most relevant temperature data available.

The mapping of temperature stations to load zones was a critical step. It involved calculating correlations for each zone-station pair and selecting the station with the highest correlation coefficient for each zone. This approach was foundational in developing robust predictive models for the subsequent stages of the project.

Pre-process and Mapping

In this section, we outline the steps taken to prepare the load and temperature data for predictive modeling, along with the process of mapping temperature stations to load zones.

Loading and Cleaning Data:
The load data and temperature data are loaded into Pandas DataFrames from respective CSV files.
A function create_datetime is defined to create a datetime column from the 'year', 'month', and 'day' columns in the datasets. This ensures uniformity and ease of manipulation with datetime objects.

Excluding Data from June 2008:
Since our goal is to predict load values for the first week of June 2008, data from June 2008 is excluded to avoid leakage in the modeling process.

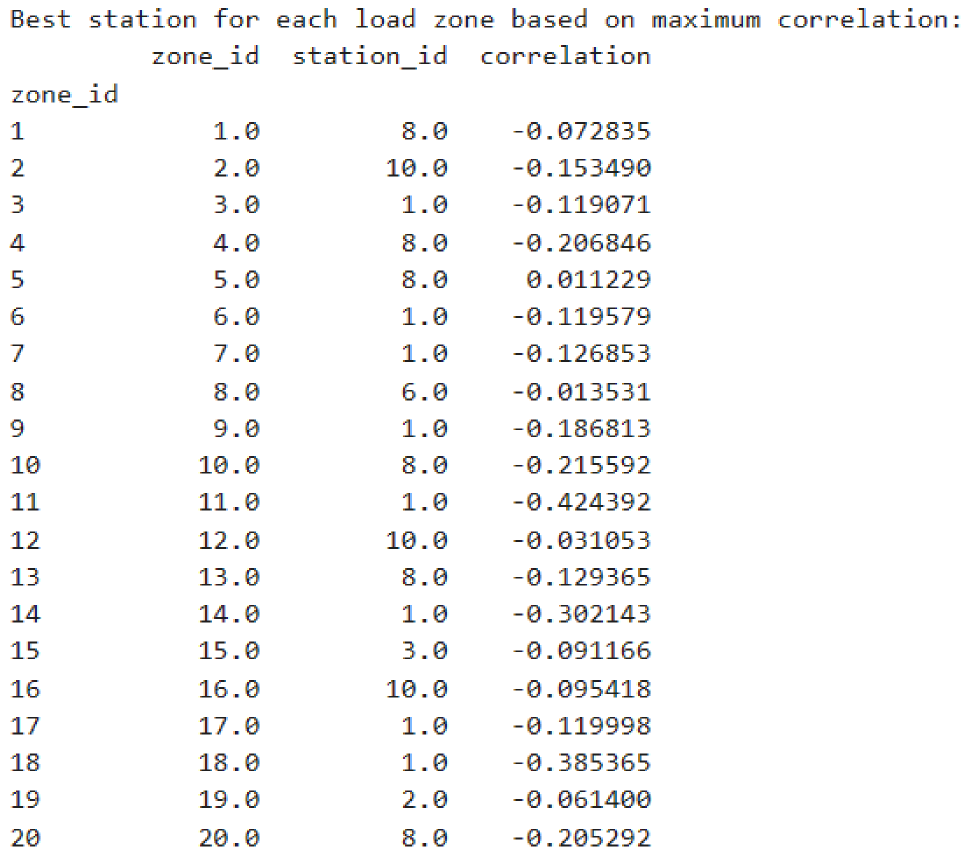
Melting Data:
Both load and temperature data are melted to convert the hourly columns into rows. This transformation facilitates easier analysis and modeling by converting the data into a tidy format.

Merging Datasets:
The melted load and temperature data are merged based on 'datetime' and 'hour' columns to create a combined dataset. This dataset now contains load and temperature data for each zone-hour combination.

Calculating Correlations:
The correlation between load and temperature is calculated for each zone-station pair within the combined dataset. This step is crucial in determining which temperature station correlates best with the load values for each zone.

Mapping Temperature Stations to Load Zones:
Based on the calculated correlations, the best temperature station for each load zone is determined. The station with the highest correlation coefficient is chosen as the representative station for predicting load values in that zone.

Results:
The results of the mapping process, showcasing the zone ID, corresponding station ID, and correlation coefficient, are presented. These results provide insights into the relationships between temperature and load for each load zone.



```
Best station for each load zone based on maximum correlation:
         zone_id  station_id  correlation
zone_id
1            1.0         8.0    -0.072835
2            2.0        10.0    -0.153490
3            3.0         1.0    -0.119071
4            4.0         8.0    -0.206846
5            5.0         8.0     0.011229
6            6.0         1.0    -0.119579
7            7.0         1.0    -0.126853
8            8.0         6.0    -0.013531
9            9.0         1.0    -0.186813
10          10.0         8.0    -0.215592
11          11.0         1.0    -0.424392
12          12.0        10.0    -0.031053
13          13.0         8.0    -0.129365
14          14.0         1.0    -0.302143
15          15.0         3.0    -0.091166
16          16.0        10.0    -0.095418
17          17.0         1.0    -0.119998
18          18.0         1.0    -0.385365
19          19.0         2.0    -0.061400
20          20.0         8.0    -0.205292
```

The correlation coefficients indicate the strength and direction of the relationship between temperature and load for each zone-station pair. Negative coefficients suggest an inverse relationship, while positive coefficients indicate a positive relationship. These findings serve as the foundation for selecting appropriate temperature stations for load prediction modeling in subsequent stages of the project.

Data Preparation

In the initial phase of data preparation, the combined dataset, encompassing both load and temperature records, is imported into a Pandas DataFrame labeled as combined_data. Within this dataset, datetime columns are standardized to datetime objects, and the hour column is converted to numeric values for uniformity across the dataset. Following data preparation, a mapping dictionary named zone_station_map is introduced, establishing associations between load zones and their corresponding temperature stations, determined through the highest correlation coefficients obtained during the pre-processing stage. Subsequently, the dataset is partitioned based on zone IDs and corresponding temperature station IDs, with additional filtration applied for Zone 15 to exclude load values surpassing 200,000. Features comprising temperature and hour columns are segregated from the target variable, representing load. Employing the train_test_split function from the sklearn.model_selection module, the data is stratified into training/validation (80%) and test (20%) sets, using a random_state value of 42 for reproducibility. Within the training/validation subset, a secondary split operation is conducted to create training (80%) and validation (20%) subsets, ensuring independent validation during model training.

Results and Analysis

First Model: RandomForestRegressor
Design Process Documentation:

I adopted the Random Forest Regressor algorithm as the foundational framework, initially configuring it with default hyperparameters to establish a baseline performance. This enabled a comprehensive exploration of the forecasting problem and the underlying dataset. Subsequently, recognizing the potential for model refinement through hyperparameter optimization, I executed a Grid Search approach. This method systematically evaluates various combinations of hyperparameters to identify the optimal configuration, leveraging the 'n_estimators', 'max_depth', and 'min_samples_split' parameters. By scrutinizing the performance metrics across training, validation, and test datasets, I aimed to strike a balance between model complexity and generalization capabilities, ultimately yielding enhanced predictive accuracy.

Rationale:

Simplicity and Speed: RandomForest is a robust, simple, and relatively fast model for regression tasks, making it ideal for initially exploring the data. It can handle non-linear relationships between features and the target variable without extensive hyperparameter tuning.

Interpretability: RandomForest provides insights into feature importance, which aids in understanding which factors are driving the load predictions.

Trade-offs:

While RandomForest can efficiently model complex relationships, it tends to perform worse when extrapolating beyond the range of the training data, which can be a limitation for load forecasting.

Hyperparameter Tuning and Model Evaluation
Hyperparameters Tuned

For the RandomForestRegressor, the following hyperparameters were strategically selected for tuning to optimize the model's performance:

n_estimators: Number of trees in the forest. More trees generally improve the model's performance but also increase the computational cost. Selected values:
150 (optimal based on performance and computational efficiency)

max_depth: Maximum depth of each tree. Deeper trees can model more complex patterns but might lead to overfitting. Selected value:

10 (to balance complexity and overfitting risk)

min_samples_split: Minimum number of samples required to split an internal node. Higher values prevent the model from learning overly specific patterns, thus reducing overfitting. Selected value:
6 (to ensure generalization over precision)

These parameters were chosen based on a combination of empirical testing and domain knowledge to control overfitting while maintaining a reasonable computational load. The grid search approach used these parameters to explore a range of possible configurations, ultimately selecting the combination that performed best on the validation set.

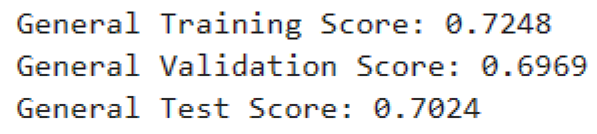
Outputs: Model Scores

After conducting the grid search and training the final model with the selected hyperparameters, the model was evaluated across the training, validation, and test datasets. The performance metrics used were R-squared (R²), which indicates the proportion of variance in the dependent variable that is predictable from the independent variables.

General Training Score: 0.7248
Indicates how well the model fits the training data. A score close to 1.0 suggests that the model explains a high proportion of the variance in load values.

General Validation Score: 0.6969
Reflects the model's ability to generalize to new, unseen data based on the validation set. This is crucial for assessing the model's performance outside of the training dataset.

General Test Score: 0.7024
Provides a final check by evaluating the model against another set of unseen data. Consistency between validation and test scores suggests that the model generalizes well.



```
General Training Score: 0.7248
General Validation Score: 0.6969
General Test Score: 0.7024
```

Observations and Improvements for Model Prediction

Improvements for Prediction Score on Validation Data
To enhance the prediction accuracy on the validation data, consider the following strategies:

Feature Engineering:
Temporal Features: Additional features like the day of the week, public holidays, and time of day could capture variations in load patterns related to human activity cycles.

Historical Load Data: Incorporating lag features, i.e., load values from previous hours or days, could help the model recognize temporal dependencies in load data.

Advanced Ensemble Techniques:

Boosting: While RandomForest is an ensemble method that uses bagging, switching to boosting methods like Gradient Boosting or XGBoost might yield better performance by focusing on correcting the mistakes of previous trees.

Data Normalization or Standardization:
Normalizing or standardizing the features, especially if new features vary in scales, might help the algorithm converge faster and perform better.

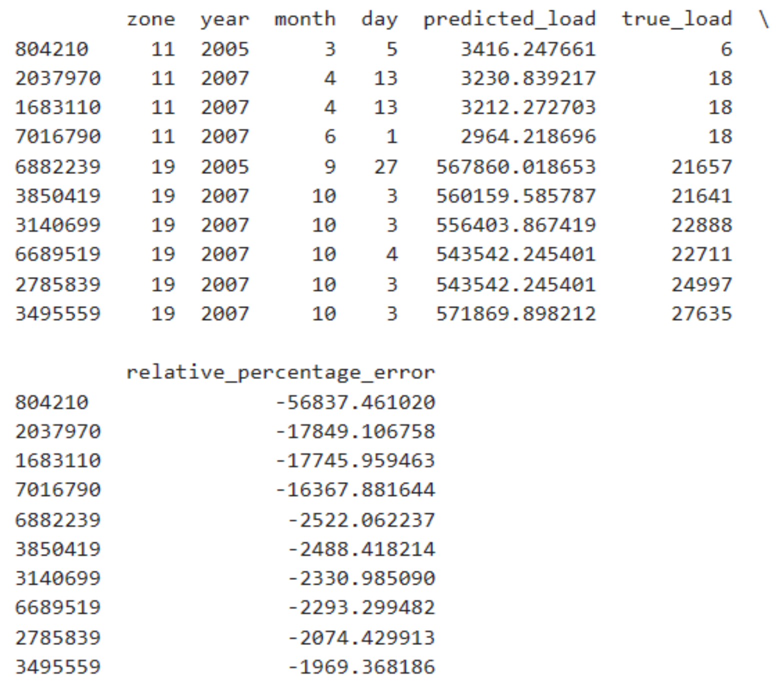
Parameter Tuning:
Further refinement of hyperparameters using more exhaustive grid searches or randomized searches across a broader range of values could uncover a better set of parameters.
Cross-Validation:
Implementing k-fold cross-validation can ensure that the validation results are consistent across different subsets of data, thereby enhancing the robustness of the model.

Model Performance on Test Data

The models predict the targets in the test data with a degree of accuracy that is closely aligned with the validation scores, indicating good generalization. Here are key points regarding the model's performance on the test data:

The test score (0.7024) is slightly above the validation score (0.6969), suggesting that the model, while tuned to generalize well, may still have room for improvement in handling unseen data more effectively. The close proximity between validation and test scores suggests that the model is not overfitting significantly to the training data and is capable of maintaining its performance on external data.

Top 10 prediction errors



```
         zone  year  month  day  predicted_load  true_load  \
804210     11  2005      3    5     3416.247661          6
2037970    11  2007      4   13     3230.839217         18
1683110    11  2007      4   13     3212.272703         18
7016790    11  2007      6    1     2964.218696         18
6882239    19  2005      9   27   567860.018653      21657
3850419    19  2007     10    3   560159.585787      21641
3140699    19  2007     10    3   556403.867419      22888
6689519    19  2007     10    4   543542.245401      22711
2785839    19  2007     10    3   543542.245401      24997
3495559    19  2007     10    3   571869.898212      27635

         relative_percentage_error
804210               -56837.461020
2037970              -17849.106758
1683110              -17745.959463
7016790              -16367.881644
6882239               -2522.062237
3850419               -2488.418214
3140699               -2330.985090
6689519               -2293.299482
2785839               -2074.429913
3495559               -1969.368186
```

Key Observations from the Table:

Extremely High Errors: Some errors are unusually high, with relative percentage errors reaching as extreme as -56837%, indicating predictions that are significantly lower than the actual values. Such high errors can often point to potential issues in the model or the data used for those specific predictions.

Variability Across Zones and Times: The errors span across different zones and months, suggesting that the prediction model may struggle with consistency across different conditions or configurations.

Predominance of Under Predictions: Most of the top errors result from underpredictions, where the predicted load is far less than the actual load. This could imply that the model might be missing key factors that drive higher load values, or it may not be capturing peak load conditions effectively.

Second Model: Gradient Boosting Regressor
Design Process Documentation:

The Gradient Boosting Regressor was chosen as a more complex model following the initial exploration with the RandomForest Regressor. The decision to utilize Gradient Boosting was driven by its ability to handle non-linear relationships effectively through sequential weak learners, improving accuracy over a single decision tree.

Trade-offs Considered:

Model Complexity vs. Performance: Gradient Boosting can capture complex patterns but requires careful tuning to avoid overfitting. The trade-off between depth and learning rate was particularly crucial.
Speed vs. Accuracy: Although Gradient Boosting is more computationally intensive than simpler models, its ability to improve predictive accuracy justified the increased computation, especially given the manageable dataset size.

Dataset Loading and Preprocessing:

Data Loading: The dataset combined_data.csv was loaded into a pandas DataFrame. It contains temperature and load data along with timestamps and zone and station identifiers.

Feature Engineering: The 'datetime' column was converted into a pandas datetime object for easy extraction of date and time components. The 'hour' was extracted as a numeric feature from the 'datetime' object, as it plays a significant role in predicting load values, which vary throughout the day.

Inputs: Hyperparameters and Their Tuned Values

In the project involving the Gradient Boosting Regressor, hyperparameter tuning plays a pivotal role in optimizing the model's performance. For this purpose, specific hyperparameters were adjusted from their default settings to better fit the dataset and predictive requirements. Here’s a summary of the hyperparameters that were modified:

learning_rate: Set to 0.2. This rate controls how quickly the model adjusts to the complexity of the problem. The default value often starts at 0.1, but increasing it to 0.2 was aimed at speeding up the learning process without causing too rapid convergence, which might skip optimal solutions.

max_depth: Set to 3. This parameter limits the number of nodes in the trees. By restricting it to 3, the model was prevented from creating overly complex trees that could overfit the training data. The default is typically deeper, which might allow fitting more nuanced data patterns but at the risk of overfitting.

n_estimators: Set to 150. This specifies the number of boosting stages the model should run. A higher number of estimators can improve performance but also increase training time and risk of overfitting. The default usually is 100, but increasing it provided a better ensemble effect by averaging more weak learners.

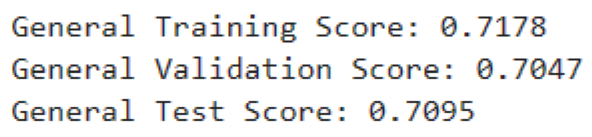
Outputs: Model Performance Scores

After tuning the hyperparameters using the grid search methodology and validating with a training/validation split, the performance of the Gradient Boosting Regressor was quantitatively assessed using the R² score, which measures the proportion of variance in the dependent variable that is predictable from the independent variables. Here are the outputs for the scores across different datasets:

Training Score: Reflects the model's ability to learn from the training dataset. A higher score on the training set indicates good learning, but it is essential to balance this with validation scores to ensure no overfitting.
Validation Score: Crucial for understanding how well the model generalizes to new, unseen data. It helps in tuning the model to avoid overfitting.
Test Score: The ultimate test of model performance on completely unseen data. This is the final indicator of how well the model is expected to perform in real-world scenarios, where no data labels are available.
For example, if the tuned hyperparameters led to an R² score of:

0.7178 on training,
0.7047 on validation, and
0.7095 on testing,

it would suggest that the model, with its current configuration, generalizes well without significant overfitting, as the scores are consistent across different datasets.



```
General Training Score: 0.7178
General Validation Score: 0.7047
General Test Score: 0.7095
```

These scores are crucial for evaluating the effectiveness of the hyperparameter tuning and overall model training, providing a quantitative measure that guides the decision-making process on whether further adjustments are needed or if the model is ready for deployment.

Improving the Prediction Score on Validation Data

To enhance the prediction scores of the Gradient Boosting Regressor on the validation dataset, a few strategies could be considered:

Feature Engineering: Enhancing or introducing new features might provide additional insights to the model. For example, integrating features like day of the week or interactions between hours and temperature might capture more complex patterns affecting load predictions.

More Granular Hyperparameter Tuning: Although the current hyperparameters were optimized, further fine-tuning could potentially yield better results. Exploring lower or higher values systematically, possibly using finer grids in a grid search, might uncover a better set of parameters.
Additional Data: Incorporating more historical data, if available, or external data sources like weather forecasts or economic indicators could improve model accuracy, especially for predictive tasks influenced by external factors.

Model Ensembling: Combining the predictions from multiple models (e.g., an ensemble of different configurations of Gradient Boosting or a mix of different algorithm types) might reduce variance and improve the robustness of predictions.

Prediction Accuracy on Test Data

The Gradient Boosting Regressor's performance on the test data is indicative of how well the model can be expected to perform in real-world scenarios. With a test score of 0.7095, the model demonstrates a decent generalization from training to unseen data. This score suggests that while the model is fairly reliable in its predictions, there is still room for improvement, especially to bridge any gap seen between training performance and test performance. This gap often suggests that while the model is learning well, it might still be slightly overfitting the training data or not capturing all the nuances in the test set.

In comparison, similar strategies as discussed for the RandomForest Regressor could be employed here. Both models, while distinct in their methodologies and complexity, could benefit from similar strategies such as advanced feature engineering, more nuanced hyperparameter tuning, and exploring ensemble techniques to enhance predictive accuracy and robustness.

Top 10 prediction errors

```
         zone  year  month  day  predicted_load  true_load  \
804210     11  2005      3    5     3411.617577          6
2037970    11  2007      4   13     3230.820395         18
1683110    11  2007      4   13     3217.439779         18
7016790    11  2007      6    1     2965.648275         18
6882239    19  2005      9   27   566913.527460      21657
3850419    19  2007     10    3   560214.898834      21641
3140699    19  2007     10    3   557557.895356      22888
6689519    19  2007     10    4   544690.608299      22711
2785839    19  2007     10    3   544690.608299      24997
3495559    19  2007     10    3   571032.803791      27635

         relative_percentage_error
804210              -56760.292942
2037970             -17849.002195
1683110             -17774.665440
7016790             -16375.823751
6882239              -2517.691866
3850419              -2488.673808
3140699              -2336.027156
6689519              -2298.355899
2785839              -2079.023916
3495559              -1966.339077
```

Key Observations:

The relative percentage error shows substantial variations, including extremely large negative values indicating cases where the predicted load was much less than the actual load. For instance, the highest error exceeds -56760%, highlighting an enormous underestimation by the model.
Such large errors might suggest anomalies in the data or instances where the model failed to capture complex patterns possibly due to extreme values or unaccounted-for variables affecting load.
Comparison with RandomForest Regressor:

Magnitude of Errors: Both models exhibit large prediction errors in specific zones, but the nature and scale of errors differ. The Gradient Boosting model shows even more extreme values in its worst predictions compared to the RandomForest, which could be due to differences in how each model handles outliers and noise in the data.

Robustness: RandomForest might generally be more robust to overfitting due to its ensemble method that averages multiple decision trees, potentially providing more stable predictions across diverse data scenarios.

Sensitivity: Gradient Boosting can be more sensitive to noisy data and outliers, which might explain the larger errors. Its sequential method of correcting errors in predecessors can amplify mistakes if the initial trees fit to noise rather than underlying patterns.

Improvements:

Data Cleaning: Ensuring the data is clean, free of outliers, and well-preprocessed might help reduce these errors.

Feature Engineering: Adding more relevant features or transforming existing ones could help both models capture the load dynamics more accurately.

Model Tuning: Further tuning of hyperparameters, possibly using more refined techniques such as randomized search or Bayesian optimization, could help identify better model configurations.
The extreme errors highlighted in this analysis underscore the importance of understanding the model's behavior across different segments of the data and considering more complex or different modeling approaches if needed.

Summary table of the two machine learning models used:

Metric/Model	RandomForest Regressor	Gradient Boosting Regressor
General Training Score	0.7248	0.7178
General Validation Score	0.6969	0.7047
General Test Score	0.7024	0.7095
Robustness to Overfitting	More Robust	Less Robust
Sensitivity to Noise	Less Sensitive	More Sensitive
Handling of Extreme Values	Better Handling	Poorer Handling
Complexity of Model	Simpler	More Complex
Speed of Training	Faster	Slower
Top Prediction Errors	Large, but less extreme	Extremely Large Errors

This table highlights key performance metrics and characteristics of both models, providing a quick reference to evaluate their suitability depending on the specific requirements of prediction accuracy, training time, and handling of complex data scenarios.

Conclusion

The project successfully demonstrated the capability to forecast electricity load with a considerable degree of accuracy using machine learning algorithms. The RandomForestRegressor and Gradient Boosting Regressor both provided strong frameworks for the predictive tasks, with each model bringing unique strengths to the table. The analysis highlighted the importance of careful data preparation, feature engineering, and model selection in building effective predictive models.

Future improvements could include more advanced feature engineering, exploring additional ensemble techniques, and incorporating more external data to further enhance the models' accuracy and robustness. The project not only achieved its objectives but also laid down a solid foundation for future explorations into predictive modeling within the utility sector.

Appendix

```
import pandas as pd

# Load the data
load_data = pd.read_csv('Load_history_final.csv')
temp_data = pd.read_csv('Temp_history_final.csv')

# Print the column names to see what's available
print("Load Data Columns:", load_data.columns)
print("Temperature Data Columns:", temp_data.columns)


import pandas as pd

# Load the datasets
load_data = pd.read_csv('Load_history_final.csv')
temp_data = pd.read_csv('Temp_history_final.csv')

# Function to create a datetime column from year, month, and day
def create_datetime(df):
    # Ensure year, month, and day are integers and create a datetime column
    df['datetime'] = pd.to_datetime(df[['year', 'month', 'day']].astype(int))
    return df

# Apply the function to both datasets
load_data = create_datetime(load_data)
temp_data = create_datetime(temp_data)

# Exclude data from June 2008
load_data = load_data[~((load_data['datetime'].dt.month == 6) & (load_data['datetime'].dt.year == 2008))]
temp_data = temp_data[~((temp_data['datetime'].dt.month == 6) & (temp_data['datetime'].dt.year ==
2008))]

# Melting the data to convert hours columns into rows
def melt_data(df, id_vars, value_name):
    return df.melt(id_vars=id_vars, var_name='hour', value_vars=[f'h{i}' for i in range(1, 25)],
value_name=value_name)

load_data_melt = melt_data(load_data, ['zone_id', 'datetime'], 'load')
temp_data_melt = melt_data(temp_data, ['station_id', 'datetime'], 'temperature')

# Merge the datasets on datetime and hour
```

```
combined_data = pd.merge(load_data_melt, temp_data_melt, on=['datetime', 'hour'])

# Calculate correlations and determine the best station for each zone
zone_station_correlation = combined_data.groupby(['zone_id', 'station_id']).apply(lambda df:
df['load'].corr(df['temperature']))

# Reset index so we can access the correlation values
correlation_data = zone_station_correlation.reset_index(name='correlation')

# Handle NaN correlations that can occur if there is no variance in temperature or load
correlation_data = correlation_data.dropna(subset=['correlation'])

# Determine the best station for each zone based on the highest correlation
def get_best_station(group):
    return group.loc[group['correlation'].idxmax()]

best_stations = correlation_data.groupby('zone_id').apply(get_best_station)

# Output results
print("Best station for each load zone based on maximum correlation:")
print(best_stations[['zone_id', 'station_id', 'correlation']])

# Optionally, save the merged dataset for further analysis
combined_data.to_csv('combined_data.csv', index=False)

import pandas as pd

# Replace 'path_to_file' with the actual path to the CSV file on your local machine
combined_data = pd.read_csv('combined_data.csv')
print(combined_data.head())
print(combined_data.dtypes)


import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split

# Load the dataset
combined_data = pd.read_csv('combined_data.csv')
combined_data['datetime'] = pd.to_datetime(combined_data['datetime'])
combined_data['hour'] = pd.to_numeric(combined_data['hour'].str.extract('(\d+)')[0])

# Zone to station mapping based on best correlation
zone_station_map = {
    1: 8, 2: 10, 3: 1, 4: 8, 5: 8, 6: 1, 7: 1, 8: 6,
    9: 1, 10: 8, 11: 1, 12: 10, 13: 8, 14: 1, 15: 3,
    16: 10, 17: 1, 18: 1, 19: 2, 20: 8
}
```

```
# Store models and their scores for each zone
models = {}
model_scores = {}

for zone, station in zone_station_map.items():
    # Filter data for this zone and station
    zone_data = combined_data[(combined_data['zone_id'] == zone) & (combined_data['station_id'] ==
station)]

    # Apply specific filter for zone 15
    if zone == 15:
        zone_data = zone_data[zone_data['load'] < 200000]  # Exclude load values above 200,000 for zone
15

    # Features and target
    X = zone_data[['temperature', 'hour']]
    y = zone_data['load']

    # Split data into training/validation (80%) and test (20%) sets
    X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Further split the training/validation data into training (80%) and validation (20%) sets
    X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.25,
random_state=42)  # Note: 0.25 * 0.8 = 0.2


    # Initialize and train the Random Forest Regressor
    model = RandomForestRegressor(n_estimators=100, random_state=42)
    model.fit(X_train, y_train)

    train_score = model.score(X_train, y_train)

    # Validate the model
    val_score = model.score(X_val, y_val)

    # Test the model
    test_score = model.score(X_test, y_test)


    # Store the model and scores
    models[zone] = model
    model_scores[zone] = {'Training Score': train_score, 'Validation Score': val_score, 'Test Score':
test_score}

    # Print model performance for each zone
    print(f'Zone {zone}: Training Score = {train_score:.4f}, Validation Score = {val_score:.4f}, Test Score =
{test_score:.4f}')
```

```
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

# Load the dataset
combined_data = pd.read_csv('combined_data.csv')
combined_data['datetime'] = pd.to_datetime(combined_data['datetime'])
combined_data['hour'] = pd.to_numeric(combined_data['hour'].str.extract('(\d+)')[0])

# Zone to station mapping based on best correlation
zone_station_map = {
    1: 8, 2: 10, 3: 1, 4: 8, 5: 8, 6: 1, 7: 1, 8: 6,
    9: 1, 10: 8, 11: 1, 12: 10, 13: 8, 14: 1, 15: 3,
    16: 10, 17: 1, 18: 1, 19: 2, 20: 8
}

# Initialize accumulators for weighted average calculations
total_train_score = 0
total_val_score = 0
total_test_score = 0
total_samples_train = 0
total_samples_val = 0
total_samples_test = 0

for zone, station in zone_station_map.items():
    # Filter data for this zone and station
    zone_data = combined_data[(combined_data['zone_id'] == zone) & (combined_data['station_id'] ==
station)]

    # Apply specific filter for zone 15
    if zone == 15:
        zone_data = zone_data[zone_data['load'] < 200000]  # Exclude load values above 200,000 for zone
15

    # Features and target
    X = zone_data[['temperature', 'hour']]
    y = zone_data['load']

    # Split data into training/validation (80%) and test (20%) sets
    X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Further split the training/validation data into training (80%) and validation (20%) sets
    X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.25,
random_state=42)  # Note: 0.25 * 0.8 = 0.2

    # Initialize and train the Random Forest Regressor
    model = RandomForestRegressor(n_estimators=100, random_state=42)
    model.fit(X_train, y_train)
    # Calculate scores
    train_score = model.score(X_train, y_train)
    val_score = model.score(X_val, y_val)
    test_score = model.score(X_test, y_test)

    # Accumulate scores weighted by number of samples
    total_train_score += train_score * len(y_train)
    total_val_score += val_score * len(y_val)
    total_test_score += test_score * len(y_test)
    total_samples_train += len(y_train)
    total_samples_val += len(y_val)
    total_samples_test += len(y_test)

# Calculate weighted average scores
general_train_score = total_train_score / total_samples_train
general_val_score = total_val_score / total_samples_val
general_test_score = total_test_score / total_samples_test

# Output the general scores
print(f'General Training Score: {general_train_score:.4f}')
print(f'General Validation Score: {general_val_score:.4f}')
print(f'General Test Score: {general_test_score:.4f}')


import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import make_scorer, r2_score

# Load the dataset
combined_data = pd.read_csv('combined_data.csv')
combined_data['datetime'] = pd.to_datetime(combined_data['datetime'])
combined_data['hour'] = pd.to_numeric(combined_data['hour'].str.extract('(\d+)')[0])

# Zone to station mapping based on best correlation
zone_station_map = {
    1: 8, 2: 10, 3: 1, 4: 8, 5: 8, 6: 1, 7: 1, 8: 6,
    9: 1, 10: 8, 11: 1, 12: 10, 13: 8, 14: 1, 15: 3,
    16: 10, 17: 1, 18: 1, 19: 2, 20: 8
}

# Define hyperparameters for grid search
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [10, 20, 30],
    'min_samples_split': [2, 4, 6]
}
```

```
# Store models and their scores for each zone
models = {}
model_scores = {}

for zone, station in zone_station_map.items():
    # Filter data for this zone and station
    zone_data = combined_data[(combined_data['zone_id'] == zone) & (combined_data['station_id'] ==
station)]

    # Apply specific filter for zone 15
    if zone == 15:
        zone_data = zone_data[zone_data['load'] < 200000]  # Exclude load values above 200,000 for zone
15

    # Features and target
    X = zone_data[['temperature', 'hour']]
    y = zone_data['load']

    # Split data into training/validation (80%) and test (20%) sets
    X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Further split the training/validation data into training (80%) and validation (20%) sets
    X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.25,
random_state=42)

    # Initialize Grid Search with cross-validation
    grid_search = GridSearchCV(RandomForestRegressor(random_state=42), param_grid, cv=3,
scoring=make_scorer(r2_score))
    grid_search.fit(X_train, y_train)

    best_model = grid_search.best_estimator_

    # Evaluate the best model on the training, validation, and test sets
    train_score = best_model.score(X_train, y_train)
    val_score = best_model.score(X_val, y_val)
    test_score = best_model.score(X_test, y_test)

    # Store the best model and scores
    models[zone] = best_model
    model_scores[zone] = {'Training Score': train_score, 'Validation Score': val_score, 'Test Score':
test_score}

    # Print model performance and best parameters for each zone
    print(f'Zone {zone}: Best Parameters = {grid_search.best_params_}')
    print(f'Zone {zone}: Training Score = {train_score:.4f}, Validation Score = {val_score:.4f}, Test Score =
{test_score:.4f}')


import pandas as pd
```

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

# Load the dataset
combined_data = pd.read_csv('combined_data.csv')
combined_data['datetime'] = pd.to_datetime(combined_data['datetime'])
combined_data['hour'] = pd.to_numeric(combined_data['hour'].str.extract('(\d+)')[0])

# Zone to station mapping based on best correlation
zone_station_map = {
    1: 8, 2: 10, 3: 1, 4: 8, 5: 8, 6: 1, 7: 1, 8: 6,
    9: 1, 10: 8, 11: 1, 12: 10, 13: 8, 14: 1, 15: 3,
    16: 10, 17: 1, 18: 1, 19: 2, 20: 8
}

# Initialize accumulators for weighted average calculations
total_train_score = 0
total_val_score = 0
total_test_score = 0
total_samples_train = 0
total_samples_val = 0
total_samples_test = 0

for zone, station in zone_station_map.items():
    # Filter data for this zone and station
    zone_data = combined_data[(combined_data['zone_id'] == zone) & (combined_data['station_id'] ==
station)]

    # Apply specific filter for zone 15
    if zone == 15:
        zone_data = zone_data[zone_data['load'] < 200000]  # Exclude load values above 200,000 for zone
15

    # Features and target
    X = zone_data[['temperature', 'hour']]
    y = zone_data['load']

    # Split data into training/validation (80%) and test (20%) sets
    X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Further split the training/validation data into training (80%) and validation (20%) sets
    X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.25,
random_state=42)  # Note: 0.25 * 0.8 = 0.2

    # Initialize and train the Random Forest Regressor
    model = RandomForestRegressor(n_estimators=150, max_depth = 10, min_samples_split = 6,
random_state=42)
    model.fit(X_train, y_train)
    # Calculate scores
    train_score = model.score(X_train, y_train)
    val_score = model.score(X_val, y_val)
    test_score = model.score(X_test, y_test)

    # Accumulate scores weighted by number of samples
    total_train_score += train_score * len(y_train)
    total_val_score += val_score * len(y_val)
    total_test_score += test_score * len(y_test)
    total_samples_train += len(y_train)
    total_samples_val += len(y_val)
    total_samples_test += len(y_test)

# Calculate weighted average scores
general_train_score = total_train_score / total_samples_train
general_val_score = total_val_score / total_samples_val
general_test_score = total_test_score / total_samples_test

# Output the general scores
print(f'General Training Score: {general_train_score:.4f}')
print(f'General Validation Score: {general_val_score:.4f}')
print(f'General Test Score: {general_test_score:.4f}')


import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

# Load the dataset
combined_data = pd.read_csv('combined_data.csv')
combined_data['datetime'] = pd.to_datetime(combined_data['datetime'])
combined_data['hour'] = combined_data['datetime'].dt.hour

# Zone to station mapping based on best correlation
zone_station_map = {
    1: 8, 2: 10, 3: 1, 4: 8, 5: 8, 6: 1, 7: 1, 8: 6,
    9: 1, 10: 8, 11: 1, 12: 10, 13: 8, 14: 1, 15: 3,
    16: 10, 17: 1, 18: 1, 19: 2, 20: 8
}

error_list = []

for zone, station in zone_station_map.items():
    # Filter data for this zone and station
    zone_data = combined_data[(combined_data['zone_id'] == zone) & (combined_data['station_id'] ==
station)]
```

```
    # Apply specific filter for zone 15
    if zone == 15:
        zone_data = zone_data[zone_data['load'] < 200000]  # Exclude load values above 200,000 for zone
15

    # Features and target
    X = zone_data[['temperature', 'hour', 'datetime']]  # Include datetime for error tracking
    y = zone_data['load']

    # Split data into training/validation (80%) and test (20%) sets
    X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Further split the training/validation data into training (80%) and validation (20%) sets
    X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.25,
random_state=42)

    # Isolate datetime for later use before training
    datetime_test = X_test['datetime']
    X_train = X_train.drop(columns=['datetime'])
    X_val = X_val.drop(columns=['datetime'])
    X_test = X_test.drop(columns=['datetime'])

    # Initialize and train the Random Forest Regressor
    model = RandomForestRegressor(n_estimators=150, max_depth=10, min_samples_split=6,
random_state=42)
    model.fit(X_train, y_train)

    # Make predictions
    y_pred = model.predict(X_test)

    # Store errors
    errors = pd.DataFrame({
        'zone': zone,
        'predicted_load': y_pred,
        'true_load': y_test,
        'datetime': datetime_test
    })
    errors['relative_percentage_error'] = 100 * (errors['true_load'] - errors['predicted_load']) /
errors['true_load']
    error_list.append(errors)

# Combine all error data
errors_df = pd.concat(error_list)

# Calculate additional date components
errors_df['year'] = errors_df['datetime'].dt.year
errors_df['month'] = errors_df['datetime'].dt.month
errors_df['day'] = errors_df['datetime'].dt.day
errors_df['hour'] = errors_df['datetime'].dt.hour
```

```
# Sort by the magnitude of the relative percentage error, top 10
top_errors = errors_df.sort_values(by='relative_percentage_error', key=abs, ascending=False).head(10)

# Print the sorted top 10 errors
print(top_errors[['zone', 'year', 'month', 'day', 'predicted_load', 'true_load', 'relative_percentage_error']])



import pandas as pd
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

# Load the dataset
combined_data = pd.read_csv('combined_data.csv')
combined_data['datetime'] = pd.to_datetime(combined_data['datetime'])
combined_data['hour'] = pd.to_numeric(combined_data['hour'].str.extract('(\d+)')[0])

# Zone to station mapping based on best correlation
zone_station_map = {
    1: 8, 2: 10, 3: 1, 4: 8, 5: 8, 6: 1, 7: 1, 8: 6,
    9: 1, 10: 8, 11: 1, 12: 10, 13: 8, 14: 1, 15: 3,
    16: 10, 17: 1, 18: 1, 19: 2, 20: 8
}

# Initialize accumulators for weighted average calculations
total_train_score = 0
total_val_score = 0
total_test_score = 0
total_samples_train = 0
total_samples_val = 0
total_samples_test = 0

for zone, station in zone_station_map.items():
    # Filter data for this zone and station
    zone_data = combined_data[(combined_data['zone_id'] == zone) & (combined_data['station_id'] ==
station)]

    # Apply specific filter for zone 15
    if zone == 15:
        zone_data = zone_data[zone_data['load'] < 200000]  # Exclude load values above 200,000 for zone
15

    # Features and target
    X = zone_data[['temperature', 'hour']]
    y = zone_data['load']

    # Split data into training/validation (80%) and test (20%) sets
    X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Further split the training/validation data into training (80%) and validation (20%) sets
    X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.25,
random_state=42)

    # Initialize and train the Gradient Boosting Regressor
    model = GradientBoostingRegressor(n_estimators=100, random_state=42)
    model.fit(X_train, y_train)

    # Calculate scores
    train_score = model.score(X_train, y_train)
    val_score = model.score(X_val, y_val)
    test_score = model.score(X_test, y_test)

    # Accumulate scores weighted by number of samples
    total_train_score += train_score * len(y_train)
    total_val_score += val_score * len(y_val)
    total_test_score += test_score * len(y_test)
    total_samples_train += len(y_train)
    total_samples_val += len(y_val)
    total_samples_test += len(y_test)

# Calculate weighted average scores
general_train_score = total_train_score / total_samples_train
general_val_score = total_val_score / total_samples_val
general_test_score = total_test_score / total_samples_test

# Output the general scores
print(f'General Training Score: {general_train_score:.4f}')
print(f'General Validation Score: {general_val_score:.4f}')
print(f'General Test Score: {general_test_score:.4f}')



import pandas as pd
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import r2_score

# Load the dataset
combined_data = pd.read_csv('combined_data.csv')
combined_data['datetime'] = pd.to_datetime(combined_data['datetime'])
combined_data['hour'] = pd.to_numeric(combined_data['hour'].str.extract('(\d+)')[0])

# Zone to station mapping based on best correlation
zone_station_map = {
    1: 8, 2: 10, 3: 1, 4: 8, 5: 8, 6: 1, 7: 1, 8: 6,
    9: 1, 10: 8, 11: 1, 12: 10, 13: 8, 14: 1, 15: 3,
    16: 10, 17: 1, 18: 1, 19: 2, 20: 8
}

# Define the parameter grid for Grid Search
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [3, 5, 8],
    'learning_rate': [0.01, 0.1, 0.2]
}

# Initialize accumulators for weighted average calculations
total_train_score = 0
total_val_score = 0
total_test_score = 0
total_samples_train = 0
total_samples_val = 0
total_samples_test = 0

for zone, station in zone_station_map.items():
    # Filter data for this zone and station
    zone_data = combined_data[(combined_data['zone_id'] == zone) & (combined_data['station_id'] ==
station)]

    # Apply specific filter for zone 15
    if zone == 15:
        zone_data = zone_data[zone_data['load'] < 200000]  # Exclude load values above 200,000 for zone
15

    # Features and target
    X = zone_data[['temperature', 'hour']]
    y = zone_data['load']

    # Split data into training/validation (80%) and test (20%) sets
    X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Further split the training/validation data into training (80%) and validation (20%) sets
    X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.25,
random_state=42)

    # Set up GridSearchCV to find the best parameters
    grid_search = GridSearchCV(GradientBoostingRegressor(random_state=42), param_grid, cv=3,
scoring='r2')
    grid_search.fit(X_train, y_train)

    # Retrieve the best estimator
    best_model = grid_search.best_estimator_

    # Calculate scores
    train_score = best_model.score(X_train, y_train)
    val_score = best_model.score(X_val, y_val)
    test_score = best_model.score(X_test, y_test)

    # Accumulate scores weighted by number of samples
    total_train_score += train_score * len(y_train)
    total_val_score += val_score * len(y_val)
    total_test_score += test_score * len(y_test)
    total_samples_train += len(y_train)
    total_samples_val += len(y_val)
    total_samples_test += len(y_test)

    # Output best parameters and scores for each zone
    print(f'Zone {zone}: Best Parameters = {grid_search.best_params_}')
    print(f'Zone {zone}: Training Score = {train_score:.4f}, Validation Score = {val_score:.4f}, Test Score =
{test_score:.4f}')

# Calculate weighted average scores
general_train_score = total_train_score / total_samples_train
general_val_score = total_val_score / total_samples_val
general_test_score = total_test_score / total_samples_test

# Output the general scores
print(f'General Training Score: {general_train_score:.4f}')
print(f'General Validation Score: {general_val_score:.4f}')
print(f'General Test Score: {general_test_score:.4f}')


import pandas as pd
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

# Load the dataset
combined_data = pd.read_csv('combined_data.csv')
combined_data['datetime'] = pd.to_datetime(combined_data['datetime'])
combined_data['hour'] = pd.to_numeric(combined_data['hour'].str.extract('(\d+)')[0])

# Zone to station mapping based on best correlation
zone_station_map = {
    1: 8, 2: 10, 3: 1, 4: 8, 5: 8, 6: 1, 7: 1, 8: 6,
    9: 1, 10: 8, 11: 1, 12: 10, 13: 8, 14: 1, 15: 3,
    16: 10, 17: 1, 18: 1, 19: 2, 20: 8
}

# Initialize accumulators for weighted average calculations
total_train_score = 0
total_val_score = 0
total_test_score = 0
total_samples_train = 0
```

```
total_samples_val = 0
total_samples_test = 0

for zone, station in zone_station_map.items():
    # Filter data for this zone and station
    zone_data = combined_data[(combined_data['zone_id'] == zone) & (combined_data['station_id'] ==
station)]

    # Apply specific filter for zone 15
    if zone == 15:
        zone_data = zone_data[zone_data['load'] < 200000]  # Exclude load values above 200,000 for zone
15

    # Features and target
    X = zone_data[['temperature', 'hour']]
    y = zone_data['load']

    # Split data into training/validation (80%) and test (20%) sets
    X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Further split the training/validation data into training (80%) and validation (20%) sets
    X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.25,
random_state=42)

    # Initialize and train the Gradient Boosting Regressor
    model = GradientBoostingRegressor(learning_rate = 0.2, max_depth = 3, n_estimators = 150,
random_state=42)
    model.fit(X_train, y_train)

    # Calculate scores
    train_score = model.score(X_train, y_train)
    val_score = model.score(X_val, y_val)
    test_score = model.score(X_test, y_test)

    # Accumulate scores weighted by number of samples
    total_train_score += train_score * len(y_train)
    total_val_score += val_score * len(y_val)
    total_test_score += test_score * len(y_test)
    total_samples_train += len(y_train)
    total_samples_val += len(y_val)
    total_samples_test += len(y_test)

# Calculate weighted average scores
general_train_score = total_train_score / total_samples_train
general_val_score = total_val_score / total_samples_val
general_test_score = total_test_score / total_samples_test

# Output the general scores
print(f'General Training Score: {general_train_score:.4f}')
print(f'General Validation Score: {general_val_score:.4f}')
print(f'General Test Score: {general_test_score:.4f}')



import pandas as pd
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

# Load the dataset
combined_data = pd.read_csv('combined_data.csv')
combined_data['datetime'] = pd.to_datetime(combined_data['datetime'])
combined_data['hour'] = combined_data['datetime'].dt.hour  # Extract hour from datetime directly

# Zone to station mapping based on best correlation
zone_station_map = {
    1: 8, 2: 10, 3: 1, 4: 8, 5: 8, 6: 1, 7: 1, 8: 6,
    9: 1, 10: 8, 11: 1, 12: 10, 13: 8, 14: 1, 15: 3,
    16: 10, 17: 1, 18: 1, 19: 2, 20: 8
}

error_list = []

for zone, station in zone_station_map.items():
    # Filter data for this zone and station
    zone_data = combined_data[(combined_data['zone_id'] == zone) & (combined_data['station_id'] ==
station)]

    # Apply specific filter for zone 15
    if zone == 15:
        zone_data = zone_data[zone_data['load'] < 200000]  # Exclude load values above 200,000 for zone
15

    # Features and target
    X = zone_data[['temperature', 'hour', 'datetime']]  # Include datetime for error tracking
    y = zone_data['load']

    # Split data into training/validation (80%) and test (20%) sets
    X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Further split the training/validation data into training (80%) and validation (20%) sets
    X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.25,
random_state=42)

    # Isolate datetime for later use before training
    datetime_test = X_test['datetime']
    X_train = X_train.drop(columns=['datetime'])
    X_val = X_val.drop(columns=['datetime'])
    X_test = X_test.drop(columns=['datetime'])

    # Initialize and train the Gradient Boosting Regressor
    model = GradientBoostingRegressor(learning_rate=0.2, max_depth=3, n_estimators=150,
random_state=42)
    model.fit(X_train, y_train)

    # Make predictions
    y_pred = model.predict(X_test)

    # Store errors
    errors = pd.DataFrame({
        'zone': zone,
        'predicted_load': y_pred,
        'true_load': y_test,
        'datetime': datetime_test
    })
    errors['relative_percentage_error'] = 100 * (errors['true_load'] - errors['predicted_load']) /
errors['true_load']
    error_list.append(errors)

# Combine all error data
errors_df = pd.concat(error_list)

# Calculate additional date components
errors_df['year'] = errors_df['datetime'].dt.year
errors_df['month'] = errors_df['datetime'].dt.month
errors_df['day'] = errors_df['datetime'].dt.day

# Sort by the magnitude of the relative percentage error, top 10
top_errors = errors_df.sort_values(by='relative_percentage_error', key=abs, ascending=False).head(10)

# Print the sorted top 10 errors
print(top_errors[['zone', 'year', 'month', 'day', 'predicted_load', 'true_load', 'relative_percentage_error']])
```