Stay Report

To: GeoSphere Austria, Vienna, Austria
Period: September 16th – October 11th, 2024
Topic: Work on Machine Learning Methods for Radiation Post-processing
Supervisors: Mag. Alexander Kann and Irene Schicker, PhD

Machine Learning Methods for Radiation Point-based Post-processing

1. Introduction

I stayed at GeoSphere Austria for four weeks, learning and working on Machine Learning (ML) methods that can be used for the post-processing of several different meteorological parameters. Currently, there are only a few post-processing methods used at DHMZ (e.g., neighborhood, analogs). Machine learning has become a vital tool in a wide range of industries, particularly in weather forecasting and renewable energy. With the growing demand for accurate solar energy predictions, there's a need for advanced ML techniques that can analyze and predict weather variables with high precision. These predictions are crucial for optimizing renewable energy production, especially for solar power generation, which heavily depends on weather conditions such as temperature, cloud cover, and solar radiation.

At this point, no post-processing technique is implemented at DHMZ to provide a better forecast for solar radiation or solar power prediction. Thus, at this point, the focus is on Surface short-wave (solar) radiation downwards (*ssrd*), and in the future, the plan is to use the additional module to predict power generation as well. However, due to only limited experience with many ML methods, the focus was on understanding and testing different methods and developing a framework that can be used regardless of the exact dataset used. For that reason, and since it is generally applicable and potentially useful for other members of the community, simple examples with code snippets are included in this report.

2. Methods

Several ML techniques have been explored in our study to model and predict meteorological variables. There are a few basic methods that I used to understand how often used machine learning algorithms work, then I investigated deeper several methods such as Random Forest, or Long Short Term Memory. Below is a summary of each method I've investigated.

i. LOG-based methods

k-Nearest Neighbors (kNN)
kNN is an instance-based method that belongs to the group of algorithms based on logistic regression. It makes predictions by finding similarities between data, but it does not build a direct model.
How it works: For each new data point to predict (e.g., tomorrow's temperature), the algorithm looks for the *k* nearest neighbors in the dataset. These neighbors are data points with the closest predictor

values (in this example: temperature, wind, day of the year). Then, kNN averages the target variable (in this case, tomorrow's temperature) of these nearest neighbors to make a prediction.

Key features:

- Nonlinear model**:** kNN does not assume linearity but uses local information to make predictions.
- Advantages**:** Simple to implement and understand.
- Disadvantages**:** Slow with large datasets because it has to search through the entire dataset for each prediction.

```
# Example
# k-Nearest Neighbors (kNN)
from sklearn.neighbors import KneighborsRegressor
...
knn = KNeighborsRegressor(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)
```

Multivariate Adaptive Regression Splines (MARS)
MARS is a more flexible regression method that can model complex and nonlinear relationships between variables. It belongs to the group of nonlinear regression models.
How it works: MARS uses splines—pieces of linear models that are connected at points called *knots*. Each segment between knots has its simple linear function. The algorithm automatically searches for optimal knots in the data to model complex relationships between predictors and the target variable.

Key features:

- Adaptive model**:** MARS can adjust its model based on the complexity of the relationships between variables.
- Advantages**:** Handles nonlinearities and interactions among variables well and automatically selects key predictors.
- Disadvantages**:** More complicated to interpret compared to simpler methods.

```
# Example:
# Multivariate Adaptive Regression Splines (MARS)
from pyearth import Earth
...
mars = Earth(feature_importance_type='gcv')
mars.fit(X_train, y_train)
y_pred_mars = mars.predict(X_test)
```

ii. Linear regression-based ML

Bayesian Ridge Regression

Bayesian Ridge Regression seeks a linear relationship between input variables (e.g., today's temperature, wind speed, day of the year) and the output variable (tomorrow's temperature). Bayesian Ridge assigns a coefficient to each variable based on probability distribution and uses these coefficients to predict tomorrow's temperature. Instead of just finding the "best" coefficients for these variables, it uses Bayes' theorem to estimate the probability of different models, incorporating uncertainty into the model. For instance, you would train the model on data from the past 5 years to predict tomorrow's temperature. If today's temperature correlates strongly with tomorrow's

temperature, the model will assign it a higher weight. Variables like wind speed or day of the year might have more or less influence depending on their contribution to prediction accuracy.

Key features:

- Prediction: Bayesian Ridge provides a prediction with a measure of uncertainty, offering not just a forecasted temperature but also a range of likely values.
- Advantage: Robust to multicollinearity—when variables are correlated, it handles that well.
- Disadvantage: Can be slower with large datasets due to the complexity of computing probability distributions.

```
# Example:
# Bayesian Ridge Regression
from sklearn.linear_model import BayesianRidge
...
bayesian_ridge = BayesianRidge()
bayesian_ridge.fit(X_train, y_train)
y_pred_bayesian = bayesian_ridge.predict(X_test)
```

Elastic Net

Elastic Net combines the advantages of Lasso (L1 regularization) and Ridge regression (L2 regularization). It uses parameters alpha (the strength of regularization) and l1_ratio (the balance between L1 and L2 regularization). Lasso attempts to reduce coefficients for irrelevant variables, while Ridge regularizes the model to prevent coefficients from becoming too large. Elastic Net uses both approaches to avoid overfitting and improve predictions when variables are highly correlated.

For example, Elastic Net would use temperature, wind, and day-of-the-year data from the past 5 years to train the model. The model balances complexity via regularization, finding the optimal coefficients for each variable.

Key features:

- Prediction: Elastic Net provides an accurate prediction without the uncertainty measure seen in Bayesian Ridge.
- Advantage: Works well with many interrelated (multicollinear) variables and reduces the influence of less relevant variables.
- Disadvantage: Careful calibration of regularization parameters (L1 and L2) is required, needing extra processing.

```
# Example:
# Elastic Net Regression
from sklearn.linear_model import BayesianRidge, ElasticNet
...
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)  # Postavljanje regularizacije
elastic_net.fit(X_train, y_train)
y_pred_elastic = elastic_net.predict(X_test)
```

iii. Support Vector Regression (SVR)

SVR is a method based on Support Vector Machines (SVM), but adapted for regression tasks. It uses kernel functions to transform data into higher dimensions, allowing the model to find a hyperplane that best predicts target values.

How it works: SVR aims to find a hyperplane that minimizes prediction error while allowing some deviation (epsilon) from actual values. Support vectors are the key data points defining this hyperplane, while other points may have less influence.

Key features:

- Kernel trick: SVR uses kernel functions (e.g., linear, polynomial, RBF) to transform data into higher dimensions, enabling the modeling of complex relationships.
- Advantages: Works well with small and medium-sized datasets and nonlinear relationships.
- Disadvantages: Sensitive to data scaling and can be slow on larger datasets.

```
# Example:
# SVR
from sklearn.svm import SVR
from sklearn.preprocessing import StandardScaler
...
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
svr = SVR(kernel='rbf', C=1.0, epsilon=0.1)
svr.fit(X_train_scaled, y_train)
y_pred_svr = svr.predict(X_test_scaled)
```

iv. Tree-based ML

Gradient Boosting Machine (GBM)

Gradient Boosting also uses decision trees as its basic units but works in a completely different way than Random Forest. While Random Forest trains many trees in parallel, GBM trains them sequentially. Each new tree aims to correct the errors of the previous trees. GBM improves the model gradually, with each new tree focusing on reducing the residual error of the previous model.

Key features:

- Prediction: GBM can give very accurate predictions by continuously correcting the errors of previous models. However, it can be sensitive to overfitting if too many trees are trained or if regularization isn't properly applied.
- Advantage: Highly accurate and flexible, especially on complex, nonlinear data.
- Disadvantage: Training can be slow, especially with large datasets, and it can be sensitive to overfitting if not handled carefully.

```
#Example:
from sklearn.ensemble import GradientBoostingRegressor
...
gb = GradientBoostingRegressor()
gb.fit(X_train, y_train)
grad_for=gb.predict(X_test)
```

From this point on, there are several methods that I used to modify and fit methods further.

Random Forest

Random Forest uses an ensemble approach—it builds many decision trees, each trained on randomly selected data subsets. The predictions of each tree are combined (usually averaged) to make the final prediction. Each tree uses random subsets of both data and predictors at each split, reducing overfitting by preventing individual trees from becoming too closely tailored to specific training data characteristics.

In a previously used example, Random Forest would use temperature, wind speed, and day of the year to train a set of trees to predict tomorrow's temperature.

Key features:

- Prediction: Highly accurate predictions as the algorithm averages the results of multiple trees, helping reduce variance and overfitting.
- Advantage: Robust to outliers and overfitting, works well with many variables or nonlinear relationships.
- Disadvantage: Can be slower in training and prediction when dealing with very large datasets and numerous trees.

```
#Example:
from sklearn.ensemble import RandomForestRegressor
...
rf = RandomForestRegressor(n_estimators=100)
rf.fit(X_train, y_train)
rand_for=rf.predict(X_test)
```

v. Multilayer Perceptron (MLP)

MLP is a type of artificial neural network consisting of at least three layers: an input layer, one or more hidden layers, and an output layer. Each layer contains a set of neurons connected through weights. MLP follows a feedforward architecture, meaning that data flows in one direction, from the input to the output layer.

The learning process involves adjusting the weights using the backpropagation algorithm, which minimizes the model's error by updating the weights after each pass through the data (epochs).

Key Features:

- Non-linear model: MLP can capture complex and non-linear relationships between input and output variables.
- Advantages: Flexibility to model complex relationships; works well with large datasets.
- Disadvantages: Can be slower to train and sensitive to overfitting, especially without proper regularization.

```
# Example:
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import StandardScaler
...
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_train)
...
mlp = MLPRegressor(hidden_layer_sizes=(100,), activation='relu', solver='adam',
   max_iter=500)
mlp.fit(X_scaled_train, y_train)
y_pred = mlp.predict(X_test)
```

vi. Long Short-Term Memory (LSTM)

It is often problematic to model data that involves causality and correlation in the time component, especially when dealing with forecast time series, i.e., have forecast initialization time series, and each initialization has a time window (e.i. lead time) ahead, often longer than 24 hours. LSTM is a specialized type of Recurrent Neural Network (RNN) designed to handle time series and sequential data. LSTM employs memory cells that help capture long-term dependencies in the data, overcoming the vanishing gradient problem that regular RNNs face.

LSTM networks have three primary gates to control information flow:

- Forget gate decides which information from previous steps to discard.
- Input gate controls which new information to store.
- The output gate determines which part of the stored information to use for the output.

Key Characteristics:

- Sequential modeling: Ideal for time series data, such as weather forecasts with time steps.
- Advantages: Excellent at retaining long-term dependencies and correlations across time.
- Disadvantages: Slower training, high computational complexity, and requires careful scaling.

```
# Example:
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from sklearn.preprocessing import MinMaxScaler
...
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X_train)
...
# Reshaping data for LSTM (3D: samples, time steps, features)
# Train-test split
...
# Creating LSTM model
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(X_train.shape[1], X_train.shape[2])))
model.add(Dense(leadtime))
model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=200, batch_size=32, verbose=0)
y_pred = model.predict(X_test)
```

3. Predictor variables

Meteorological forecasting models often rely on various predictor variables. These are usually meteorological parameters, including, for instance, temperature, cloud cover, wind speed, and visibility when forecasting *ssrd* (Surface short-wave radiation downwards). I tested several combinations and concluded that in addition to these standard predictors, time-based information (such as the hour of the day and month of the year) is particularly useful.

I was working with *ssrd* 6-hourly cumulative sums, and *ssrd* is a parameter with a strong daily cycle (no radiation in the nighttime). However, at first, I was not able to produce an output that captured the amplitude and periodic behavior of such a parameter (Example in Figure 1). Thus, I decided to try the fraction of night hours as an additional predictor. That additional predictor helped to model a reasonable daily cycle and thus notably improved the accuracy of the model.

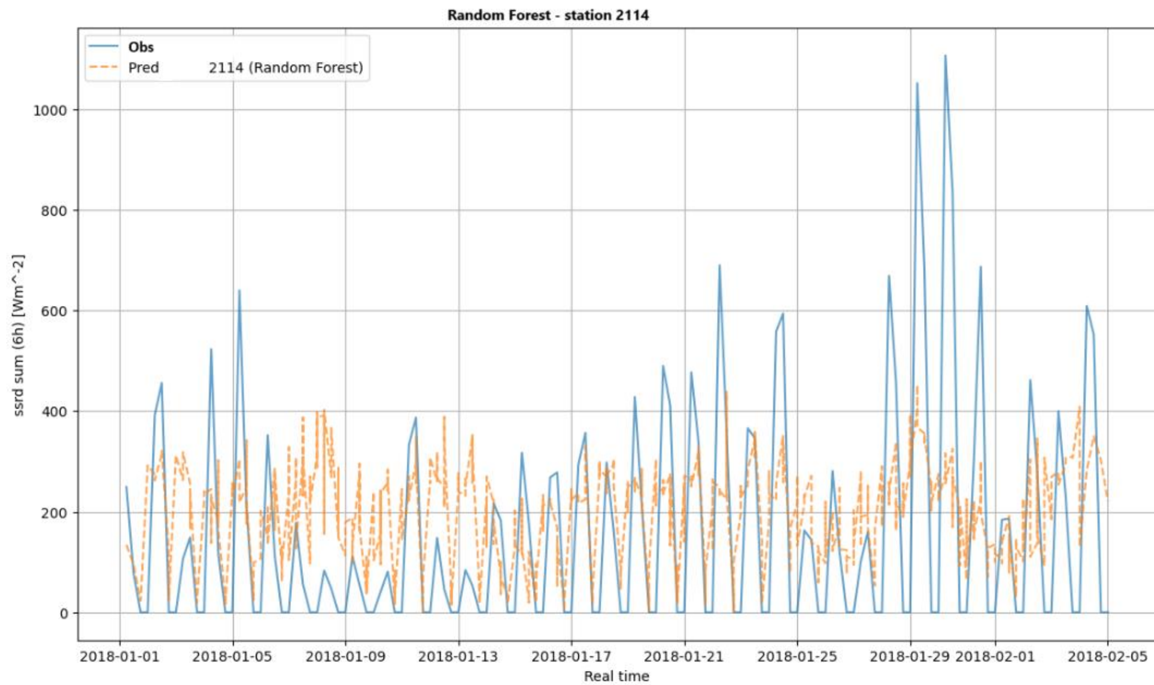

Figure 1. An example of a poorly captured daily cycle and amplitude of ssrd

The fraction of night hours is calculated based on solar events at different geographic locations. This predictor allowed us to model how much of the forecast period occurs during nighttime, providing a better representation of variables like solar radiation and temperature (Figure 2).

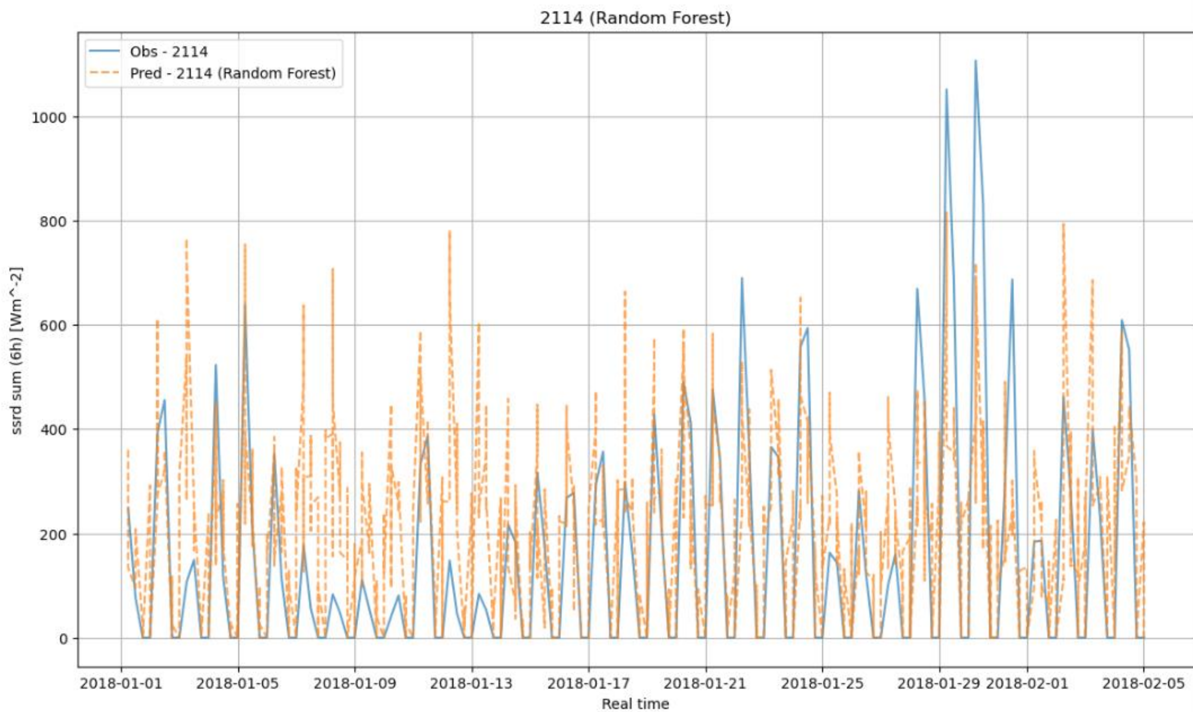

Figure 2. An example of a better captured daily cycle and amplitude of ssrd using the „night fraction" as an additional predictor and the same data as in Figure 1

Here is an example of how I computed the night fraction for each station:

```
from astral.sun import sun
from astral import LocationInfo
def calculate_night_fraction(latitude, longitude, timestamp):
    city = LocationInfo(latitude=latitude, longitude=longitude)
    s = sun(city.observer, date=timestamp)
    sunset = s['sunset']
    sunrise = s['sunrise']

    if timestamp < sunrise or timestamp > sunset:
        return 1.0  # It's night time
    else:
        day_duration = (sunset - sunrise).total_seconds()
        time_since_sunrise = (timestamp - sunrise).total_seconds()
        return 1.0 - (time_since_sunrise / day_duration)

# Apply this function across a dataset of stations and timestamps
df['night_fraction'] = df.apply(lambda row:
calculate_night_fraction(row['latitude'], row['longitude'], row['valid_time']),
axis=1)
```

4. Parameter Tuning

When modeling using ML methods, parameter tuning is crucial because the performance of machine learning models highly depends on the values of specific parameters, often referred to as hyperparameters. These parameters control the model’s behavior during training, and tuning them allows the model to generalize better to unseen data, improving overall accuracy, reducing overfitting, and optimizing the computational resources required for training.

Some typical examples of hyperparameters include:

- Number of layers and neurons in neural networks (e.g., in MLP and LSTM).
- Learning rate and optimizer type.
- Regularization parameters like alpha (in a ridge or lasso regression).
- Max depth and number of trees (in Random Forest).
- Kernel type and C parameter (in Support Vector Regression).

Without tuning, models may underperform, as default parameters might not fit the complexity or scale of the problem. Through careful tuning, models can achieve their optimal performance.

Even though I did not manage to investigate all the possibilities of optimally tuning hyperparameters due to time constraints, I did try to include such a 'placeholder' in my algorithms. The systematical approach to gathering the results is lacking at this point, but I applied parameter tuning to various models, specifically targeting methods like Random Forest, MLP, and LSTM, where hyperparameters significantly affect model performance. To automate this tuning, I utilized approaches like *GridSearchCV* and *RandomizedSearchCV*. Thus, I developed a frame that I can afterward use to optimize ML modeling further.

Random Forest - I optimized parameters like the number of trees (n_estimators) and maximum tree depth (max_depth), ensuring that the Random Forest was neither too shallow (which would lead to underfitting) nor too deep (which could overfit the data):

```
param_grid_rf = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10, 20, 30]
}
grid_search_rf = GridSearchCV(RandomForestRegressor(), param_grid_rf, cv=3)
grid_search_rf.fit(X_train, y_train)
```

MLP: For MLP, I tuned hyperparameters like the number of hidden layers, the activation function, and the solver (optimizer). The goal was to balance the model's complexity and learning rate to ensure it could capture the underlying patterns without overfitting:

```
param_grid_mlp = {
    'hidden_layer_sizes': [(100,), (50, 50)],
    'activation': ['relu', 'tanh'],
    'solver': ['adam', 'sgd'],
    'alpha': [0.0001, 0.001]
}
grid_search_mlp = GridSearchCV(MLPRegressor(), param_grid_mlp, cv=3)
grid_search_mlp.fit(X_train, y_train)
```

LSTM: For LSTM, I adjusted parameters such as the number of LSTM units (neurons in the LSTM layers), the number of epochs, batch size, and the learning rate to account for the model's sequential nature. I used grid search to find the optimal architecture:

```
param_grid_lstm = {
    'batch_size': [32, 64],
    'epochs': [100, 200],
    'units': [50, 100]
}
random_search_lstm = RandomizedSearchCV(LSTMRegressor(), param_grid_lstm, cv=3)
random_search_lstm.fit(X_train, y_train)
```

5. MultiOutputRegressor (MOR)

MultiOutputRegressor (MOR) is a meta-estimator that extends any base regressor to handle multiple output variables simultaneously. Normally, ML models are designed to predict a single target variable, but MOR enables the model to produce multiple predictions at once. MOR works by fitting a separate model for each output variable, allowing us to handle complex forecasting problems where multiple target variables (e.g., forecasting several hours into the future) need to be predicted simultaneously.

I introduced MOR because this forecasting task involved predicting multiple time steps in the future (21 forecast steps). In this context, each forecast step is treated as a separate output variable, and MOR allowed me to:

- Efficiently predict all steps at once: Rather than running individual models for each forecast step, MOR provided a framework to handle all predictions in parallel.
- Better capture relationships between steps: Although each model in MOR is independent, using the same underlying predictors and fitting them together could potentially capture interdependencies between steps, improving the overall forecast.
- Consistency with time-dependent data: By training on all forecast steps together, MOR ensured consistency in predicting time-dependent outputs, making it better aligned with the temporal nature of meteorological data.

In other words, instead of training 21 separate models (one for each step), MOR simplified this process by bundling the models together while maintaining separate outputs. One could apply MOR to a variety of base models. I used it mostly for Random Forest and MLP, whereas I adapted LSTM somewhat differently since the idea of modeling time series is already incorporated in such a model.

This is a code example for MOR wrapping Random Forest:

```
from sklearn.multioutput import MultiOutputRegressor
from sklearn.ensemble import RandomForestRegressor
```

```
base_model = RandomForestRegressor()


mor_model = MultiOutputRegressor(base_model) # Wrap it with MultiOutputRegressor
mor_model.fit(X_train, y_train) # Fit on training data
y_pred = mor_model.predict(X_test) # Predict
```

6. Preliminary results

The Figure 3 illustrates the Root Mean Square Error (RMSE) across all stations for three different machine learning models: Random Forest (RF), Multi-Layer Perceptron (MLP), and Long Short-Term Memory (LSTM). The RMSE values are computed at various forecast hours (from 0 to 120) and depict It can be noticed that the RMSE follows a recurring cycle with peaks at specific intervals (~24-hour periods), reflecting the diurnal cycle of solar radiation, which aligns with the typical daily variations in solar energy availability. For all models, RMSE decreases significantly after the initial peak of each cycle, indicating that the models perform better during periods with lower variability, such as during nighttime when solar radiation is minimal, which i san expected result.

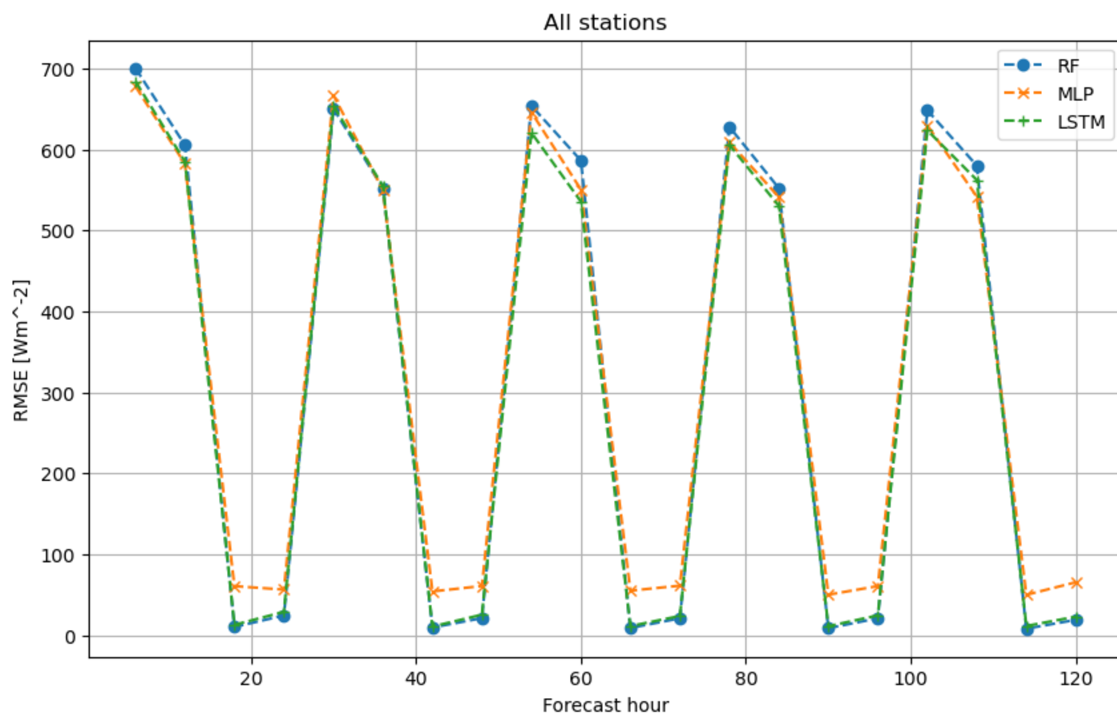

Figure 3. An example of the intercomparison of three trained ML models (RF, MLP and LSTM) for different forecast hours, across 15 locations in Austria avereged for 2 months (January and July).

Across most forecast hours, the Random Forest model (blue line) exhibits consistently higher RMSE values compared to the Multi-Layer Perceptron (orange line) and Long Short-Term Memory model (green line). This discrepancy is particularly noticeable at the peaks of each cycle, where Random Forest demonstrates sharper increases in error, especially around the 24-hour intervals (hours 0, 24, 48, etc.). In contrast, MLP and LSTM models show closer performance, with MLP marginally outperforming RF at certain points. Notably, LSTM tends to yield the lowest RMSE across the forecast horizon, particularly around forecast hours 20, 44, 68, and 92, which suggests its stronger capability in capturing temporal dependencies in the data.

This improved performance from LSTM, likely due to its ability to effectively model time series, is evident even when the MultiOutputRegressor (MOR) was applied to non-time-series models such as RF and MLP. However, it is important to note that these models have not undergone exhaustive hyperparameter tuning. With further optimization, the results might vary significantly, possibly leading to more competitive performances across all models.

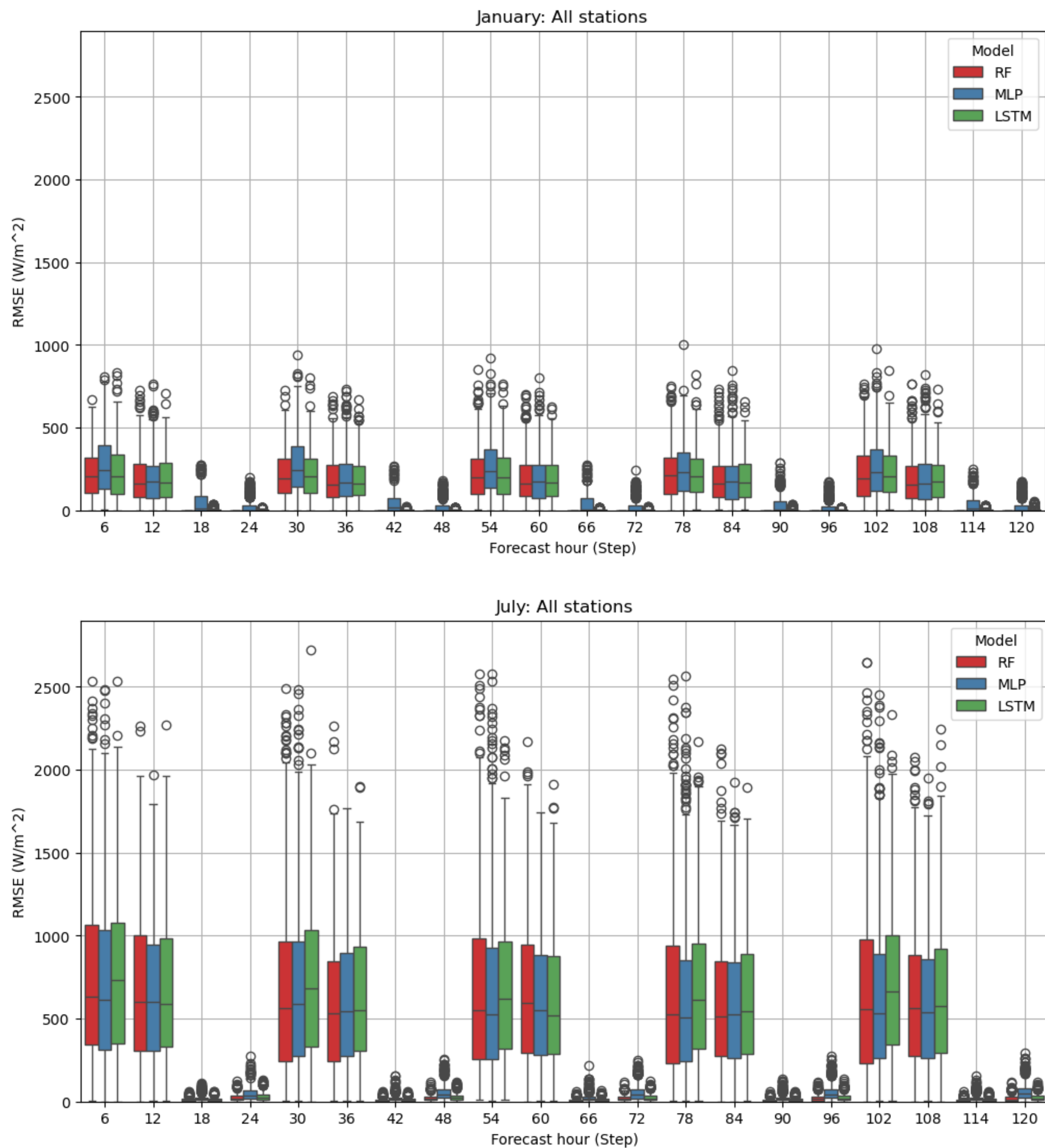

Figure 4. The intercomparison of different ML models' error distribution acress 15 locations in Austria, depending on the forecast hour for January (top) and July (bottom).

The boxplots comparing RMSE across different locations depending on forecast steps for all models—Random Forest, MLP, and LSTM—indicate some key seasonal differences and patterns. In January, the

error distributions across forecast steps are relatively consistent, with most models showing similar performance throughout the day. The RMSE values are generally lower, which can be attributed to the shorter daylight hours and reduced solar radiation variability during the winter months.

In contrast, the results for July show a clear increase in RMSE across most forecast steps, particularly during daylight hours. This heightened error can be attributed to increased solar radiation and the higher variability in weather patterns during the summer months. The differences between models remain subtle, with RF slightly underperforming in certain forecast steps compared to MLP and LSTM. The overall distribution of RMSE for each model is similar, but there is a more noticeable spread across stations in July compared to January. This indicates that spatial variability plays a larger role in model performance during the summer, with certain locations potentially experiencing more pronounced forecast errors.

All three models—Random Forest (RF), MLP, and LSTM—demonstrate comparable distributions, with only slight deviations between the models, suggesting that none stands out significantly in terms of error.

In conclusion, while the current models perform reasonably well, there is clear room for improvement, particularly during the summer months. By focusing on seasonal feature engineering, enhanced temporal and spatial modeling, and exploring ensemble approaches, the forecast accuracy can likely be improved.

7. Acknowledgments

The authors would like to thank our colleagues in the ACCORD and RC-LACE consortiums for their support of our work.