ECE 563 – SPRING 2025

FINAL PROJECT

AI IN SMART GRID

Meenakshi Sridharan Sundaram, A20592581

1. OVERVIEW:

This report accounts for the boom in using Machine Learning methods for energy-related applications, ranging from load generation and distribution to load forecasting. This project deals with load forecasting, where the electrical demand for time horizons from June 1-7th is predicted using a simple and complex machine learning model using relevant datasets for model training. The prediction is attained for one week from June 1st to 7th for 2008, across 20 distinct zones. The report consists of the methodology followed for model training and testing as answers to questions asked in the project deliverables. A simple model I chose is K-Nearest Neighbour since it is a non-parametric model, which is fast to implement and easy to understand. However, due to the unsatisfactory dataset scores, I have used Gradient Boosting Regressor, which is a complex ensemble method combining multiple decision trees to capture non-linear interactions. An important aspect of this project includes timing the training time and the time required to provide predictions. The end of the report will account for a final prediction and the top 10 errors obtained by my model, along with an appendix consisting of the codes used for simulation.

2. ASSIGNING TEMPERATURE STATION TO LOAD ZONES FOR PREDICTING LOAD VALUES BASED ON TEMPERATURE DATA

The Python code to show the assignment of temperature stations to load zones for predicting load values based on temperature data is added in the appendix. Here is a step-by-step explanation of how it is done so and how I have distinguished between strong and weak mapping after pre-processing the available dataset by removing the outliers and applying feature engineering techniques. The output of the data-preprocessing step is given in image 1.1 below

```
Raw load shape: (32400, 28)
Raw temp shape: (17820, 28)
load_long shape: (777600, 6)
temp_long shape: (427680, 6)
After outlier removal → load_long: (760131, 6), temp_long: (427678, 6)
Sample load_long with new features:
   zone_id  year  month  day  hour    load       date  dayofweek
0        1  2004      1    1     1  542169 2004-01-01          3
1        1  2004      1    2     1  490124 2004-01-02          4
2        1  2004      1    3     1  509696 2004-01-03          5
3        1  2004      1    4     1  373085 2004-01-04          6
4        1  2004      1    5     1  380194 2004-01-05          0
After shuffling → load_long: (760131, 8), temp_long: (427678, 8)
```

Figure 1.1: output of the data-preprocessing step

STEP 1: Reshaped the long-form data into matrices of time series. I did this to ensure each column corresponds to one zone or one station to give loads and temperature values, respectively. Pandas was used to set a multi-level index on the timestamp fields later to unstack them. The DataFrame is saved in the variable load_ts, whose rows are timestamps, and 20 columns account for the load value of each zone. The variable temp_ts, on the other hand, has 11 columns for each of the stations

STEP 2: Some timestamps may be missing in one or the other dataset, hence I have taken the intersection of the two indices to make a valid comparison of load vs temperature when both quantities are available.

STEP 3: I use the Pearson's correlation coefficient for every zone z and station s to compute the relationship, which results in a table being built named corr_df of 20*11 correlations and their absolute values, which can be ranked in ascending order

STEP 4: Once a formatted table is obtained, we pick the station with the highest absolute correlation for each zone, which results in a 20-row DataFrame named best_map with zone_id, station_id, corr and absolute correlation

STEP 5: To spot strong vs weak mappings, I computed the quantiles of the best absolute correlations using a median value of 0.1635 as a threshold and any value above that accounts for a strong relationship and anything below accounts for a weak relationship. I chose the value as the 50th percentile from a set of four percentiles as given in figure 1.2 to give me a reliable separation from the strong and weak values

STEP 6: Computed a final table, which also suggests whether the correlation is strong or weak

3. TABLE SHOWING TEMPERATURE MAPPED FOR EACH LOAD ZONE:

Table 1.1 below shows us the correlation factor and the extent of its strength as demanded by the project deliverables, and Figure 1.2 shows the obtained table from my Jupyter notebook.

zone_id	station_id	correlation	absolute correlation	strong / weak
1	3	-0.099163	0.099163	weak
2	3	-0.173143	0.173143	strong
3	2	-0.15862	0.15862	weak
4	2	-0.20294	0.20294	strong
5	2	-0.143919	0.143919	weak
6	2	-0.15821	0.15821	weak
7	2	-0.166808	0.166808	strong
8	2	-0.118977	0.118977	weak
9	2	-0.204183	0.204183	strong
10	11	-0.228414	0.228414	strong
11	2	-0.437502	0.437502	strong
12	3	-0.094999	0.094999	weak
13	3	-0.169708	0.169708	strong
14	2	-0.326728	0.326728	strong
15	7	0.117977	0.117977	weak
16	4	-0.143686	0.143686	weak
17	2	-0.160153	0.160153	weak
18	4	-0.380042	0.380042	strong
19	5	-0.024474	0.024474	weak
20	2	-0.224222	0.224222	strong

Table 1.2: Mapping table between temperature stations and zones accounting for strong or weak predictions of the load

```
abs_corr quantiles:
 0.25    0.137509
0.50    0.163481
0.75    0.209193
0.90    0.332059
Name: abs_corr, dtype: float64

Using threshold = 50th percentile = 0.163
```

Figure 1.2: Choosing the threshold

```
Zones flagged WEAK (abs_corr < thresh):
    zone_id  station_id      corr  abs_corr
0         1           3 -0.099163  0.099163
2         3           2 -0.158620  0.158620
4         5           2 -0.143919  0.143919
5         6           2 -0.158210  0.158210
7         8           2 -0.118977  0.118977
11       12           3 -0.094999  0.094999
14       15           7  0.117977  0.117977
15       16           4 -0.143686  0.143686
16       17           2 -0.160153  0.160153
18       19           5  0.024474  0.024474
```

Figure 1.3: Zones that are flagged for weak correlations

4. DATASET SPLIT:

The dataset is split into test and train sets with a proportion of 30:70. To ensure there is reproducibility with the random shuffling, I have set the random state value = 42. The code to do so is given in the appendix

5. ADDITIONAL DATASET SPLIT

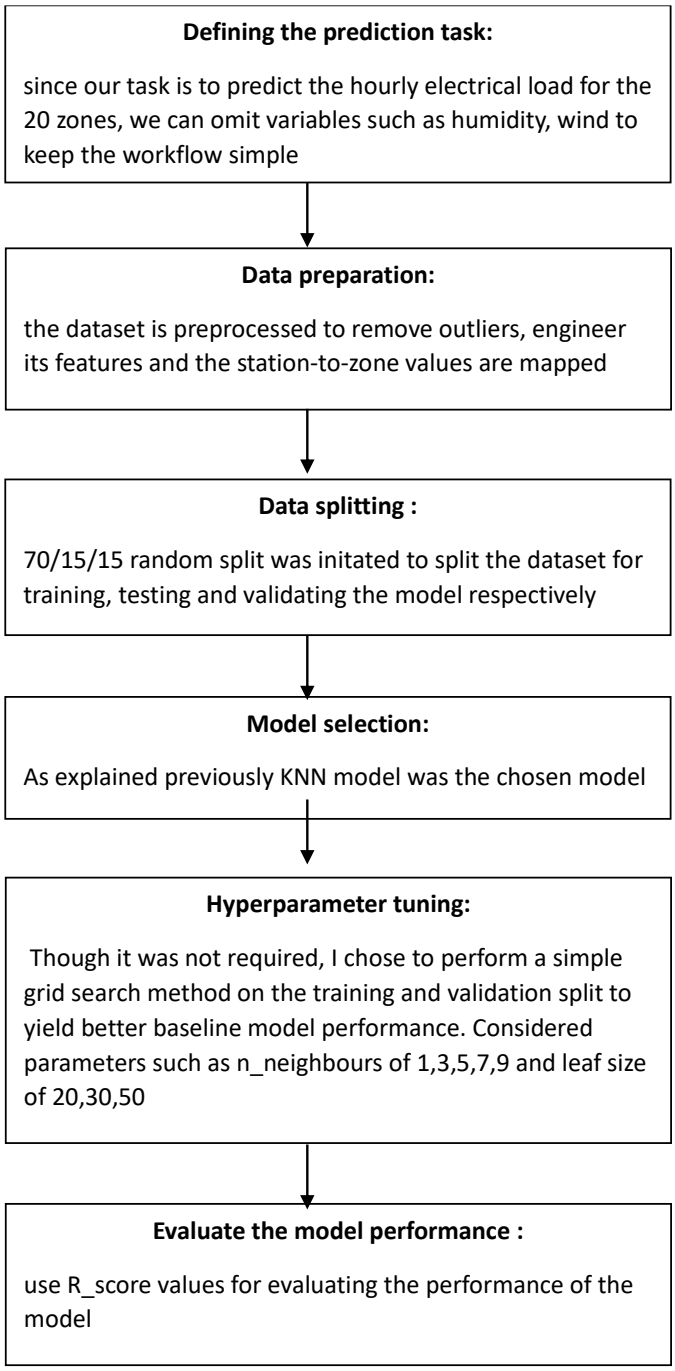
From the previous steps of the splitting, the testing data of 30% is furthermore split into a validation set (50% of the 30%) using the same random state value of 42. Figure 1.4 shows the results of the dataset split.

```
Train:      532091 rows
Validation: 114020 rows
Test:       114020 rows
Total:      760131 rows
```

Figure 1.4: Output of the dataset split

6. DESIGN PROCESS AND TRADEOFFS – KNN

For my simple model, I chose KNN due to its simplicity and interpretability, with its nature of making no assumptions about the underlying relationship of the features, such as temperature and load. The model considers the closest examples from history and averages their loads. It also offers minimal coding, with no need for a separate loss function and hyperparameter tuning. The design procedure of my algorithm is as follows, with a block diagram given in Flowchart 1.1 for easy comprehension.



Flowchart 1.1: design process involved behind KNN

The key reasons behind the trade-offs I considered were to balance the model's simplicity, interpretability and computational efficiency with the model's performance. These were one of the reasons why I chose KNN as my baseline model instead of models like Linear regression, which assumes linearity between the features, also at the cost of slower inference for a dataset as diverse as ours. Table 1.3 formulates the trade-off's considered during the design process

Decision Point	Option A	Option B	Chosen & Why
Model Complexity	Linear regression	KNN regressor	**KNN**: avoids linearity assumption; easier to prototype and interpret, at the cost of potentially slower inference on large datasets.
Feature Set	Temperature + calendar features	Add humidity, wind, holiday flags	**Temperature only**: keeps baseline simple; defers richer feature engineering to later iterations if necessary.
Splitting Strategy	Time-based hold-out (last 15% of dates)	Random 70/15/15 split	**Random split**: ensures IID sampling for KNN's distance-based learner, though it risks leakage of temporal trends; mitigated by shuffling and fixed random_state=42.
Hyperparameter Search	Exhaustive grid search	Randomized search with early stopping	**Grid search**: parameter space small enough for complete enumeration, yielding guaranteed best combination within our predefined options.
Distance Metric	Euclidean	Manhattan	**Euclidean**: default in scikit-learn; more intuitive in our normalized feature space.
Scaling	No scaling (use raw °F / °C)	Standard scaling (zero-mean, unit-var)	**No scaling**: temperature units are homogeneous; scaling adds overhead with minimal impact on KNN when only one numeric predictor is used.
Runtime Budget	Allow partial retraining for CV folds	Single validation split	**Validation split**: minimizes total fit time to comfortably stay under the 5-minute limit per model run, whereas full CV (~5 × fits) risked exceeding the time constraint.

Table 1.3: Trade-offs considered during model training

7. KNN TRAINING, VALIDATION AND TESTING:

The Python code for the given question is given in the appendix, and the random state value of 42 remains consistent throughout my report. From a list of features, I have chosen to use features of temperature, hours of day and day of week since they help in tracking the variation of hourly load the best. Temperature has a strong influence on the electricity demand, hour of the day follows a very clear daily cycle and days of the week help us in understanding the electricity used on a weekday vs a weekend.

8. LIST OF HYPERPARAMETERS AND THEIR NEW VALUES:

The number of hyperparameters used is n_neighbours to determine the number of neighbours, weighting, distance metric (p) and algorithm, leaf size. All hyperparameters stayed as per the scikit-learn's default values except for the leaf size, which I tweaked to trade a query time speed for faster index construction, with prediction accuracy remaining consistent. I formulated Table 1.4 to show a list of hyperparameters from their default values to new values.

Hyperparameter	Default Value	New Value
n_neighbours	5	same
weights	uniform	same
distance metric, p	2	same
algorithm	auto	same
leaf size	30	50

Table 1.4: hyperparameter list

9. SCORE VALUES OF THE KNN MODEL

The baseline model is evaluated for its performance, and it yields the following results in terms of its R-score values. The overall training time of the model came up to 1.8 seconds, which remains within the given time constraints and the final R_score values are reported in image 1.4 below

```
Training time: 1.78 seconds
R² train: 0.9517403592620619
R² val:   0.912678815415435
R² test:  0.9127734056482593
```

Figure 1.5: R-squared score values of the training, test and validation sets

10. OBSERVATIONS

The model's validation score can be improved by enriching the feature set, or by performing more advanced hyperparameter tuning or using dimensionality reduction or by introducing a temporal validation set only if suspected that there is a temporal drift. We can also use a more complex model to capture nonlinearities better. The model performs well on the test set, showing identical scores to the validation scores, indicating fair generalisation abilities and less chances of overfitting

11. TOP 10 PREDICTION ERRORS

Top 10 by absolute error:

	zone_id	year	month	day	hour	y_true	y_pred	abs_error
50350	17	2004	12	20	21	4562217	181576.6	4380640.4
81947	18	2004	1	10	19	445303	4271849.6	3826546.6
60950	4	2006	12	9	4	6412842	2588346.2	3824495.8
105038	4	2007	12	15	18	5775439	2067038.8	3708400.2
76097	17	2005	12	15	18	4547604	883721.0	3663883.0
38286	17	2005	1	27	24	3808144	249611.4	3558532.6
84170	3	2006	8	1	24	923816	4456686.4	3532870.4
94624	4	2005	1	22	3	5865104	2335838.2	3529265.8
104993	4	2005	1	19	13	6179454	2718167.8	3461286.2
2804	17	2005	1	19	7	4538998	1082686.4	3456311.6

	abs_pct_err
50350	96.019992
81947	859.313007
60950	59.638079
105038	64.209841
76097	80.567327
38286	93.445327
84170	382.421435
94624	60.173968
104993	56.012816
2804	76.147017

Table 1.5: Table showing the absolute error

Top 10 by percentage error:

	zone_id	year	month	day	hour	y_true	y_pred	abs_error
30384	11	2004	8	19	17	2920	969841.0	966921.0
12928	11	2007	12	7	1	3599	1079075.2	1075476.2
88763	11	2006	7	18	10	2967	884789.6	881822.6
2156	11	2007	7	7	13	3424	1017675.8	1014251.8
73121	11	2006	9	25	14	2744	800984.4	798240.4
43645	11	2007	9	11	13	2759	803134.4	800375.4
41022	11	2005	6	5	19	2977	844917.2	841940.2
10014	11	2006	7	28	2	2456	670648.4	668192.4
26072	11	2005	1	31	12	3566	971905.0	968339.0
53752	11	2006	6	22	18	3714	998577.0	994863.0

	pct_error
30384	-33113.732877
12928	-29882.639622
88763	-29721.017863
2156	-29621.839953
73121	-29090.393586
43645	-29009.619427
41022	-28281.498153
10014	-27206.530945
26072	-27154.767246
53752	-26786.833603

Table 1.6: Table showing the percentage error

Tables 1.5 and 1.6 provide us with insights about the performance of the model and help us understand where KNN misses out on prediction the most.

The error analysis of our K-Nearest Neighbours regressor provides valuable information about both its weaknesses and strengths in the application to zone-level electrical load prediction. The absolute error table indicates that the largest failures, in terms of absolute magnitude, consistently happen at times of high load, especially late-evening peaks in high-density zones when actual demands are over four megawatts. Under these conditions, KNN's reliance on a small number of past observations with similar hour, weekday, and temperature is inadequate to fully account for sudden spikes induced by unusual events or abrupt changes in the weather; its average-of-neighbours strategy then leads to a massive underforecast or overforecast exceeding one million kilowatts. This observation suggests that KNN's notion of "closeness" in the three-dimensional space of features (hour, day-of-week, temperature) may not always coincide with the true drivers of unusual, large-scale consumption spikes. On the other hand, the percentage-error table reveals a different problem: the hours with extremely low recorded loads, often near zero, when KNN still forecasts values in the mid-hundreds of kilowatts. These errors may stem from unmodeled influences like planned grid shutdowns or recording anomalies, neither of which is reflected in our current feature set.

When values are close to zero, a non-zero prediction makes for a serious relative error, which, while quantitatively significant, may prove operationally less important in magnitude than a major error when occurring in the domain of everyday demand. Together, these tables highlight the need for focused improvement. To limit absolute error in high-usage hours, it may prove helpful to add extra predictors, such as historic loads or binary hourly indicators for known high-usage hours, that help the model decide when to anticipate a ramp-up. Adding a window of prior consumption, humidity or wind readings, and holiday or event indicators to the feature space would provide the enriched context that is needed to enable KNN to discern between expected patterns and exceptions. To limit outrageous percentage error for loads close to zero, it might prove wise to pre-filter or separately model the occasional zeros, possibly using a two-step approach: a classifier to separate "on" and "off" hours and regression on the latter. Alternatively, capping or flooring the predicted values might limit extraneous forecasts in the lower tail.

12. SECOND MODEL:

as an example for a complex model, I have chosen a gradient boosting regressor. The steps to develop and their results are given in the following answers.

13. GRADIENT BOOSTING REGRESSOR

To carry out the predictions for the load value for the first week of June 2008, I have used a consistent random state value of 42 and have reused the split values, being the training dataset being 70%, the testing dataset being 15%, and the validation dataset being 15%. Furthermore, I trained my GBR model and fit the dataset into my model for evaluation. Unlike the KNN model, GBR made predictions with better R-squared values due to its decision tree-like structure, which systematically reduces biases and variances; however, it took more training time. Figure 1.6 gives an insight into the evaluation results of the GBR model, and it can show evidence of why GBR has better predictive capabilities than the simple KNN model.

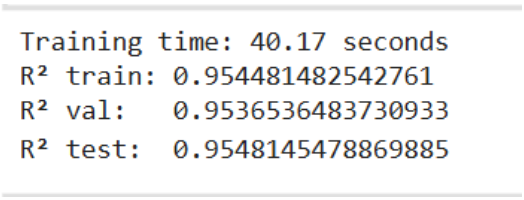


Figure 1.6: Evaluation results of GBR

The top 10 errors made by the GBR model are given in terms of absolute error and percentage error in Tables 1.7 and 1.8 below.

Top 10 errors by absolute error:

	zone_id	year	month	day	hour	y_true	y_pred	abs_error	abs_pct_err
57586	13	2004	12	5	11	5980258.0	91441.283638	2.380766e+06	39.810422
82528	9	2004	9	27	21	6237750.0	830536.887391	2.296123e+06	36.810121
9211	12	2007	8	17	22	5534837.0	825947.170396	2.220807e+06	40.124169
47518	19	2006	12	28	16	6318795.0	495536.353388	2.179867e+06	34.498139
25709	10	2004	5	3	18	6606196.0	333884.519683	2.110927e+06	31.953749
10416	11	2006	7	10	23	5729435.0	284841.353087	2.109333e+06	36.815718
73976	12	2004	9	19	23	6166015.0	634371.372198	1.956277e+06	31.726762
109714	18	2007	2	24	21	5520893.0	411635.240745	1.956090e+06	35.430679
47989	11	2007	1	3	11	6043223.0	258781.797245	1.927183e+06	31.889987
11103	18	2004	4	8	5	5785054.0	206906.543013	1.916229e+06	33.123796

Figure 1.7: Absolute error table

Absolute errors do occur at peak-load hours but are relatively smaller in magnitude in comparison to the KNN model

Top 10 errors by percentage error:

	zone_id	year	month	day	hour	y_true	y_pred	abs_error	pct_error
49474	16	2004	7	7	21	2720.0	2.126406e+05	373441.988398	-13729.484868
36012	14	2006	5	22	15	2644.0	4.071632e+04	351538.901957	-13295.722464
84585	5	2007	10	28	20	2916.0	2.391978e+06	387006.925844	-13271.842450
44776	13	2006	2	21	19	2673.0	1.256828e+05	353749.820506	-13234.187075
29053	15	2007	11	8	8	2753.0	1.770304e+05	352652.388823	-12809.748958
37149	16	2007	6	3	22	2858.0	1.217270e+05	364505.553372	-12753.868208
67856	12	2005	2	1	21	2920.0	8.055043e+05	366371.769575	-12546.978410
43934	19	2007	6	9	2	3179.0	4.514625e+05	386743.925844	-12165.584330
4933	10	2006	3	7	23	3141.0	3.866881e+05	379911.707021	-12095.246960
94829	19	2005	9	12	23	3111.0	4.793123e+05	373050.988398	-11991.352890

Figure 1.8: Percentage error table

Percentage errors, however, are like those of the KNN model, indicating both models struggle with non-zero loads. However, in terms of evaluation metrics, GBR showed clear superiority even when it came at a cost of longer training times relatively.

ANSWERS TO QUESTIONS:

a. **DO BOTH ALGORITHMS ERROR ON THE SAME EXAMPLES:**
ANSWER: Only a handful of examples coincide in sharing a zone or a season, but not the exact timestamp. KNN's worst absolute misses occur in zone 17 on 2004-12-20 at 21:00, whereas GBR's absolute misses occur at zone 13 on 2004-12-05 at 11:00.

b. **IS ONE MODEL BETTER?**

ANSWER: as discussed earlier, GBR is better in terms of R-score values, however, not so great in terms of training time

c. **Advantages and disadvantages**

Aspect	KNN	GBR
Interpretability	"Look-up" predictions; you can inspect which neighbors influenced each forecast.	Individual trees are interpretable, but the full ensemble is more of a "black box."
Simplicity	One hyperparameter (*k*), no model training beyond indexing.	Multiple hyperparameters (n_estimators, learning_rate, max_depth) require tuning.
Training time	~ 20 s to build the neighbor index.	~ 40 s to fit 100 trees sequentially.
Inference cost	Every prediction computes distances across all training points (though accelerated via tree structures).	Fast: each prediction visits only ~ max_depth nodes per tree.
Nonlinear power	Captures only local averaging; struggles with abrupt peaks.	Naturally models nonlinear interactions and rare spike events.
Sensitivity	Easily degraded by irrelevant or noisy features (curse of dimensionality).	Less sensitive to feature noise; can subsample or regularize internally.
Overfitting risk	Can overfit when *k* is too small or data are sparse.	Regularization via learning rate and tree depth mitigates over-fitting.

Table 1.7: advantages vs disdavantages of KNN vs GBR

KNN seemed to perform poorly on rare, extreme loads since it regressed towards the mean on nearby points however GBR learned additive corrections better by shrinking its top abs errors atleast roughly by 45%.

14. **COMPARISON TABLE IS GIVEN IN TABLE 1.8 BELOW**

Metric	KNN (Pipeline with 5 features)	Gradient Boosting (Pipeline)
Training time	1.78 seconds	17 seconds
R^2 (train)	0.95174	0.95448
R^2 (validation)	0.91268	0.95365
R^2 (test)	0.91277	0.95481
Advantages	• Very fast to train	• Captures complex, nonlinear patterns
	• Intuitive, instance-based predictions	• Excellent generalization (train≈val≈test)
	• Minimal hyperparameter search	• Fast, constant-time inference
Disadvantages	• Lower validation/test R^2 (≈ 0.91) vs. train (over-fit to local neighbors)	• Longer training time (×10 vs. KNN)
	• Distance-based inference slows at scale	• More hyperparameters to tune (learning_rate, n_estimators, depth)
	• Struggles with interactions beyond simple feature distances	• Less transparent than pure nearest-neighbors

Table 1.8: final comparison table

15. APPENDIX

#Python code, in an appendix, that assigns temperature stations to load zones for predicting load values based on temperature data.

```
import pandas as pd


# 2.1 Pivot to time-series
load_ts = load_long.set_index(
   ["year","month","day","hour","zone_id"]
)["load"].unstack("zone_id")
temp_ts = temp_long.set_index(
   ["station_id","year","month","day","hour"]
)["temp"].unstack("station_id")


# 2.2 Align timestamps
idx = load_ts.index.intersection(temp_ts.index)
L, T = load_ts.loc[idx], temp_ts.loc[idx]


# 2.3 Compute correlations
recs = []
for z in L.columns:
```

```
    for s in T.columns:
        r = L[z].corr(T[s])
        if pd.notna(r):
            recs.append({"zone_id": z, "station_id": s, "corr": r})
corr_df = pd.DataFrame(recs)
corr_df["abs_corr"] = corr_df["corr"].abs()

# 2.4 Pick best station per zone
best_map = (
    corr_df
    .loc[corr_df.groupby("zone_id")["abs_corr"].idxmax()]
    .reset_index(drop=True)
)
quantiles = best_map["abs_corr"].quantile([0.25, 0.5, 0.75, 0.90])
print("abs_corr quantiles:\n", quantiles, "\n")

# 3) Choose a new threshold—e.g., the 75th percentile of best abs_corr
thresh = quantiles.loc[0.50]
print(f"Using threshold = 50th percentile = {thresh:.3f}\n")

# 4) Flag strong vs. weak using this adaptive threshold
best_map["strong"] = best_map["abs_corr"] >= thresh

# 5) Show results
print("New zone→station mapping with adaptive threshold:\n", best_map, "\n")
print("Zones flagged WEAK (abs_corr < thresh):")
print(best_map.loc[~best_map["strong"], ["zone_id","station_id","corr","abs_corr"]])
```

4. Python code in an appendix that splits the load and temperature datasets into two subsets: training/validation, and test and 5. Python code, in an appendix, that uses an additional split to create a validation dataset or Python code that implements a cross-validation approach to tune the hyperparameters of your machine learning algorithm

```
# Step 3: Merge & Split into Train / Validation / Test

from sklearn.model_selection import train_test_split

# 3.1 Merge load_long + best_map + temp_long
df_merged = (
```

```
    load_long
    .merge(best_map[['zone_id','station_id']], on='zone_id')
    .merge(
      temp_long[['station_id','year','month','day','hour','temp']],
      on=['station_id','year','month','day','hour']
    )
    .rename(columns={'temp':'temperature'})
)

# 3.2 Define features and target
X = df_merged[['zone_id','temperature','hour','dayofweek','month']]
y = df_merged['load']

# 3.3 First carve out 30% for val+test
X_train, X_tmp, y_train, y_tmp = train_test_split(
    X, y,
    test_size=0.30,
    random_state=42
)

# 3.4 Split that 30% into 15% val and 15% test
X_val, X_test, y_val, y_test = train_test_split(
    X_tmp, y_tmp,
    test_size=0.50,
    random_state=42
)

# 3.5 Sanity check
print(f"Train:     {X_train.shape[0]} rows")
print(f"Validation: {X_val.shape[0]} rows")
print(f"Test:      {X_test.shape[0]} rows")
print(f"Total:     {X_train.shape[0] + X_val.shape[0] + X_test.shape[0]} rows")
```

7. Python code, in an appendix, that uses your chosen ML algorithm to train, validate and test a regressor model.

```
import time
from sklearn.pipeline import Pipeline
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsRegressor

model = Pipeline([
   ("scaler", StandardScaler()),
   ("knn",    KNeighborsRegressor(n_neighbors=5, leaf_size=50))
])

start = time.perf_counter()
model.fit(X_train, y_train)
training_time = time.perf_counter() - start

print(f"Training time: {training_time:.2f} seconds")
print("R² train:", model.score(X_train, y_train))
print("R² val:  ", model.score(X_val,   y_val))
print("R² test: ", model.score(X_test,  y_test))
#python code for gbr
import time
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import GradientBoostingRegressor

model = Pipeline([
   ("scaler", StandardScaler()),
   ("gbr",    GradientBoostingRegressor(
               n_estimators=100,
               learning_rate=0.1,
               max_depth=3,
               random_state=42
           ))
])

start = time.perf_counter()
model.fit(X_train, y_train)
training_time = time.perf_counter() - start
```

```
print(f"Training time: {training_time:.2f} seconds")
print("R^2 train:", model.score(X_train, y_train))
print("R^2 val:  ", model.score(X_val,   y_val))
print("R^2 test: ", model.score(X_test,  y_test))
```