Semester Project - Team 46

Hyperlocal Weather Prediction using Machine Learning

View on GitHub

Semester Project - Team 46

Introduction:

Our project is about developing a hyperlocal weather prediction model using machine learning. The traditional methods like Inverse Distance Weighting (IDW) are often inaccurate and ignore critical terrain features. Therefore, our model will learn from the data of approximately ten nearby stations using powerful algorithms like Random Forest[1] and Gradient Boosting[2]. This approach allows the model to capture complex, non-linear interactions between weather readings, station distance, and geographic predictors like elevation. To validate our approach, we will benchmark our model's prediction against recorded weather observations, allowing us to quantify its effectiveness in providing accurate hyperlocal insights.

To achieve our objective, we will use three different datasets.

NOAA ISD Data: provides 1 hour granularity of weather conditions at a particular weather station, elevation, and lat/long.
https://www.ncei.noaa.gov/pub/data/noaa/isd-lite/2025/

NASA Shuttle Radar Topography Mission: identifies the topology of the nearby area like valleys, ridges, mountains, etc.
https://portal.opentopography.org/raster?opentopoID=OTSRTM.082015.4326.1

Natural Earth Data - Coastline/Water Body Shapefiles: identifies how far from seas and bodies of water.
https://www.naturalearthdata.com/downloads/

Problem Definition:

Weather predictions help inform the population of daily conditions and approaching dangerous conditions. Weather companies currently use numerical weather predictions (NWP). While these models have been sufficient thus far, their forecasts can take hours, require immense

computational efforts, and have high variability based on similar starting conditions [3]. With the rise of Machine Learning, there is potential for reducing forecasting time and improving accuracy. Our group aims to explore different methods of using machine learning to predict the temperature at specific locations and see how it compares to the actual outcome.

Methods:

Preprocessing:

- We can apply sklearn.decomposition.PCA to the weather station's coordinates, elevation, distance-to-coast, and other features to extract 3-5 principal components, which reduces collinearity and retains spatial variance (helps train smaller models). (Unsupervised)
- We can use torch.nn to learn a non-linear compressed version of the spatial locations, which the PCA might lose. For example, elevation and distance from coast may follow a non linear relationship, so the autoencoder would find those embeddings more effectively for the model. It could also learn new features! (Unsupervised)
- We can use metric_learn.LMNN to learn what makes a station nearby more than just Euclidean distance. For example, places with similar elevation may be "closer" than two places with a smaller euclidean distance between them in terms of weather. (Supervised)
- Normalization - We should apply normalization since most models (especially Neural net based ones) can have exploding gradients with high values. (Unsupervised)
- Feature Engineering: We can look through to find things like is_urban, as urban areas can experience urban heat effect. We can derive things like how elevation changes around the area, which would signal like a valley or ridge, etc, which can affect wind formations.
- We also do Autoencoder to perform an unsupervised feature on static geographic data. The main goal is to create a dense, low-dimensional spatial embedding for each weather station. This embedding is designed to capture the complex, non-linear relationships between multiple geographic features such as elevation, terrain, etc. Therefore, the model will take the input and help us understand the unique geographic context of each station. (Unsupervised)
- We use Pytorch to implement a feed-forward neural network autoencoder (AE). The architecture for the model has 2 main components: encoder and decoder
    - Encoder: A multi-layer perception (MLP) compresses the input features into a low dimensional latent vector. Here is the idea for encoder: Input (D_in) -> Linear(64) -> ReLU -> Linear(32) -> ReLU -> Linear(8)
    - Decoder: The MLP will try to reconstruct the original input features from the latent vector. Here is the idea for decoder: Linear(8) -> ReLU -> Linear(32) -> ReLU -> Linear(64) -> Input (D_in) (For example: the input dimension is 6 for the sf_coast_island region and the latent dimension D_latent is set to 8)
- We witnessed that the model failed to train (get NaN result) until a pre-processing pipeline was established. Here are the steps:

- Step 1: Load data/station_index_<region>.parquet file that contains status features for each station
- Step 2: Select 6 geographic features: lat, lon, elevation_m, dist_coast_m, relief_1km_m, and tpi_1km_m
- Step 3: Check and remove any feature columns with zero variance (where all values are identical). This is necessary as some regions do not have data for these features that would cause the scaler to fail.
- Step 4: To normalize each feature to have a mean of 0 and a standard deviation of 1, we use sklearn.preprocessing.StandardScaler. This is important to stabilize the neural network's training and prevent exploding gradients

Methods:

- We can create a simple, fast, and interpretable non-linear model using xgboost.XGBR regressor, which will basically learn feature importance, its pretty robust due to its symmetric tree splits, and its been used before for a similar problem (Watt-Meyer, et al. 2021). (supervised)
- We can use a sklearn.gaussian_process.GaussianProcessRegressor with kernel (Matern(nu=1.5) × RBF + WhiteKernel) which models spatial autocorrelation and provides uncertainty predictions and uncertainty estimates. (supervised)
- We can use a GNN; a GNN is interesting because through the message sending process, stations communicate through edges which learn spatial propagations (second, third order hopping effects between stations). (supervised)

Results and Discussion:

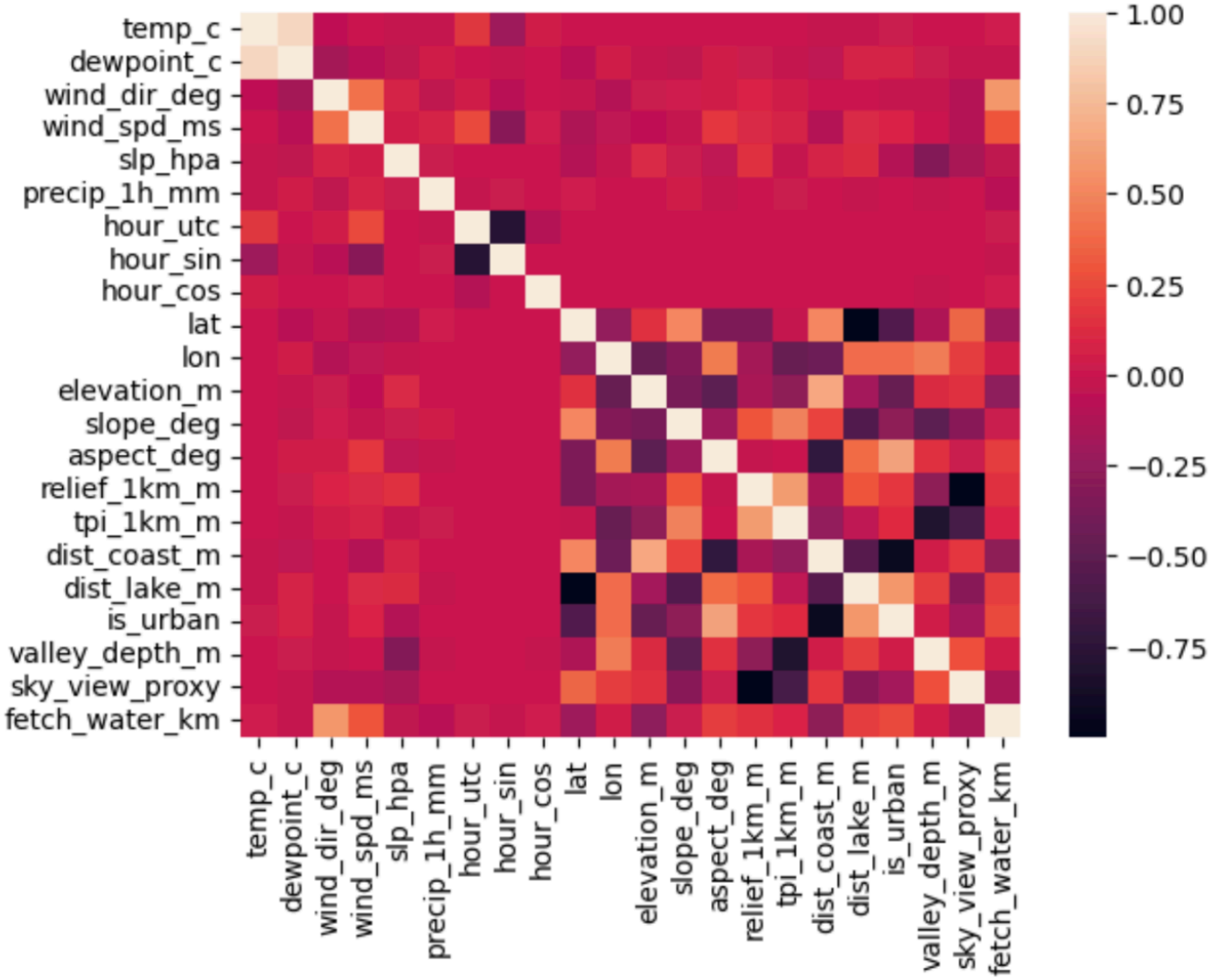
Preprocessing Feature Engineering:

We engineered several station-level features to better capture local influences. The first is is_urban, a simple proxy for urban heat effects based on elevation and distance to coastline. We then used terrain information to compute valley_depth_m and sky_view_proxy. valley_depth_m measures how topographically sheltered a station is, where more surrounding terrain can trap heat and reduce airflow, while sky_view_proxy approximates how open the sky is above the station, with more openness generally allowing for stronger cooling. Finally, for stations near the coastline, we generated an hourly upwind water-fetch feature to capture coastal air influence by assigning a positive fetch value if a station is within 10 km of a coastline and the wind from the ocean sector is blowing towards land.

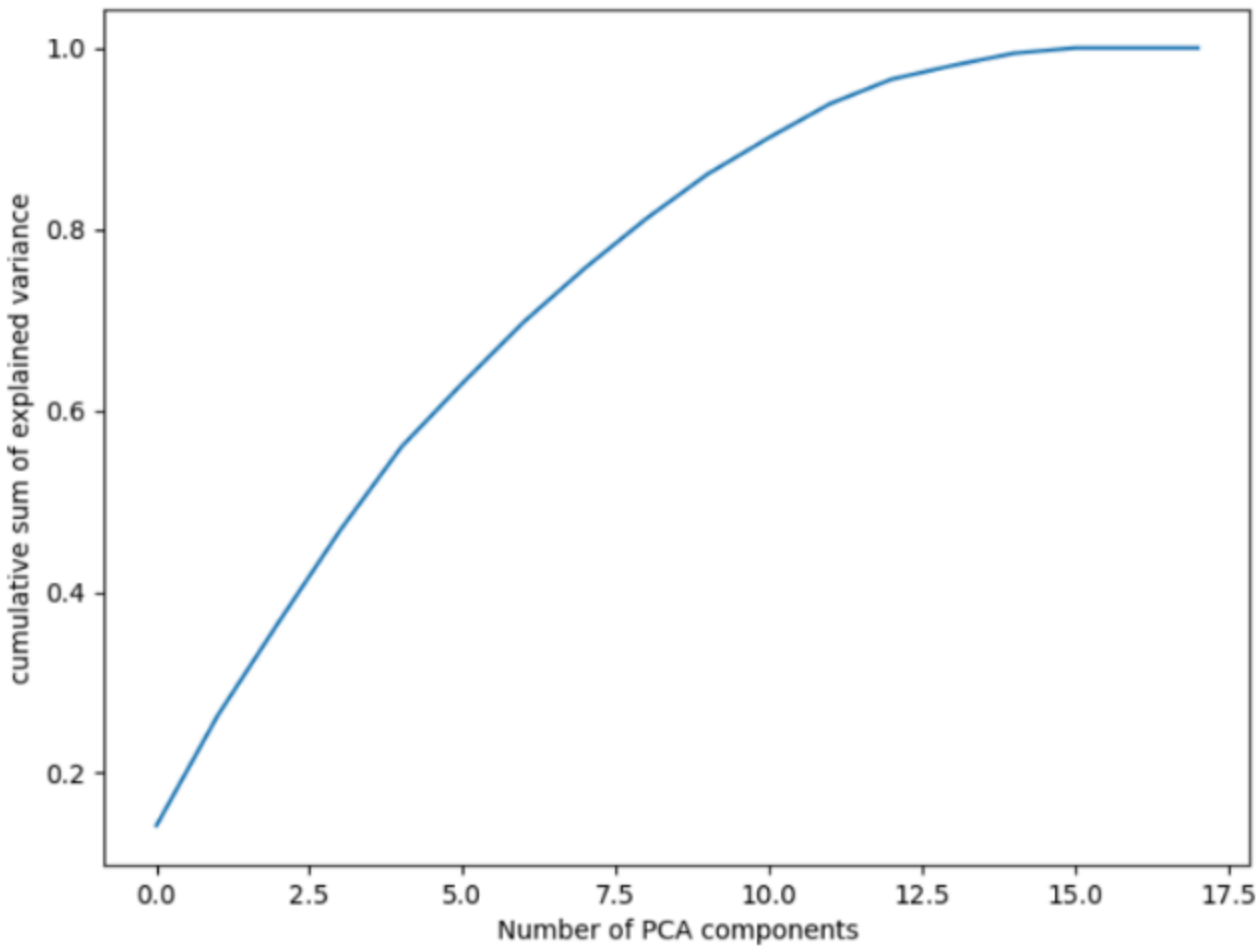
Preprocessing PCA Results:

For PCA we used the original hourly features and the engineered features. First, we examined how many entries were null for each column for each region, then either backfilled or set them to 0.

Since precip_6h_mm usually had 95%+ null rates, we excluded that column in the analysis. Next we explored the data by making correlation matrices to quantify how correlated each pair of features are and took note at which pairs had a correlation value greater than .70 in absolute value.



Once we were ready to do PCA, we used a rolling normalization to normalize the data as we are dealing with time data. If we did a standard normalization, we would be including future points in the mean and standard deviation calculation, which could lead to data leakage. We also split the data into a train and validate set where we fitted PCA on the train set, then transformed both the train and validate set with the resulting PCA. We found that for 10 components we recovered around 80% to 90% of the variance which is nearly a 50% reduction in the number of features as we started with 18 features.

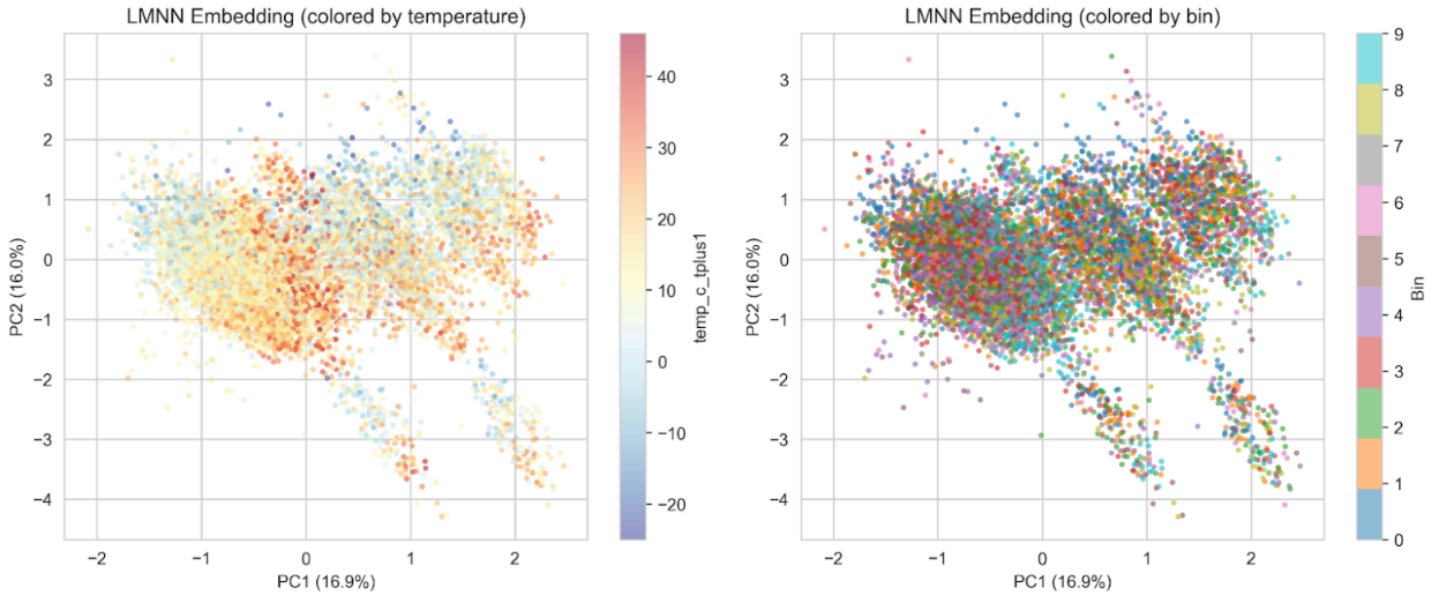


Notice that after 10 components, the trade-off of increasing the number of components to the additional sum of explained variance seemed not worth it, thus, we settled at 10 pca components.

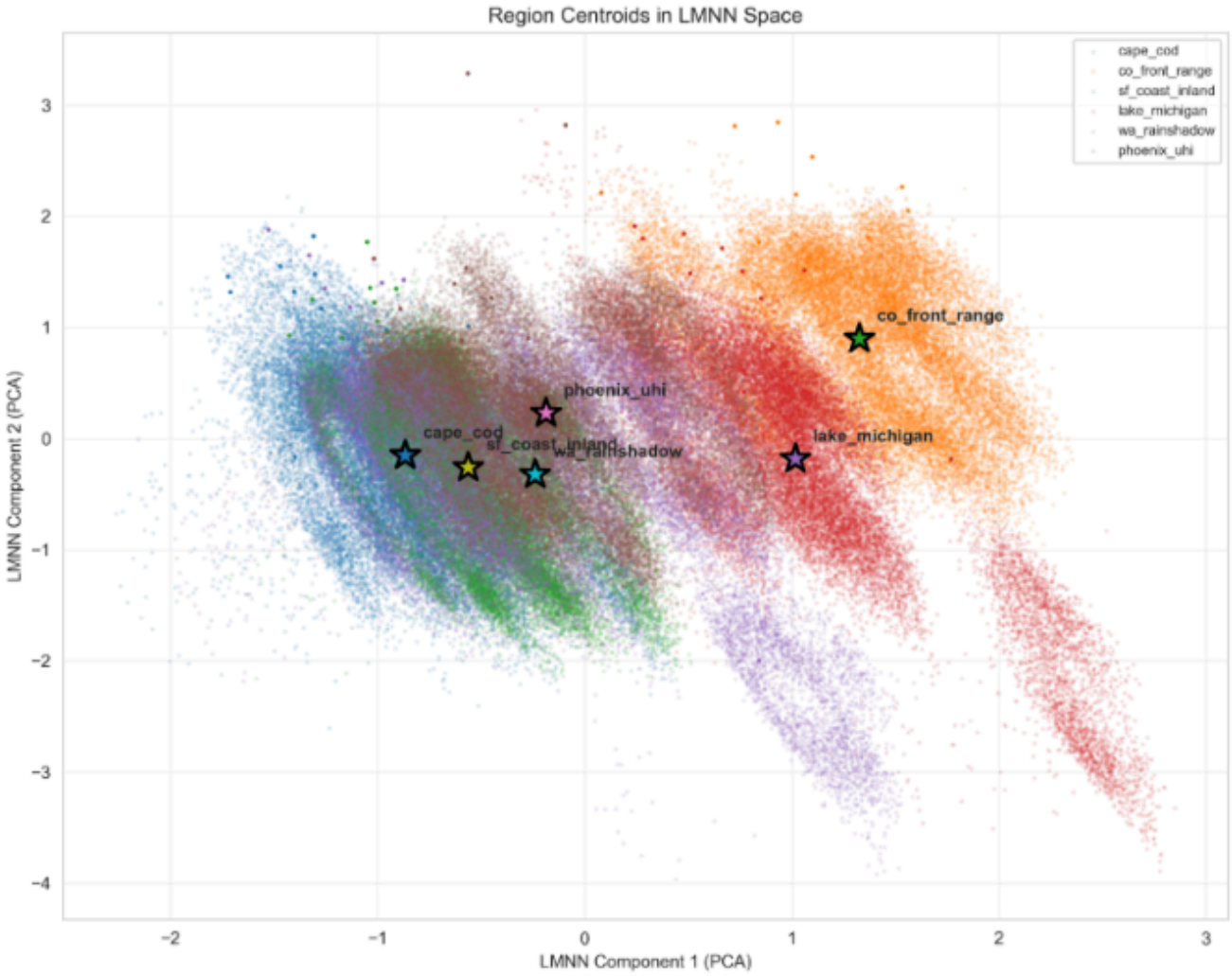
LMNN Results/Interpretations

Large Margin Nearest Neighbor (LMNN) is a supervised metric-learning method that learns a Mahalanobis distance so that points with the same label end up close together while differently labeled points are pushed apart by a margin. It basically produces an embedding where simple k-nearest neighbors work better. In our case, we train it with temperature bins so “nearby” means “similar next-hour temperature behavior.” We will use the resulting embedding as an informed PCA since we supervised it.

We set up an LMNN (large-margin nearest neighbor) on the leak safe rolling normalized dataset with about 700k train rows and 170k val rows. LMNN was told what similar outcomes mean by turning next-hour temperature into 10 quantile bins and using k = 3 “target neighbors,” so it learns a distance where points that lead to similar next-hour temps end up close together (turns it into a supervised problem). We also used a 20,000-row subsample for fitting, and produced a 10-D embedding that we then applied to the full train/val sets.

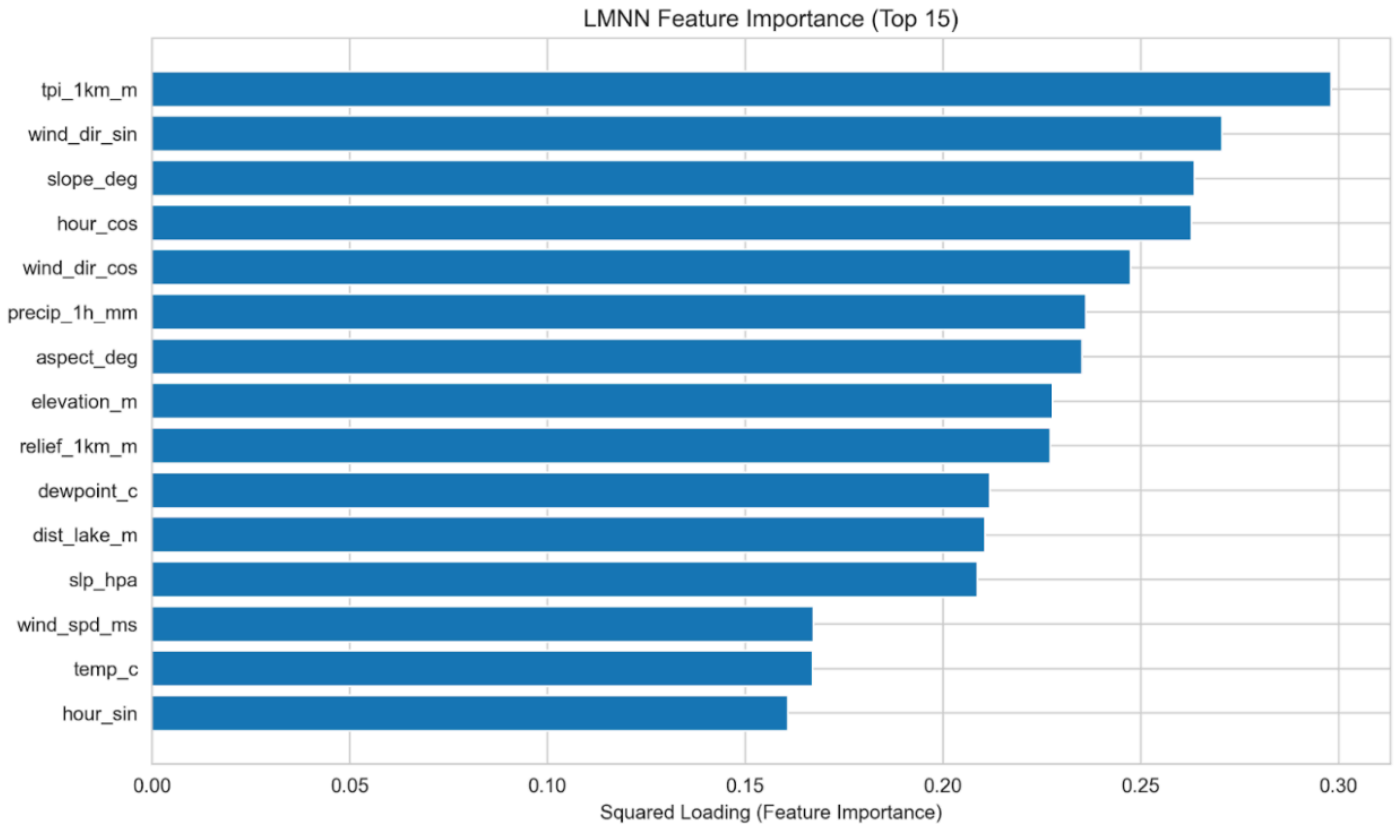


We created 2-D plots by taking the 2 PCA of the learned embedding. In the 2-D LMNN plots temperature colors change smoothly along long “plumes,” which means the space groups stations by similar next-hour temperature behavior, not just map distance. If it were just map distance, this would be quite an unremarkable finding, as obviously there are differences between regions. Each plume is a trail of examples that behave similarly (same station or very similar stations across time), with smooth color changes from cooler to warmer along the plume. That smooth color change tells us LMNN arranged the data so that moving a small step in this space corresponds to a small, predictable change in temperature. Plumes are interesting because they show structure the raw location data doesn’t. Two stations far apart geographically can still land on the same plume if their terrain, winds, and time-of-day make them behave alike. That’s pretty much the behavior we want if we’re going to “borrow” information from the right neighbors.

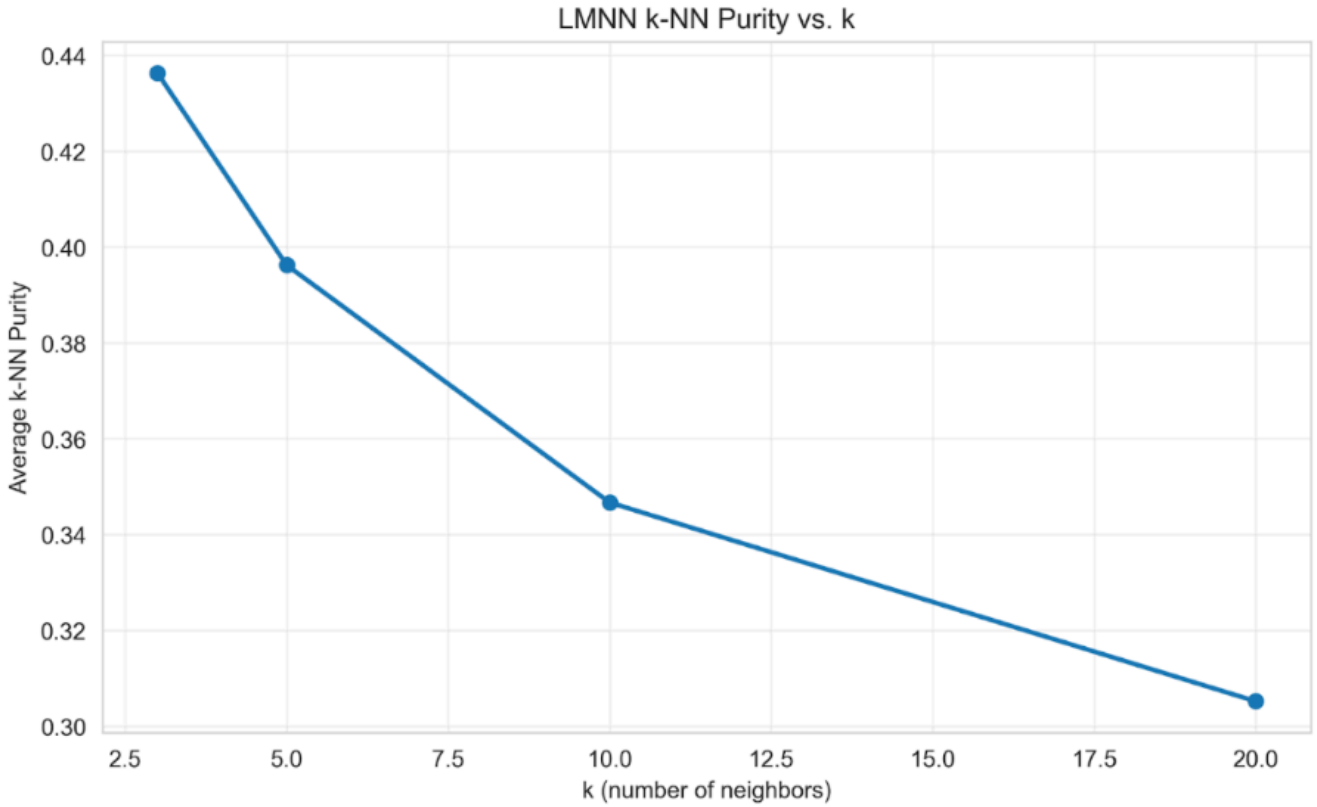

Region Centroids in LMNN Space
cape_cod
co_front_range
sf_coast_inland
lake_michigan
wa_rainshadow
phoenix_uhi
co_front_range
phoenix_uhi
cape_cod
sf_coast_inland
wa_rainshadow
lake_michigan
LMNN Component 2 (PCA)
LMNN Component 1 (PCA)
3
2
1
0
−1
−2
−3
−4
−2
−1
0
1
2
3

Region and station centroids are different but arranged along shared directions, and the cross-region neighbor heatmap (shown a couple pages below) is mostly diagonal, which shows strong within-region structure with some mixing where dynamics are similar such as places with coastal effects

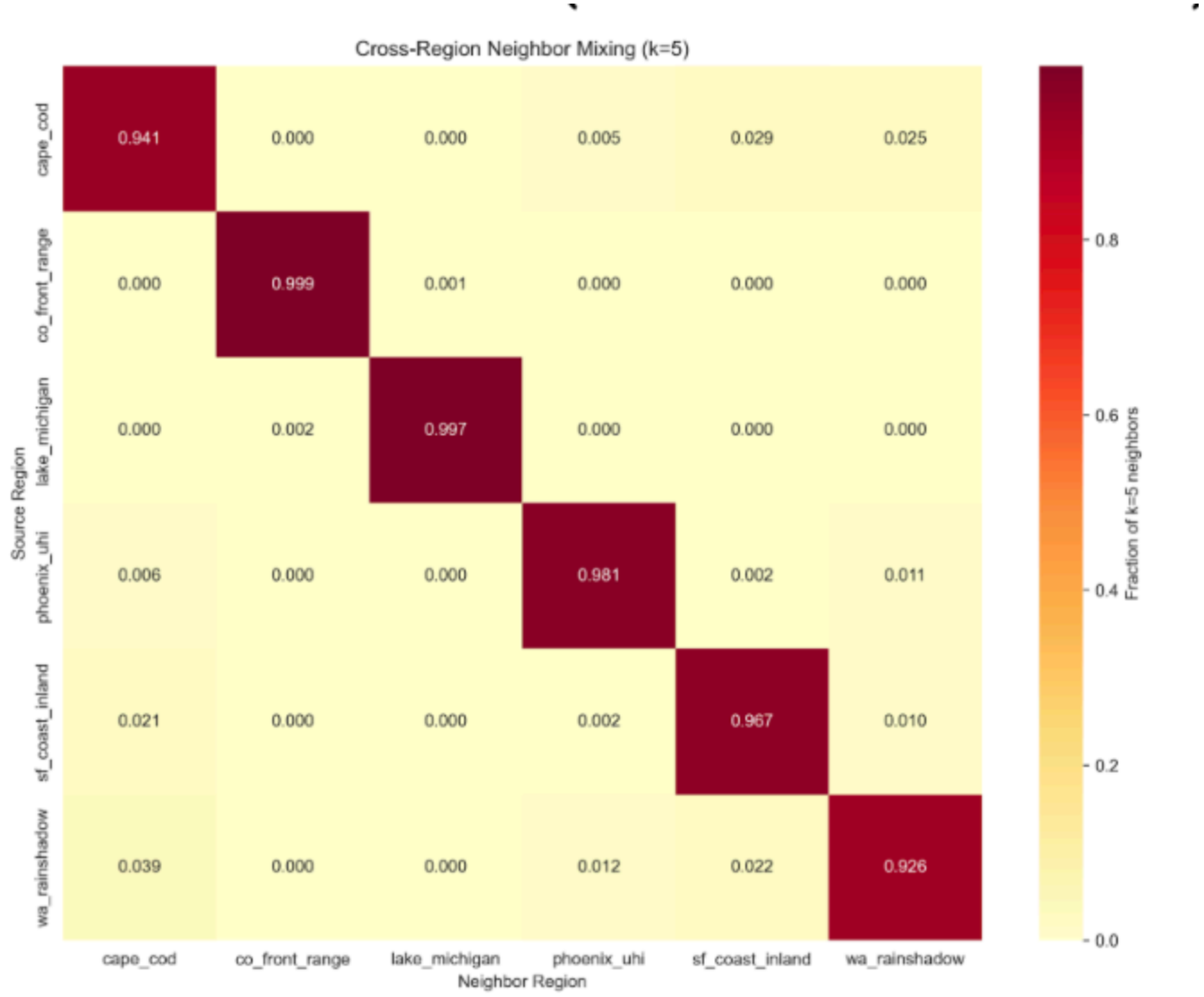
Region centers tend to line up along a shared horizontal direction. This is like a slider that moves from cooler/maritime conditions towards inland ones, or like low to high elevation changes. When two regions' centroids are close , it means the same physical story is being told (both along the coast for example), but when they are far, it means that there is a different driver of temperature changes (mountain vs flat desert).



The feature importance chart backs this up. LMNN leaned on distance to water, terrain/orientation (slope, aspect, relief, elevation), time-of-day, wind direction, and pressure which is the kind of stuff just distance ignores.



We also ran a KNN purity study. A purity of 0.47 at k=3 means about half of the time, the three closest neighbors share the same bin, which is much better than chance. As k grows, purity falls since you are looking further out, which is expected. High purity tells us local neighborhoods are consistent as if you find five nearby points in LMNN space, they most likely represent very similar temperature outcomes, so averaging or learning from them is sensible (like in a local tree model).



The cross-region neighbor heatmap shows most neighbors come from the same region but there's light mixing between regions that share physics like coastal regimes, which is good as this shows LMNN found similarities between regions without squeezing everything together.

This gives us multiple good findings for our models. We can give our models smarter neighborhood signals. We can build the GNN graph using LMNN distances so stations pass messages to similar neighbors. This is a much more informed definition of similarity.

Modeling Method 1: XGBoost

As a baseline method we chose an XGBoost regressor to predict next hour temperature. Gradient boosted trees are non linear methods that try to fit the error of the more general tree before it. It tries to predict the gradients of the old tree (which in MSE and Cross Entropy look like the plain error). This is an additive ensemble method. More specifically, after finding meaningful local regions, the tree predicts the mean of that region as a regressor. The XGBoost is a great baseline as it is quick, cheap to do on a CPU, robust to weak preprocessing, and is industry standard as it

generalizes well over a deeper model that is prone to overfitting. Also, we don't have to feed it long temporal sequences, so the model is compute bound over memory bound, which is a sign of efficiency. As a prior, we thought as a group that over something like a linear regression, a non-linear model could learn the complex interactions between things like wind speed, elevation, and current temperature.

We fit the model on just the raw, rolling window normalized data, the PCA data, and autoencoded data, and the combination of all three types. We trained the XGBoost for regression (reg:squarederror) with 500 histogram-split trees (tree_method='hist'), max_depth=8, a conservative learning_rate=0.05, and regularization via subsample=0.8, colsample_bytree=0.8, min_child_weight=3, gamma=0.1, reg_alpha=0.1, reg_lambda=1.0, using random_state=42 and all CPU cores.

These are the results.

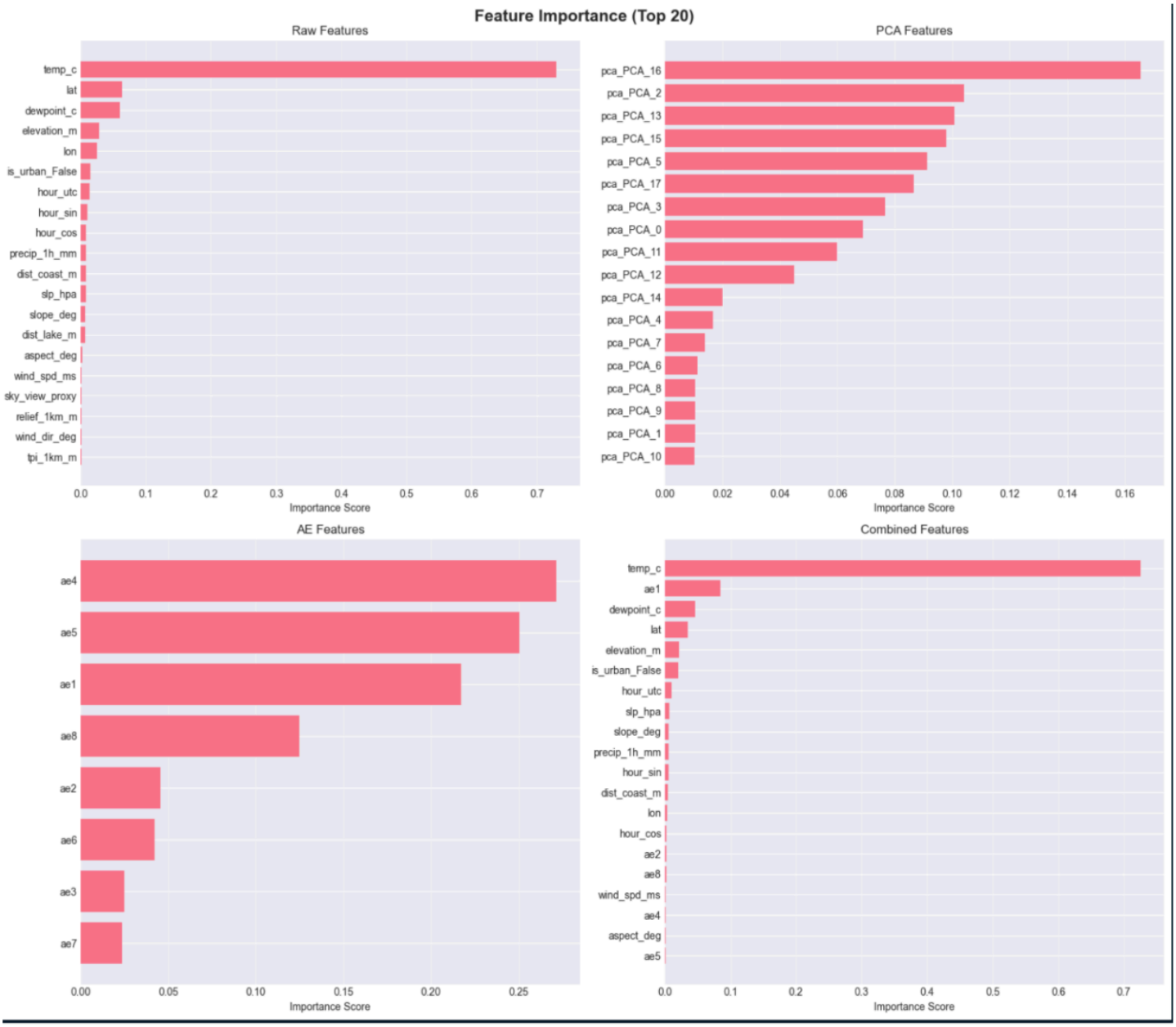
```
================================================================================
PERFORMANCE COMPARISON
================================================================================
                     Model      RMSE      MAE      Bias        R²
     Baseline (Persistence)  1.550000 1.150000       NaN       NaN
     XGBoost (Raw Features)  1.139348 0.780318 -0.071406  0.985448
     XGBoost (PCA Features)  9.727294 7.745762 -1.003327 -0.060699
      XGBoost (AE Features) 10.352133 8.162478 -1.249218 -0.201345
XGBoost (Combined Features)  1.161697 0.797170 -0.078219  0.984872
================================================================================

Results saved to model/performance_metrics.csv
```

Clearly, there is high auto correlation between temperatures (temperatures don't change too much over the hour), so even without neighboring station data or sequential time data, the baseline XGBoost does well, getting an R^2 of around 98.5%. This was as expected, since temperature dynamics are slow. Also the mean is shifted due to seasonality, so R^2 is trivially boosted. The RMSE tells us that on average, our best model is off by 1.14 degrees Celcius, which is pretty good (think about the difference between 25 and 26 degrees Celcius), vs the AE is off by 10 degrees (we will get into why).

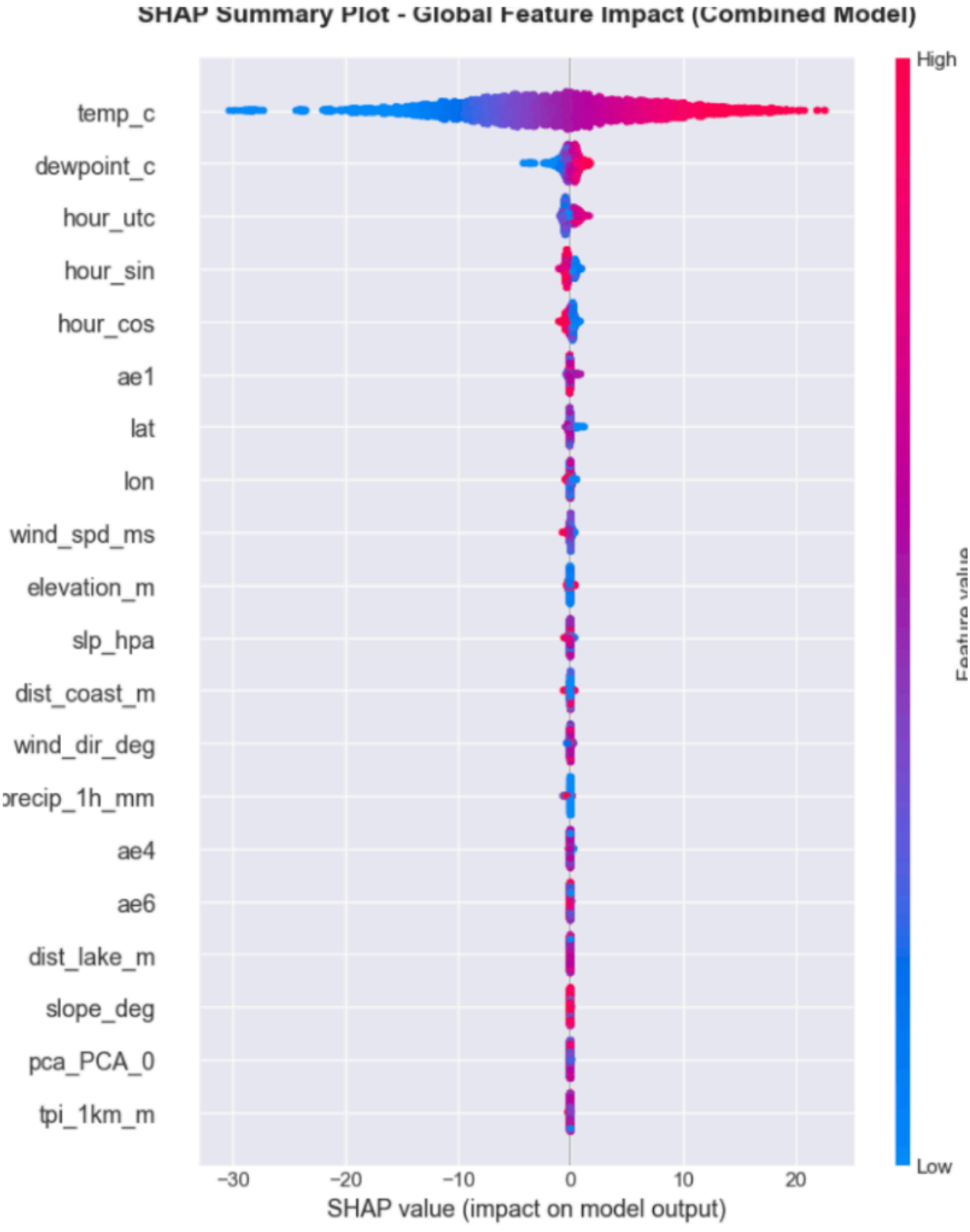
Two interesting findings are that the XGBoost trained on the raw data greatly outperforms the other models. This is quite confusing as our PCA captured about 90% of our data variation and the AE should be similar in capturing variation.

This makes a lot more sense when you look at the importance graphs.

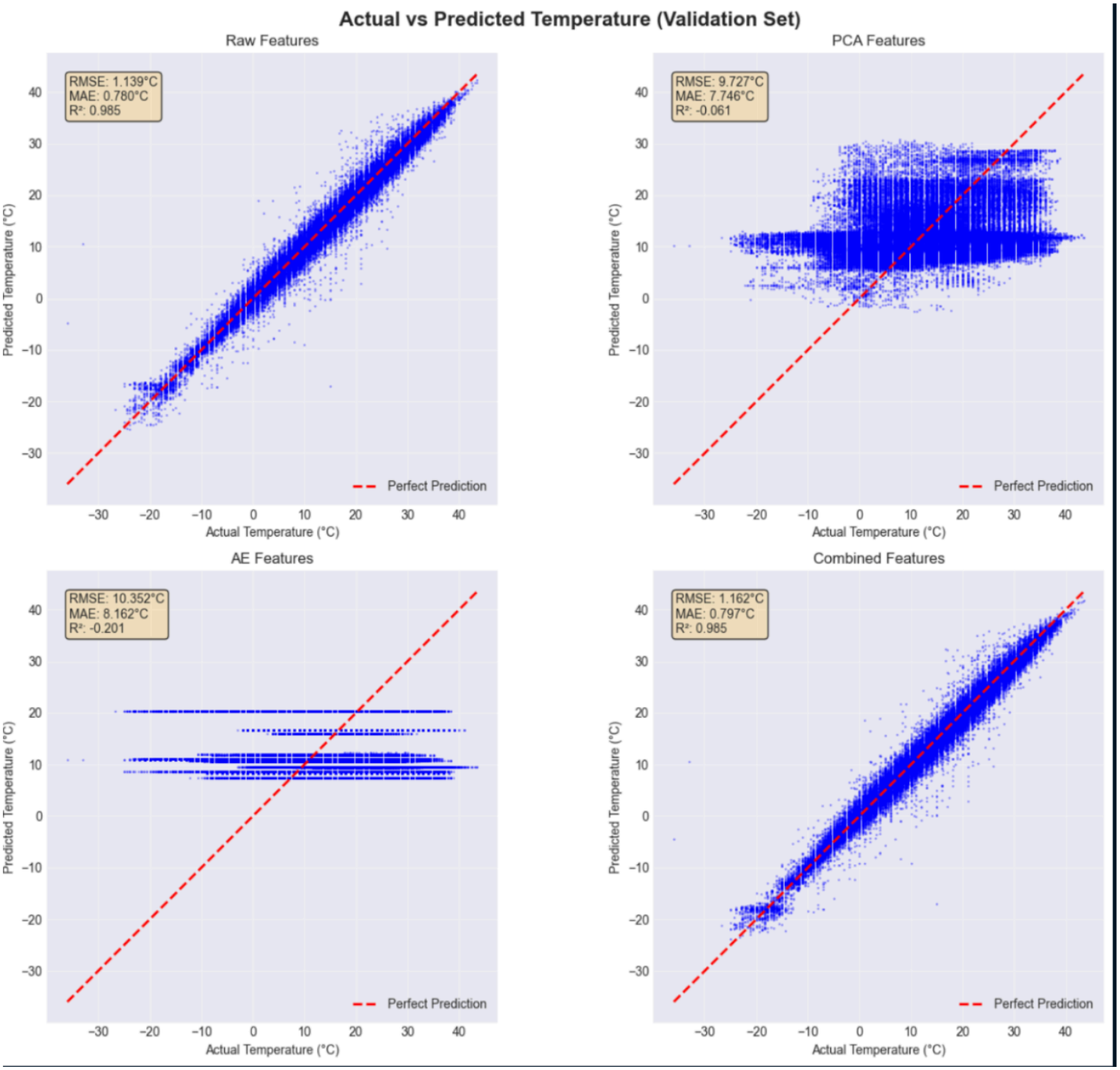


Clearly, our most important signal is the current temperature. However, when we run PCA or AE, we treat all features as equally important to retain, so we end up fogging our super important current temperature feature. The negative R^2 value tells us that predicting the mean temperature would be better. If we included a seasonality term, that would definitely help as well (sin transformation of month). It's also quite interesting to see that ae1 is the second most important feature in the combined model, which shows the non linear model is learning useful higher dimensional features that the PCA isn't able to. This term could be a rate of change of temperature based on elevation, or something of that nature. The fix to this PCA/Autoencoder problem is to weight the loss function of the autoencoder/PCA to give higher weight to the error of the current temperature term, or to append the current temperature to the representation so that it is perfectly preserved.

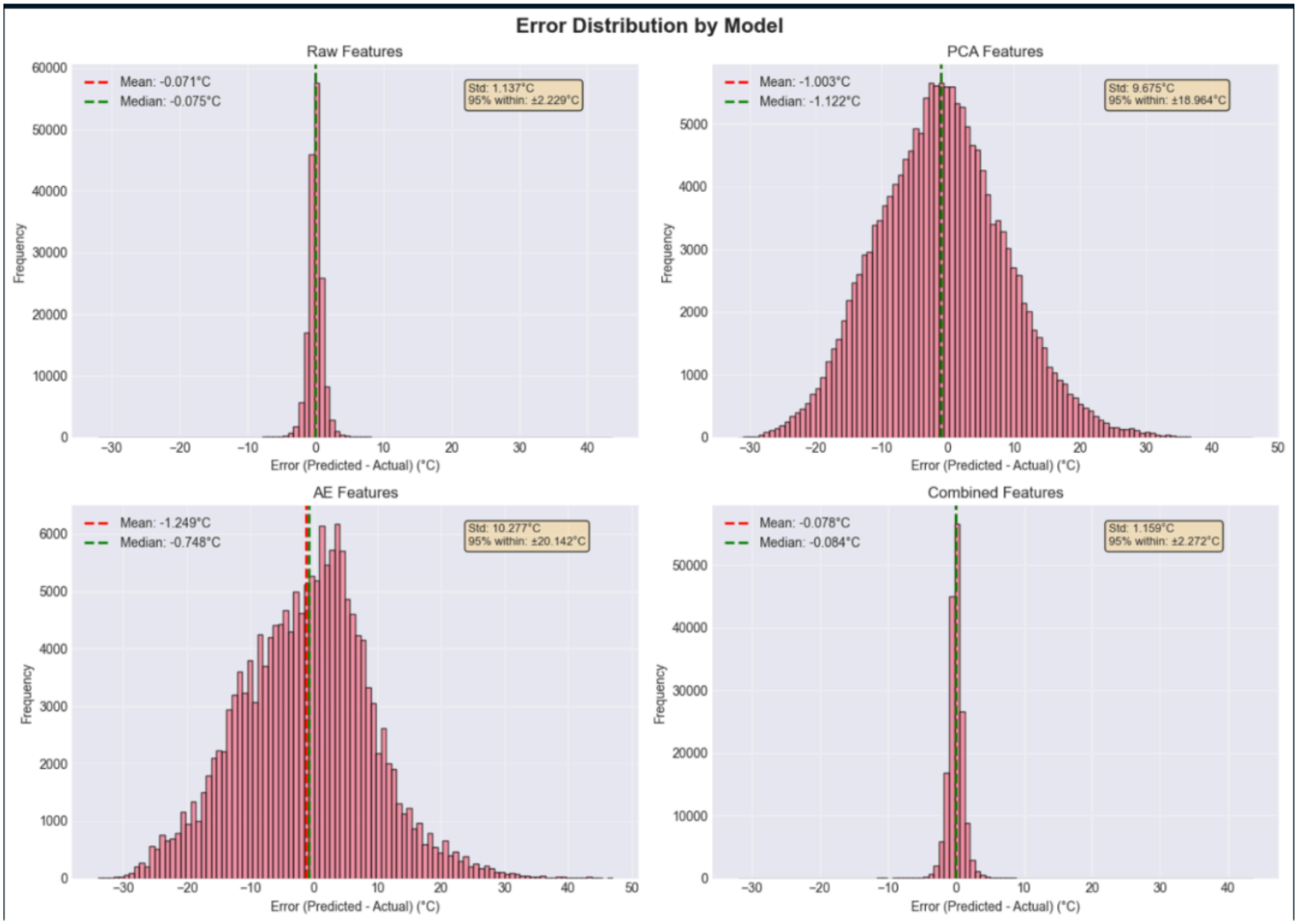
A SHAP analysis reveals a similar story.


SHAP Summary Plot - Global Feature Impact (Combined Model)
temp_c
dewpoint_c
hour_utc
hour_sin
hour_cos
ae1
lat
lon
wind_spd_ms
elevation_m
slp_hpa
dist_coast_m
wind_dir_deg
precip_1h_mm
ae4
ae6
dist_lake_m
slope_deg
pca_PCA_0
tpi_1km_m
−30
−20
−10
0
10
20
SHAP value (impact on model output)
High
Low
Feature value

An interesting finding is that time of day tells us a lot, which makes sense as night/day shifts differ from place to place (deserts get much colder at night and much hotter in the day).



This graph of predicted vs actual shows what's happening with the PCA/AE features more in depth. The learned representations are super compact in its vector space. With the AE, you can see their predicted values are in horizontal lines, and a noisier version is seen in the PCA. This tells us most of the variation captured is uninformative for the dependent variable, while the most important feature, current temperature, is thrown away.



If we look at the errors, we see that they look like Gaussian noise. This tells us our XGBoost model isn't underfitting to the data. Rather, the PCA/Autoencoder data is just bad and loses its signal via the transformation.

The next steps are pretty clear. We must maintain the most important feature (curr_temperature) and we can use some of the learned features via PCA/Autoencoders as engineered features to capture useful non-linearities (this can be implemented via an append). Also, the raw XGBoost model does very well, achieving an R^2 value of 0.985. The tight, mean 0 errors tell me there is little accuracy to be gained from a single step prediction model. A GAN or model that looks at its neighboring stations should be able to capture R^2 gains, as it will learn cross sectional relationships. I think we can get into the 0.995 range with that!

Modeling Method 2: Gaussian Process Regressor

For our next modeling method, we used a Gaussian Process regressor (GPR). While XGBoost learns a deterministic mapping from inputs to outputs, GPR instead treats the prediction as a distribution over functions, giving us both a prediction and its uncertainty. This is extremely valuable in weather

forecasting, where knowing how confident the model is can matter just as much as the predicted value itself.

The core idea behind GPR is that nearby data points in feature space should behave similarly, and this reflects a strong environmental prior that weather is smooth and spatially correlated. Therefore, instead of learning fixed weights like in linear regression, GPR learns a kernel function that measures similarity between two stations. The closer two stations are in spatial or feature space, the stronger their influence on each other's predictions.

We use the composite kernel:

Kernel = Matern(nu=1.5) × RBF + WhiteKernel

where the Matern × RBF term models smooth geographic variation in temperature across lat/lon, elevation, relief, and coastal distance, the WhiteKernel accounts for station-level noise and random effects, and taking the product allows the model to find structured local regions and, similar to an XGBoost tree, estimate the mean function of the data inside those regions.

Training Setup

After assembling the combined dataset (raw features + engineered features + autoencoder embeddings + PCA embeddings), we filtered the data to include only numeric, non-missing columns. This produced:

- 15,072 clean training rows
- 3,786 clean validation rows

Because exact GPR scales on the order of n^3 in the number of training samples, we trained the model on a random subset of 5,000 representative training points while still evaluating on the entire clean validation set.

The optimized kernel learned during training was:

18.5^2 × RBF(length scale = 135) + WhiteKernel(noise = 0.00901)

This kernel reflects extremely smooth structure in the temperature field and a small amount of station-level noise. The learned parameters are consistent with weather behaving as a smooth physical process.

Results

The Gaussian Process model performed strongly:

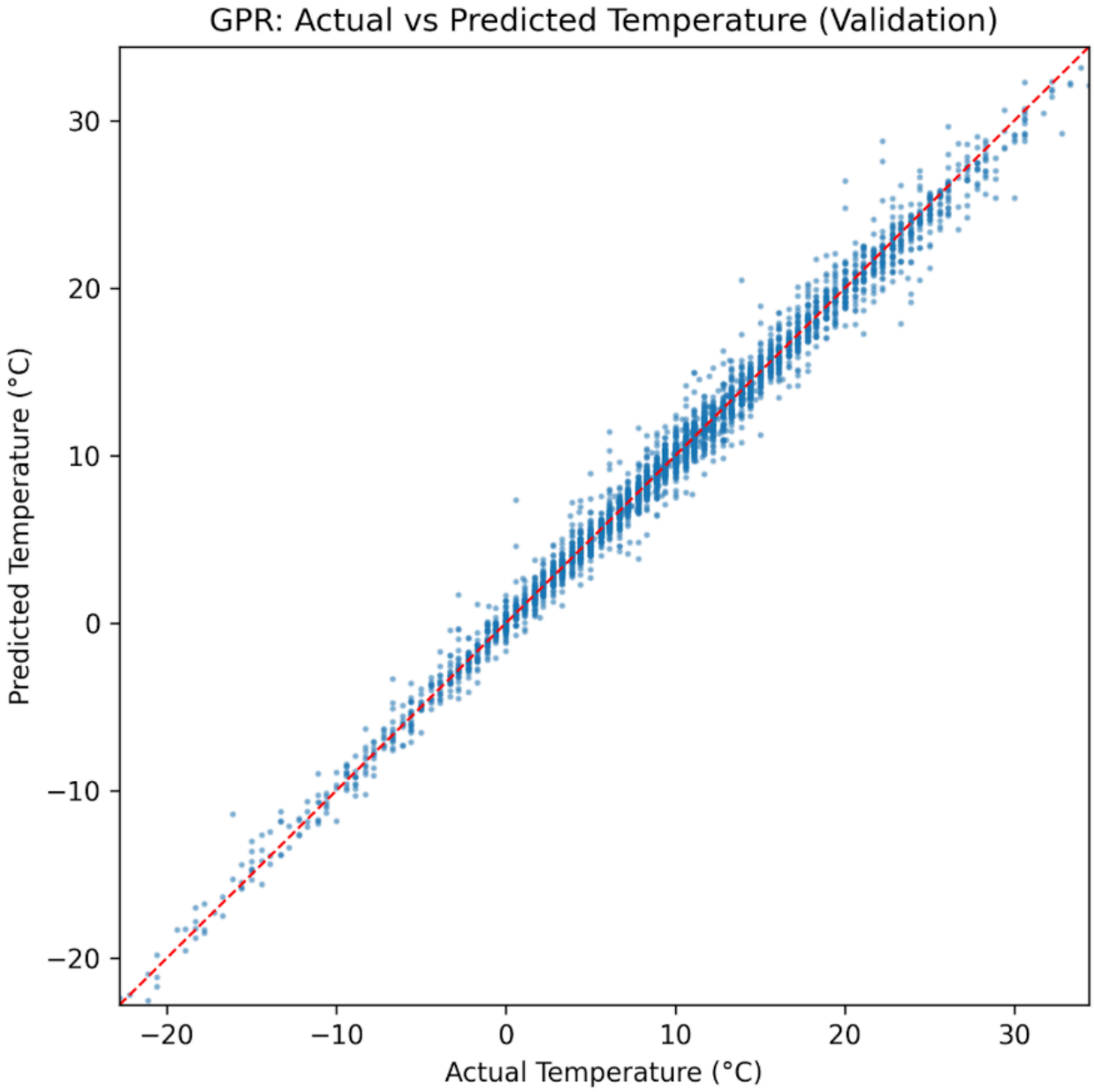
- MSE: 0.8096
- RMSE: 0.8998 °C

- R^2: 0.9885
- Mean predicted uncertainty (std dev): 0.9268 °C

The fact that the model's predicted uncertainty (~0.93 °C) nearly matches the actual RMSE (~0.90 °C) indicates that the GP is well calibrated, since it's both accurate and honest about how confident it is.

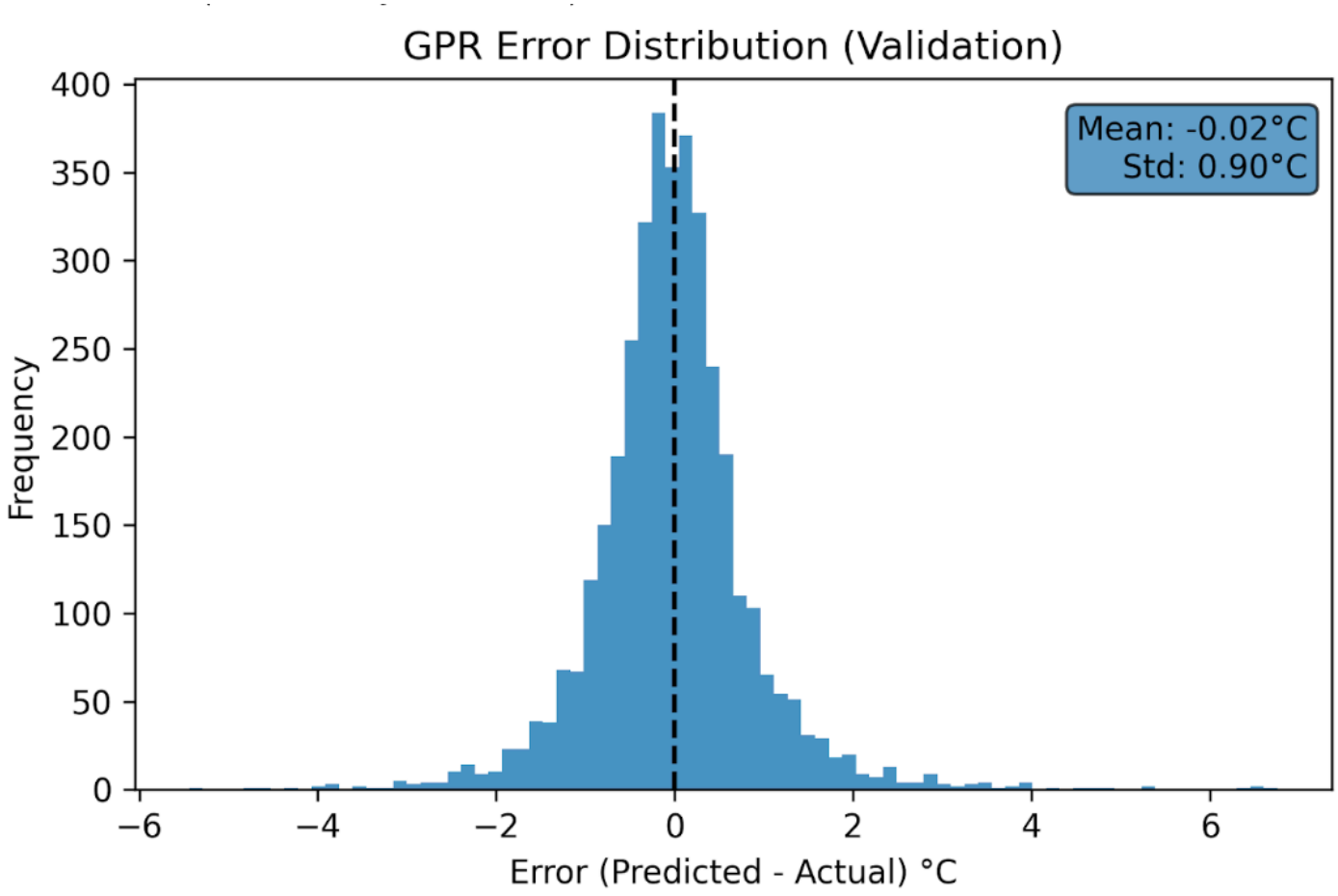
Actual vs Predicted Temperatures

The model's predictions lie tightly along the 1:1 line, with only slight widening at the extremes. This shows that GPR captures both low and high temperature ranges accurately.



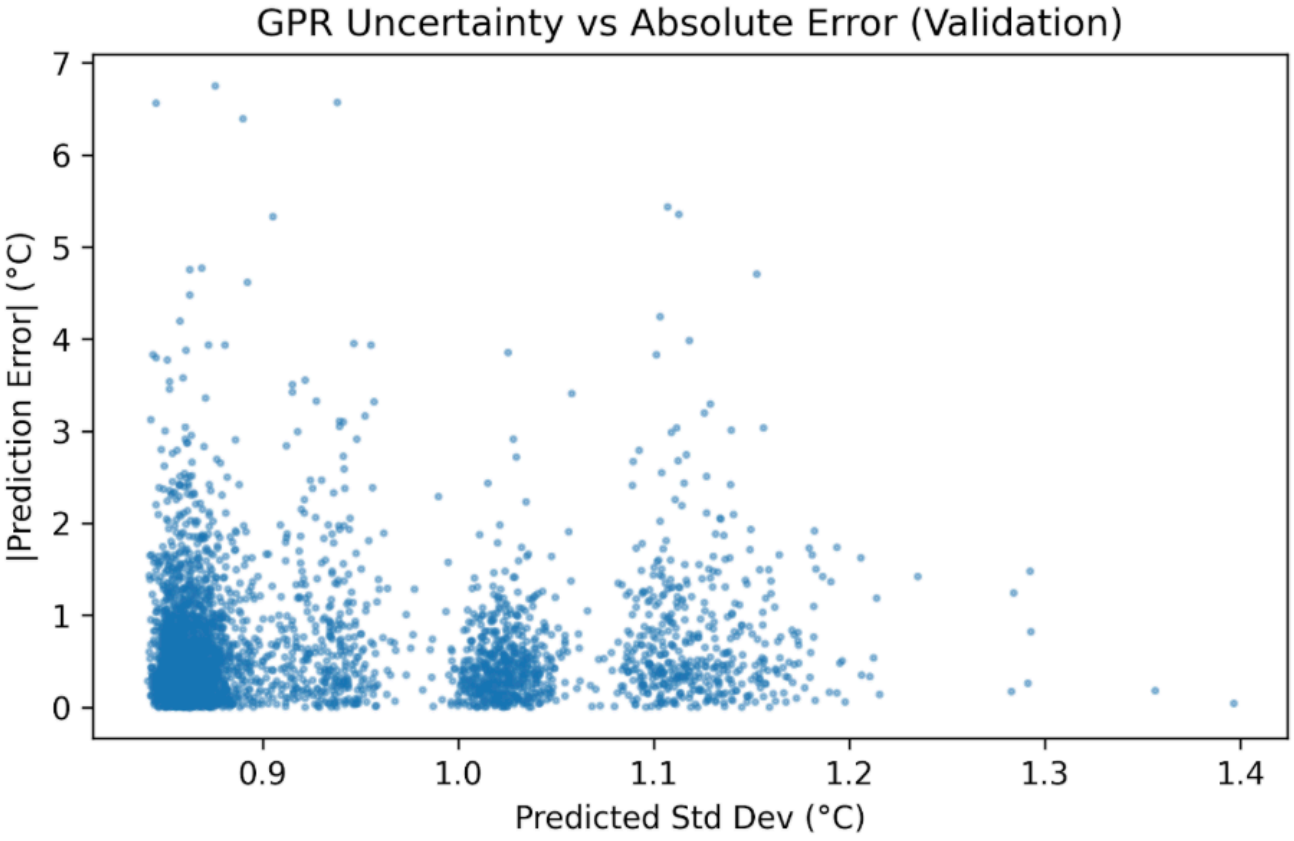
Prediction Error Distribution

Errors are centered around zero, symmetric, and approximately Gaussian. The mean error is –0.02 °C (essentially unbiased), and the standard deviation is 0.90 °C.



Uncertainty vs Absolute Error

Predicted uncertainty correlates positively with actual error, which is expected for a well calibrated Bayesian model. When the GP reports higher uncertainty, the predictions are less accurate.



Analysis & Comparison

Although GPR performs nearly as well as XGBoost numerically, achieving nearly 99% explanatory power with sub-degree error. Additionally, unlike deterministic models like the XGBoost, GPR provides very solid uncertainty estimates and understands when and where temperature is harder to predict, which is especially valuable in weather forecasting. However, GPR still assumes smooth spatial behavior and does not explicitly model interactions between nearby stations. This is visible in regions with rapidly changing weather dynamics, where uncertainty increases. This observation motivates the next stage of our next modeling method, a Graph Neural Network, which directly passes information between stations and attempts to learn spatial propagation rather than just spatial smoothness.

Modeling Method 3: Graph Neural Network

For our final method, we use a Graph Neural Network (GNN). Unlike XGBoost and GPR, which treat each station as an independent sample, the GNN treats stations as nodes in a spatial graph and allows them to communicate through edges, mimicking how weather actually propagates across geography.

For implementation, we constructed separate GNNs for each region because there is no physical reason to believe that stations in different regions will influence each other. Nodes represent the stations and edges are formed using a K-nearest neighbors approach over geographic features.

To predict next-hour temperature, the GNN performs message passing, where each station aggregates information from its neighbors over the graph. This allows the model to learn how temperature evolves across space, rather than treating each station as isolated. Unlike GPR, which assumes smooth spatial structure, the GNN can learn asymmetric propagation patterns driven by elevation, wind direction, and terrain. In this way, it acts as a physically informed model that attempts to capture processes such as coastal inflows, valley heat trapping, and temperature fronts moving inland. By encoding spatial relationships directly into the architecture, the GNN moves beyond smooth interpolation and begins modeling spatial dynamics, which is closer to how weather actually behaves. We then pass these states per station into an LSTM which learns how to evolve hidden states throughout time! Since temperature is highly autoregressive, our priors lead us to believe this would be an effective modeling solution. We then added a linear layer with a non linearity as part of a prediction head.

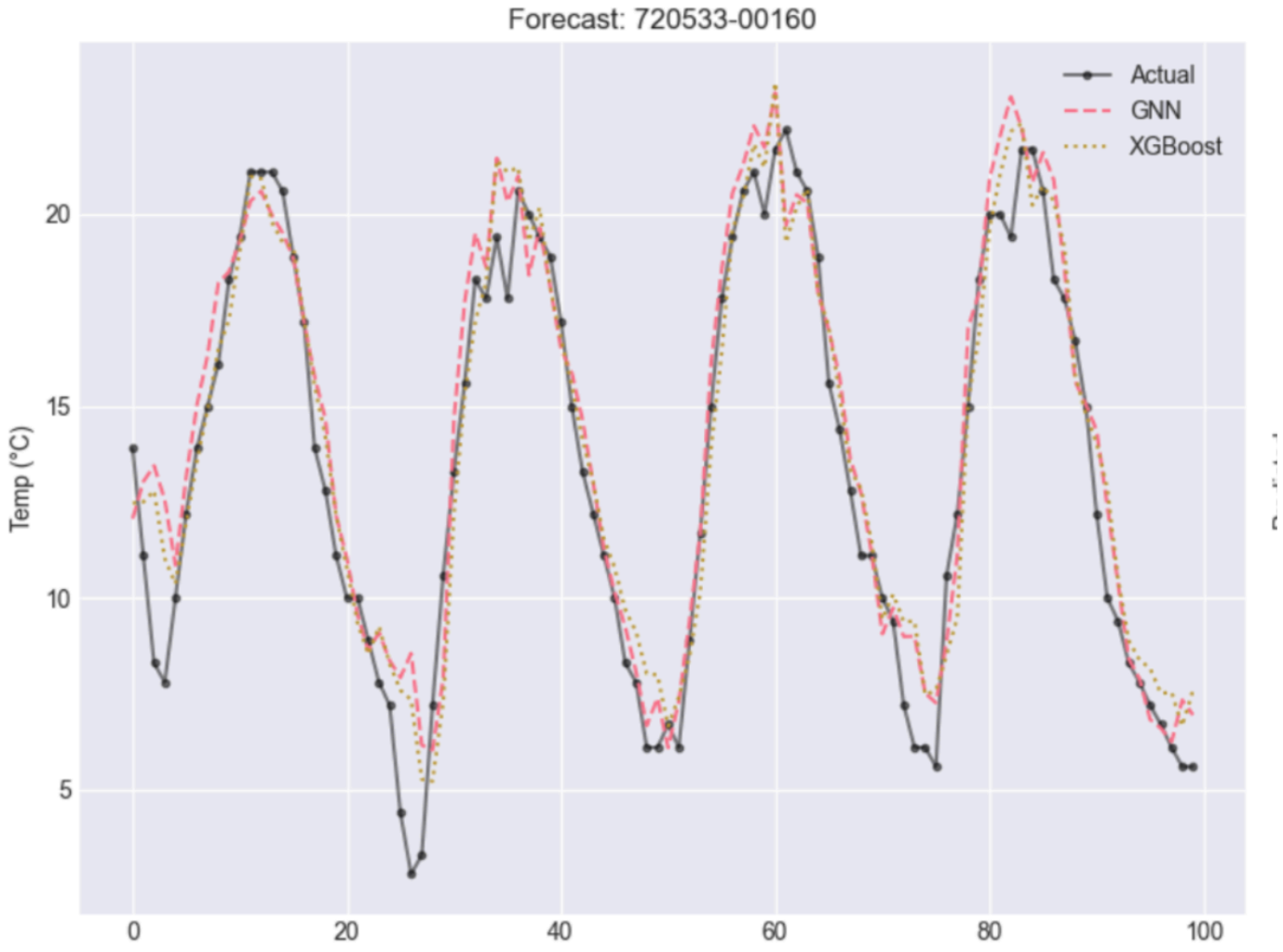
Training Setup + Implementation

We constructed fully connected graphs within each geographic region to allow for instantaneous information flow across local clusters, rather than relying on sparse nearest-neighbor connections. These spatial states were processed using a GConvLSTM architecture, which combines graph convolutions for spatial message passing with Long Short-Term Memory (LSTM) units to model the highly autoregressive temporal evolution of temperature.

The model was trained to predict the next hour's temperature using a 1-hour lookback window, utilizing a hidden dimension of 64 and a batch size of 256 to balance model capacity with gradient stability. We optimized the network using the Adam optimizer with a learning rate of 0.002 over 10 epochs, using Mean Squared Error (MSE) as the loss function. The 1 hour look back is because the dynamics are highly autoregressive and so most of the information should be encoded in the last hour. For tailed events, a longer window would help, but this would greatly increase computational complexity for transferring these long sequences into our GPU and the LSTM would start running into exploding/vanishing gradient issues. A hidden dimension of 64 gives us a rich dimension to learn complex relationships between variables. The input vector is already very dense, so a bottleneck would not do very much for us. The learning rates and epochs were chosen empirically and using widely used deep learning heuristics. Since MSE correlates well with RMSE and MAE, we chose this as our objective function (and its more easily optimized). Note we added day/month sinusoidal time embeddings to inject seasonality information.

Results We also evaluated a persistence baseline, which assumes the next hour's temperature equals the current hour's. R^2 tells us how much better we are at predicting a value than a random prediction (the mean). A more calibrated, low effort way to calculate this is to predict via persistence, which is the idea that the next hour the temperature will stay the same. We therefore add this as another baseline.

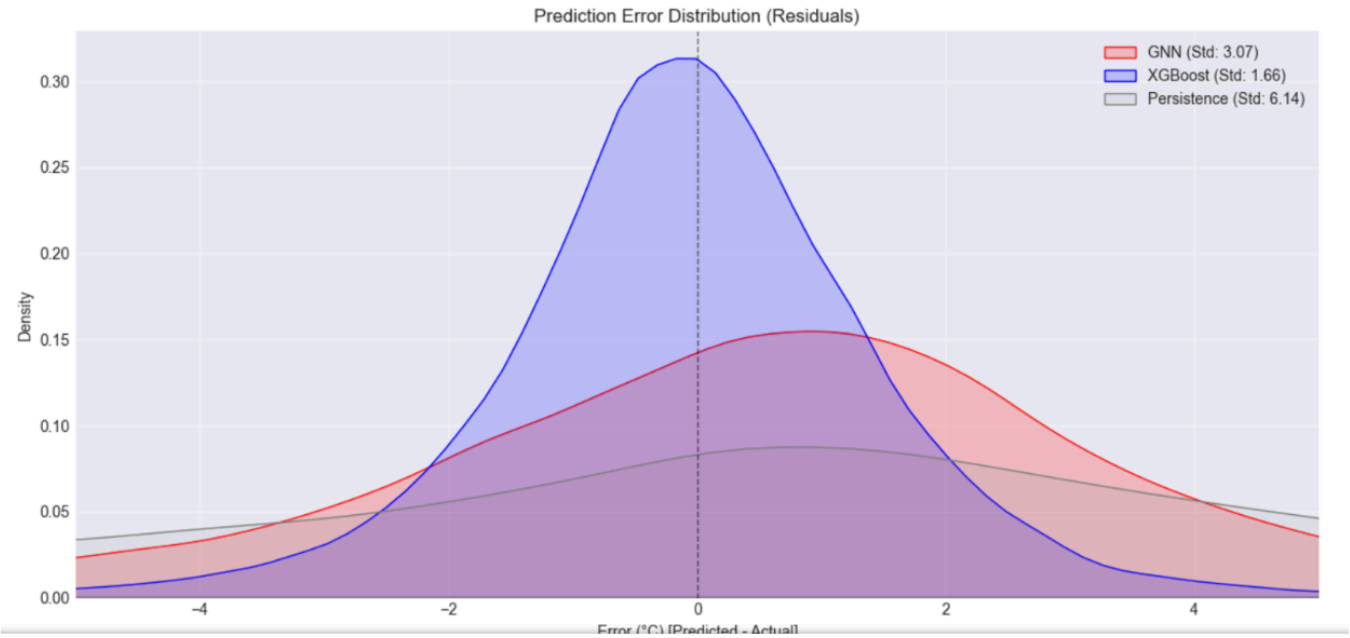
	GNN	XGBoost	Persistence
Station Avg R^2	0.952	0.985	0.70
Station Avg RMSE	1.3266	1.1314	1.97849
Skill vs Persist	29.24%	38.59%	



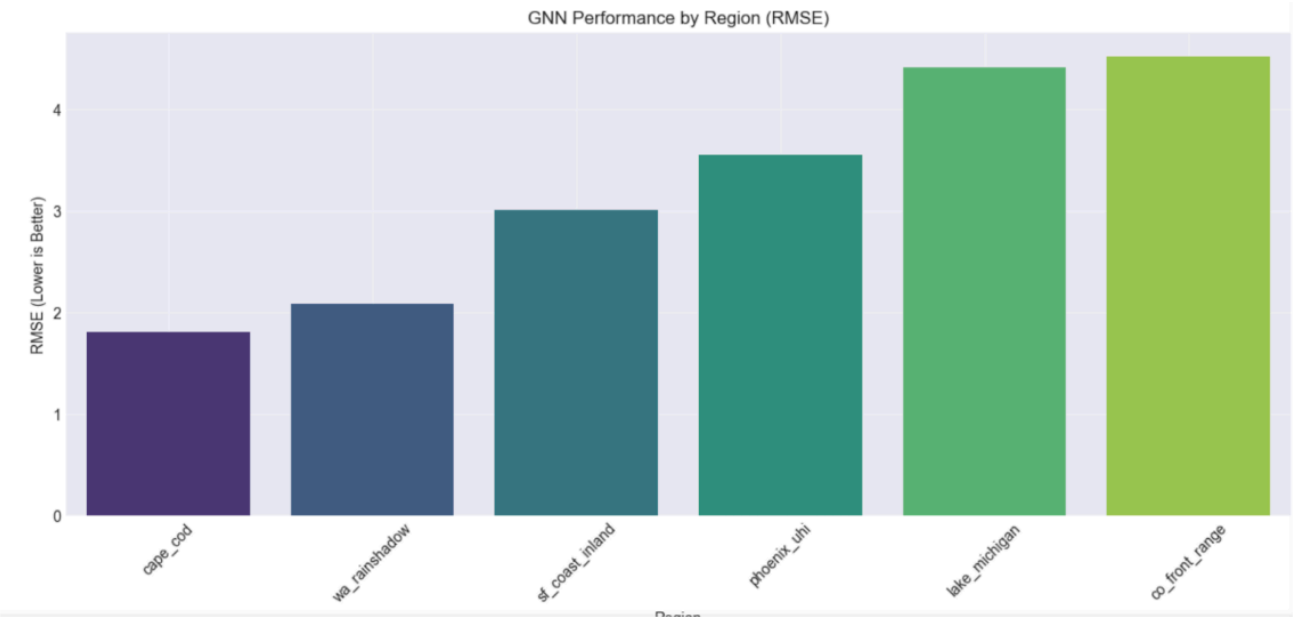
While this persistence baseline achieved a respectable Station Average R^2 of 0.70 (RMSE: 1.98), both machine learning approaches significantly outperformed it. The XGBoost baseline demonstrated the highest accuracy in this experiment, achieving a Station Average R^2 of 0.985 and an RMSE of 1.15, representing a 38.59% skill improvement over persistence. The GNN also showed strong predictive power with an R^2 of 0.972 and RMSE of 1.33 (29.24% skill improvement), confirming that it successfully learned underlying spatial dynamics, even if the gradient-boosted

trees achieved slightly higher precision in this specific configuration. We omitted using LMNN metrics because the flexible nature of deep learning models allows us to learn this!

Error Distribution

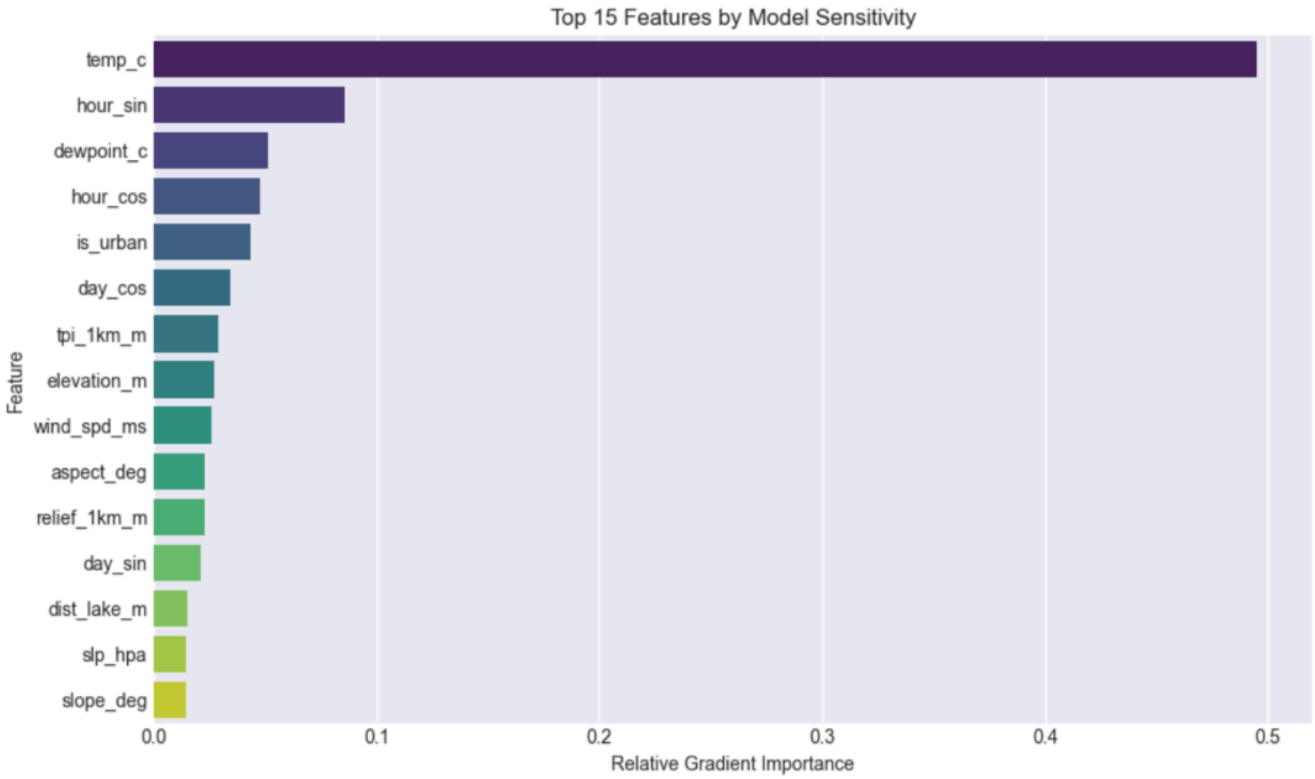


The error residuals, aggregated across all stations, exhibit Gaussian tendencies, consistent with the Central Limit Theorem. The XGBoost model demonstrates superior performance with residuals tightly clustered around zero and thinner tails compared to the GNN. However, a negative bias (underprediction) is observed in the XGBoost model, whereas the persistence baseline and deep learning model consistently overpredicts. This suggests the models may be struggling to capture local station intricacies or seasonal signals, resulting in conservative predictions. The XGBoost model's thinner tails suggest its ensemble of trees effectively "memorized" extreme tail-end conditions, whereas the GNN produced smoother residuals, indicating weaker sensitivity to extreme temperature events.



Regionally, the GNN demonstrates strong predictive capability in stable coastal areas like Cape Cod and Washington. However, performance degrades significantly in high-variance regions such as Colorado and Lake Michigan. This discrepancy underscores the difficulty of applying global weights to diverse climates. The model struggles to adapt to the high volatility of inland and mountain regions, highlighting the potential need for edge-specific learned weights to better model local weather dynamics.

Analysis & Comparison



Feature analysis reveals that the GNN relies less heavily on the immediate previous temperature compared to XGBoost. This indicates that the deep learning model is successfully capturing temporal dynamics, such as time-of-day and seasonal trends, rather than simply piggybacking on the last known value. Despite this, the GNN currently underperforms the XGBoost baseline. This performance gap is likely driven by data limitations; deep learning models are notoriously sensitive to missing data, and the imputation ("filling") required for the frequent NaN values in this dataset likely hampered the GNN's ability to learn robust features. In contrast, tree-based models like XGBoost handle sparsity and missing values more robustly. Furthermore, the GNN trails the Gaussian Process Regression (GPR) baseline by approximately 4% in R^2. The GPR likely outperforms here because weather data is inherently smooth and spatially correlated, which are properties that are explicitly encoded in GPR kernels. GPRs can model these spatial covariances effectively even with smaller or sparser datasets, whereas a GNN requires significantly more data to "learn" these same physical relationships from scratch.

Next Steps

To bridge the performance gap, future iterations should revisit the weight-sharing mechanism. Currently, the GNN applies shared kernels across the graph. To achieve hyper-local accuracy, we should introduce edge-specific weights or attention mechanisms. This would allow the network to learn distinct spatial relationships for specific station pairs (e.g., how Lake Michigan affects Chicago vs. how the Atlantic affects Cape Cod) rather than applying a "one-size-fits-all" spatial rule. Temporal spatial models are a natural next step for this use case. Since temperature is highly autoregressive,

we could use state of the art temporal models like selective state models (SSMs). The S6 paper, Mamba is very popular as it doesn't run into the compute limits of attention based models, so it can operate on long sequences. Their inductive biases are also quite helpful, since they assume semi-static system dynamics. This works well since weather follows physical laws. Transformer based models, like PatchTST, run attention over patches on the sequences. They achieve state of the art results on weather forecasting. We could employ a station specific PatchTST encoder and then run attention between stations in a specific region. Any further work should explore a temporal encoding component and a cross station mixing component, not just limited to a GNN.

References:

[1] Breiman, L. (2001). Random forests. Machine learning, 45(1), 5-32.

[2] Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. Annals of statistics, 1189-1232.

[3] Jakaria, A. H. M., Hossain, M. M., & Rahman, M. A. (2020). Smart weather forecasting using machine learning: A case study in Tennessee. arXiv:2008.10789. https://arxiv.org/abs/2008.10789

[4] Price, I., Sanchez-Gonzalez, A., Alet, F. et al. Probabilistic weather forecasting with machine learning. Nature 637, 84–90 (2025). https://doi.org/10.1038/s41586-024-08252-9

[5] Singh, V., Das, A., Jadhav, S. et al. Hyper-Localized Weather Forecasting System. Journal of Information Systems Engineering and Management (2025). https://www.researchgate.net/publication/391511327_Hyper-Localized_Weather_Forecasting_System

[6] Zhang, H., Liu, Y., Zhang, C., & Li, N. (2025). Machine learning methods for weather forecasting: A survey. Atmosphere, 16(1), 82. https://doi.org/10.3390/atmos16010082

Contributions

Name	Midterm Contributions
Everett	PCA, Github Maintenance, GNN
Nhi	Autoencoder, Slides
Ruby	Feature engineering, Slides, GPR + GNN write-ups
Moksh	LMNN Results and XGBoost results + GNN Optimization

Name	Midterm Contributions
Jackson	Data aggregation, GitHub Pages, GPR training + analysis

Gantt Chart

https://gtvault-my.sharepoint.com/:x:/g/personal/epollard9_gatech_edu/EftWT4f2LUtDt8bhLDkBoUIBAOp1QbAZ8qWRGja2OSnbug?e=bJp7tu

YouTube Video

https://www.youtube.com/watch?v=-72Rh3EwrDw
