BDM - Assignment 2

Christian Bank Lauridsen (chbl@itu.dk)

GitHub Repository

Table 1: Small snippet of the weather forecast data

Direction	Lead hours	Source time	Speed
SSE	1	1639227600	11.17600
SSW	1	1639238400	8.04672
SW	1	1639270800	9.83488

1 Preprocessing

Before setting up my pipeline, I had to preprocess my data from the two given datasets.

Listing 1: Loading my two datasets

```
power_df = read_csv_with_time_index(POWER_CSV)
wind_df = read_csv_with_time_index(WEATHER_CSV)
```

1.1 Filtering the data

The first dataset to analyse was the power set that contained the target values my models need to predict. It consisted of three columns: `ANM`, `Non-ANM`, and `Total` (power generated). Inspecting this data, I only needed the `Total` column, as this contained the target values (the total generated power) my models needed to predict. I was therefore only interested in the total power generation, hence I filtered away the other unnecessary information.

My other dataset to analyse was the weather forecast datasets that consisted of four columns: `Direction` (of the wind), `Lead hours`, `Source time`, and `Speed` (of the wind). By inspecting this data in Table 1, we see that both the `Lead hours` and `Source time` columns did not provide any relevant information that was useful for training my models, since for each measurement, the values were almost identical in every row. This means that these features were irrelevant to train with, since, regardless of using them or not, they did not provide any new information for a feature vector. As a result, I only used the features `Direction` and `Speed` for training the models.

Listing 2: Filtering the two datasets

```
power_df = power_df[["Total"]]
wind_df = wind_df[["Speed", "Direction"]]
```

1.2 Merging the dataframes

The next part of the preprocessing of my data was to merge my two datasets together. Each datapoint from both sets contained a timestamp of the measurement, hence this was the column to merge with.

However, when inspecting the datasets, I identified a problem: the power generation datapoints are generated every minute, while the weather forecast datapoints are generated every 3 hours. Hence, the ratio is 1:180 in measurements, meaning that for every weather forecast datapoint, 180 power generation datapoints are generated. This makes aligning the datasets together problematic since I would need to answer which power value does the forecast represent, in other words, which of the 180 power measurements represented the correct weather forecast measurement?

To tackle this problem, I used three different approaches.

1.2.1 Inner join

The first approach was to join the datasets using a simple inner join on the matching timestamps.

Listing 3: Using a simple inner join to merge the datasets

```
joined_dfs = power_df.join(wind_df, how="inner")
```

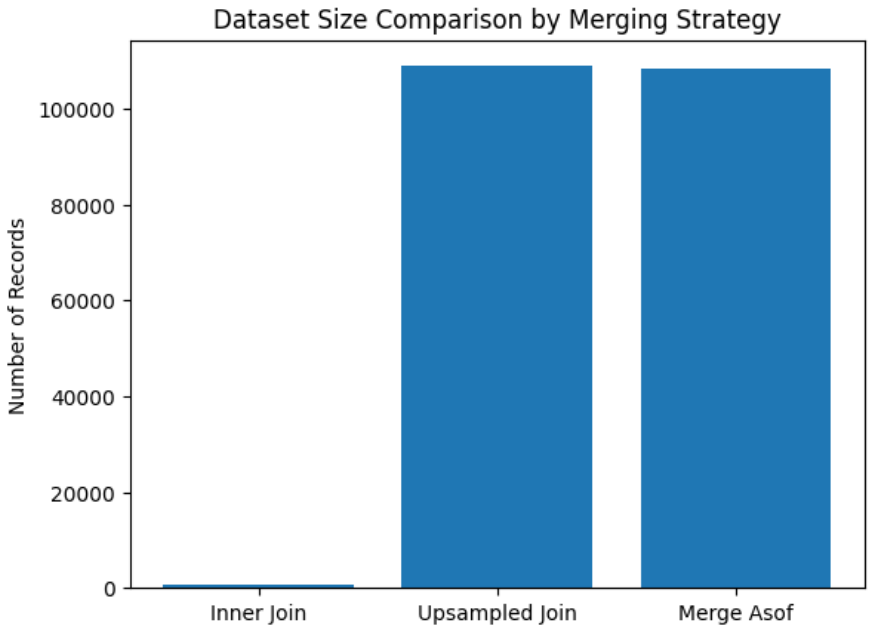

Figure 1: Comparison with the resulting data sizes after merging the power and weather forecast datasets using the three different approaches

The advantage of this approach is that it only matches on the correct matching timestamp and does not make any new assumptions about the data. As a disadvantage, we therefore lose a significant amount of data points, which we can see in Figure 1. Most of our data is discarded due to the unmatching time intervals, and as a result of doing the inner join, we only have a data size of 610 datapoints.

1.2.2 Upsampling the data

My second approach was to upsample my weather forecast data to a one-minute frequency, to match the same number of data points for the power dataset and then do an inner join between the datasets.

Upsampling was done by using the method `resample('1min')`, which increases the frequency of weather data to one data point per minute. The `ffill()` (forward fill) method, was then applied to fill in the new values by copying the most recent known weather measurement forward, until a new measurement becomes available.

Listing 4: Using upsamplig followed by a inner join to merge the datasets

```
weather_upsampled = wind_df.resample('1min').ffill()
upsampled_merged_dfs = power_df.join(weather_upsampled, how="inner")
```

The advantage of this approach is that every power measurement gets a weather measurement, meaning we have a lot more datapoints (108847), which we can see in Figure 1. However, the disadvantage of doing this is that it makes the assumption that the weather conditions remain constant until the next measurement, which may not be accurate. In other words, this approach assumes that the weather conditions remain the same for the whole 3 hours, which is very unrealistic.

In addition to this, this approach requires two processing steps: first, resampling the weather dataset, and then performing the join. This increases the latency of computation.

1.2.3 ASOF Merge

My final approach was conceptually similar to upsampling, but it performs the alignment and merging in a single step, thereby removing the need to go through the weather dataset twice.

Here, the `merge_asof()` method is applied using the nearest timestamp within a defined tolerance threshold. In this case, I used a 90-minute threshold, meaning that each power measurement is matched to the closest available weather forecast measurement within a maximum difference of 1.5 hours.

Listing 5: Using ASOF merge to merge the datasets

```
merged_dfs = pd.merge_asof(
    power_df.sort_index(),
    wind_df.sort_index(),
    left_index=True,
    right_index=True,
    direction='nearest',
    tolerance=pd.Timedelta('90min')
)
```

With this approach, half of the 180 power measurements within a 3-hour weather forecast interval are matched to the previous weather observation, while the other half are matched to the following one, depending on which timestamp is closer in time.

Like the upsampling approach, the ASOF merge still assumes that the weather conditions are constant within the matching interval. However, in this case, the assumption is limited to a maximum of 1.5 hours rather than the full 3-hour interval, making it somewhat a more realistic representation. This method gets 108305 data points (see Figure 1), and is computationally more efficient, as it avoids upsampling the weather data before merging.

1.3 Handling missing data

After merging the datasets, I have forgotten to remove missing data or measurements that might contain null values. However for a future improvement, this should be part of the preprocessing.

1.4 Encoding the weather direction

To generate the feature vectors of my weather forecast data, the `Direction` column would need to be converted to numerical values. The simplest approach is to use a simple encoding (one-hot vector) where I map the current string direction to a value from 0 to 15, or map them to a degree from 0 to 360.

However, doing this gives me a problem, for example, the values 1 and 15 or 0 and 360 will be totally different in terms of the distance between them, making these features very distinct. But in reality, these features might represent similarities, since they both might represent a close or similar direction. In other words, it removes the circular relationship between them. For example, the directions N or NNW are physically close, but the one-hot encoding will assume they are very far apart (N=0°, and NNW=337°), and assume instead that N and S are closer (S=180°), but in reality they should be far apart.

To keep the circular relationship, I instead map the direction to a degree (sixteen values), which corresponds to the compass directions in degrees (see Listing 6, lines 1-7).

To handle the circularity, I further encode them into sine and cosine representations with the method `encode_wind_direction()` (see Listing 6, lines 9-15). This method converts the dataframe from instead of having two columns `Direction` and `Speed` to having 3 columns: `Dir_x`, `Dir_y`, and `Speed`. The two new columns `Dir_x` and `Dir_y` represent the cosine and sine values, which together maintain the circular relationship for the directions. But this approach also adds an extra feature and increases the length of the feature vector.

Another approach for the encoding and still keep two features, was to combine the speed and directions (cosine and sine directions) by multiplying the direction with the `Speed`. This I did with another encoding method called `wind_vector()` (see Listing 6, lines 17-23). This method converts the dataframe to only have two features `Wind_x` and `Wind_y`.

To summarise, I used two different encodings, which I used to train my models with.

Listing 6: Mapping function which map a direction to its coresponding compass directin in degrees

```python
# Encode the weather direction to degrees (numerical values)
direction_to_deg = {
    'N': 0, 'NNE': 22.5, 'NE': 45, 'ENE': 67.5,
    'E': 90, 'ESE': 112.5, 'SE': 135, 'SSE': 157.5,
    'S': 180, 'SSW': 202.5, 'SW': 225, 'WSW': 247.5,
    'W': 270, 'WNW': 292.5, 'NW': 315, 'NNW': 337.5
}

# Sin/cos encoding
def encode_wind_direction(df):
    radians = np.deg2rad(df['Direction']) # Convert degrees to radians
    df['Dir_x'] = np.cos(radians)  # Cosine component
    df['Dir_y'] = np.sin(radians)  # Sine component
    df.drop(columns=['Direction'], inplace=True)
    return df

# Wind vector encoding (combining speed and direction x or y direction)
def wind_vector(df):
    radians = np.deg2rad(df['Direction']) # Convert degrees to radians
    df['Wind_x'] = df['Speed'] * np.cos(radians)  # X component
    df['Wind_y'] = df['Speed'] * np.sin(radians)  # Y component
    df.drop(columns=['Direction', 'Speed'], inplace=True)
    return df
```

1.4.1 Analysing the data after the encoding

Before encoding, I observed a positive relationship between speed and power generation, as shown in Figure 2. However, this relationship is not fully linear, hence only using Linear regression models to train with is not viable, and I would need to include models that capture non-linearity.

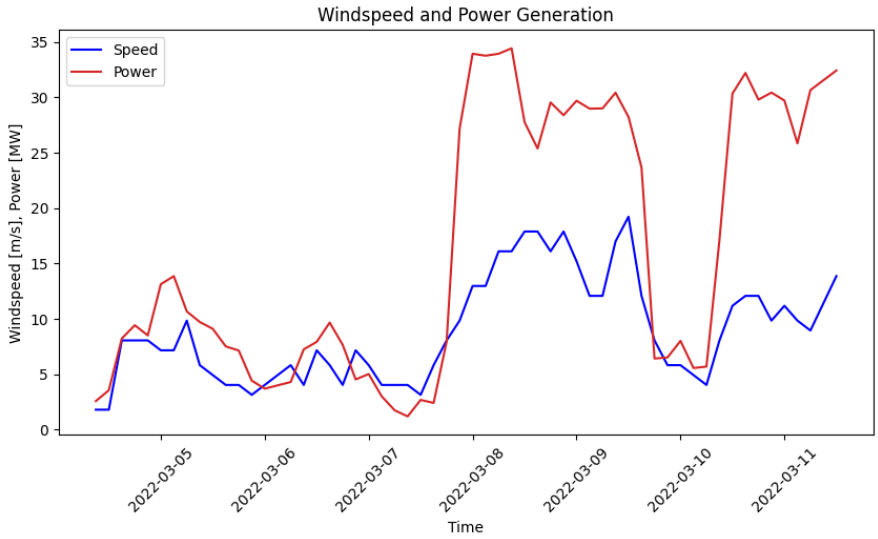

Figure 2: Timeseries plot of speed and power generation

After doing the encoding, I observed the data distribution. I could see that the strong winds come from the NE towards the SW, as shown in Figure 3. In this assignment, I do not evaluate whether wind direction is a useful feature. To determine this, I could do a comparative experiment which compares models with and without wind direction and use MLflow to track the results, by seeing it helps reduce errors.

Another approach is to compute the correlation between wind direction and the power generated. If the direction strongly correlates with the power generated when the wind comes from optimal angles, the feature is relevant.

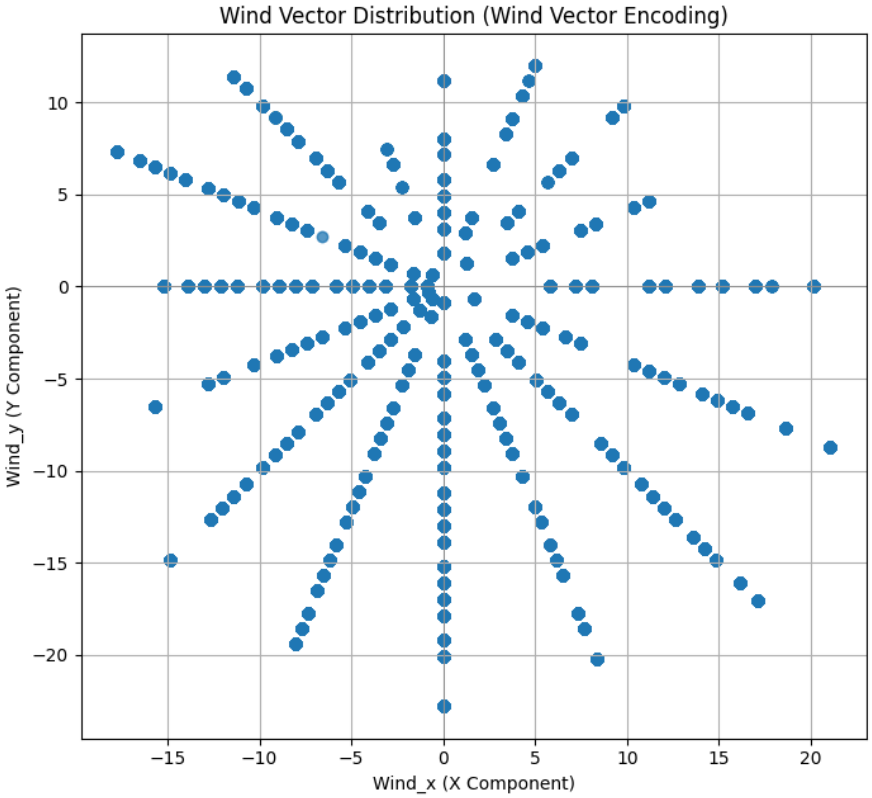

Figure 3: The datapoint distribution of the dataset after doing wind_vector encoding

2 Building and training my models

To build and train my models, I used the pipeline from sklearn, and used it together with four different regression approaches: `LinearRegression`, `PolynomialFeatures`, `RandomForestRegressor`, and `GradientBoostingRegressor`. I chose these models as they can capture the complex relationship, using the linear regression as a baseline model. Polynomial regression was included to capture basic non-linear relationships, which we observed in Figure 2. Random forest and gradient boosting were chosen as more advanced methods to capture non-linear patterns in the data.

The code for my Pipeline and models can be seen in Listing 7. My pipeline uses standardization (`StandardScaler`) instead of normalization. Standardization transforms the features to have a mean of 0 and a standard deviation of 1, and makes the model more robust to outliers while preserving a meaningful distribution of the data. Using normalization would scale the data to a fixed range of 0 to 1, and is therefore highly sensitive to outliers. One extreme outlier could compress the majority of valid readings into a tiny numerical value.

For tree-based models `RandomForestRegressor` and `GradientBoostingRegressor`, I configured two hyperparameters to control model complexity and prevent overfitting. `n_estimators=100` sets the number of decision trees in the ensemble. Using 100 trees provides a more robust estimate by averaging out the errors of individual trees, reducing overall variance. Using `max_depth=5` limits the maximum depth of each tree, and prevents the model from becoming too complex and overfitting the training data, ensuring better generalization to the test data.

Using these different models allowed for a fair comparison between simple and more sophisticated approaches, which makes it possible to evaluate whether increasing the model complexity leads to better performance.

Before training my model, I used the `train_test_split()` method to split my data into training and test data. By setting the parameter `shuffle` to false, I ensured that the timeseries data were persisted, and the temporal order would not be destroyed. The code for training my models can be seen in Listing 8.

For comparing the models, I used `MLflow` to set up multiple experiments and compare the performance and accuracy of the different models on different datasets.

Listing 7: My Pipeline and models

```
def build_pipeline(model):
    if isinstance(model, Pipeline):
        return model   # Polynomial pipeline already includes scaler
    else:
        return Pipeline([
            ("Scaler", StandardScaler()),
            ("regressor", model)
        ])

def get_models():
    return {
        "LinearRegression": LinearRegression(),
        "Polynomial_2deg": Pipeline([
            ("poly", PolynomialFeatures(degree=2, include_bias=False)),
            ("scaler", StandardScaler()),
            ("regressor", LinearRegression())
        ]),
        "Polynomial_3deg": Pipeline([
            ("poly", PolynomialFeatures(degree=3, include_bias=False)),
            ("scaler", StandardScaler()),
            ("regressor", LinearRegression())
        ]),
        "RandomForest": RandomForestRegressor(n_estimators=100, max_depth=5),
        "GradientBoosting": GradientBoostingRegressor(n_estimators=100,
            max_depth=5)
    }
```

Listing 8: The training of my models

```
def train_and_log(df, dataset_name, feature_cols, models_dict,
    base_experiment_name, all_runs):

   X = df [feature_cols]
   y = df ['Total']

   # Chronological Split for train and test sets
   X_train, X_test, y_train, y_test = train_test_split(
      X,
      y,
      test_size=0.2, # 80% for training, 20% for testing
      shuffle=False # Important for time series data
   )

   for model_name, model in models_dict.items():
        pipeline = build_pipeline(model)

        experiment_name = f"{base_experiment_name}_{model_name}"
        mlflow.set_experiment(experiment_name)

        # Fit model on training data
        pipeline.fit(X_train, y_train)
        y_pred = pipeline.predict(X_test)
        test_mae = mean_absolute_error(y_test, y_pred)
        test_mse = np.sqrt(mean_squared_error(y_test, y_pred))
        test_r2 = r2_score(y_test, y_pred)
        test_rmse = root_mean_squared_error(y_test, y_pred)

...
```

I do six experiments in total, in which I do one experiment with the SinCos encoding on the three different datasets, and I do the same experiment just for the WindVector encoding. Inside each experiment, I train all the models and track their runs inside MLflow. Each run will: (1) Fit the pipeline, (2) Predict on the test set, (3) Compute the metrics: MAE, MSE, RMSE and R2 score, and (4) Log and store the results.

Using MLflow, I identify and register the best-performing models for each encoding strategy. Although I run multiple experiments, I only register two models: the best model using SinCos encoding and the best model using WindVector encoding. For each encoding strategy, I compare experiment runs and select the model with the lowest MAE score to register in MLflow.

2.1 Deploying my models

Using MLflow to register my models provides several advantages, as you do not need to manually save model files, storing results in notebooks or do versioning through Git. MLflow automatically tracks experiments, including data, metrics, parameters, software environment used for training. This ensures full reproducibility and allows any model version to be restored and executed.

To make my models accessible to others, I deployed an MLflow server on an Azure VM. This setup exposes registered models and allows others to load models, inspect and interact with them.

Once a model is registered in MLflow, it can be deployed as an API, which enables sending requests for predictions for power generation based on weather forecast data. This supports collaboration and enables integration with external tools, which researchers or stakeholders can use for their applications that rely on the model's output.

2.2 Further improvements

I am currently only running a single experiment when training my models, which limits the ability to explore or optimise hyperparameters. I could improve the accuracy of models that make use of different hyperparameters by running multiple epochs in which I iteratively change the hyperparameters.

By registering each trained model version in MLflow, I can compare performance metrics between versions and evaluate how different hyperparameter configurations help improve the accuracy. This would help select more optimal models and better utilise MLflow.

Another potential improvement is to validate my choice of feature scaling. I currently use standardization to mitigate the effect of outliers, it would be beneficial to run a comparative experiment using normalization. By training the same models on normalized data, I could verify whether standardization is more beneficial for the datasets or normalization yields better results.

2.3 Is 90 days of data a good interval?

I am currently only taking into consideration the first 90 days as an interval of data points. This may be misleading since I would not consider seasonality, as wind patterns depend on the season. Only using 90 days (3 months) might capture the short-term behaviour, but not all the annual seasons. Using short intervals, e.g. 30-90 days, might help capture recent power generation behaviour, which helps to train faster with and store less data. However, you risk overfitting to short-term behaviour, which might not reflect the most recent weather conditions due to seasonal changes. Using long intervals, e.g. 180-365 days, helps capture seasonal wind changes, however, this is more expensive as it requires storing more data, which takes a longer time to train on. However, due to climate change, the data might shift, since old data may include weather conditions that no longer apply to the most recent conditions. Too much old or irrelevant data can harm the accuracy by introducing patterns that do not reflect the current behaviour of the power generation, thus, an increase in dataset size is not guaranteed to improve accuracy.

3 Evaluation

My evaluation metrics I use are MAE, MSE/RMSE, and R2 score. The MAE compares the deviation between the target and the prediction value. Both MSE and RMSE are similar to MAE, however, they penalise large errors more than MAE does. Lastly, R2 score shows how much variance there is.

Using these metrics helps compare models consistently and determine which performs best.

3.1 Comparing the best models

All model results can be seen in the Appendix.

3.1.1 Best model for SinCos Encoding

In Figure 4, we observe the MAE value of the best model using the SinCos encoding. The best model is the RandomForest model, using this model on top of the inner joined data set gives an MAE value of 3.32.

We can observe in Figure 5 that the predictions follow the underlying function of the power generation.

3.1.2 Best model for WindVector Encoding

In Figure 6, we observe the MAE value of the best model using the WindVector encoding. Again, we observe that the best MAE value of 3.55 is the RandomForest model trained on the inner joined dataset.

We also observe in Figure 7, that the predictions follow the underlying function of the power generation.

3.2 Reflection on my results

3.2.1 The best model

Random Forest outperformed Linear Regression because it naturally handles the non-linear nature of wind data, which affects the power generation. While Linear regression is simple, its assumptions are too simple for capturing the underlying nature of wind data, which resulted in an underperformance.

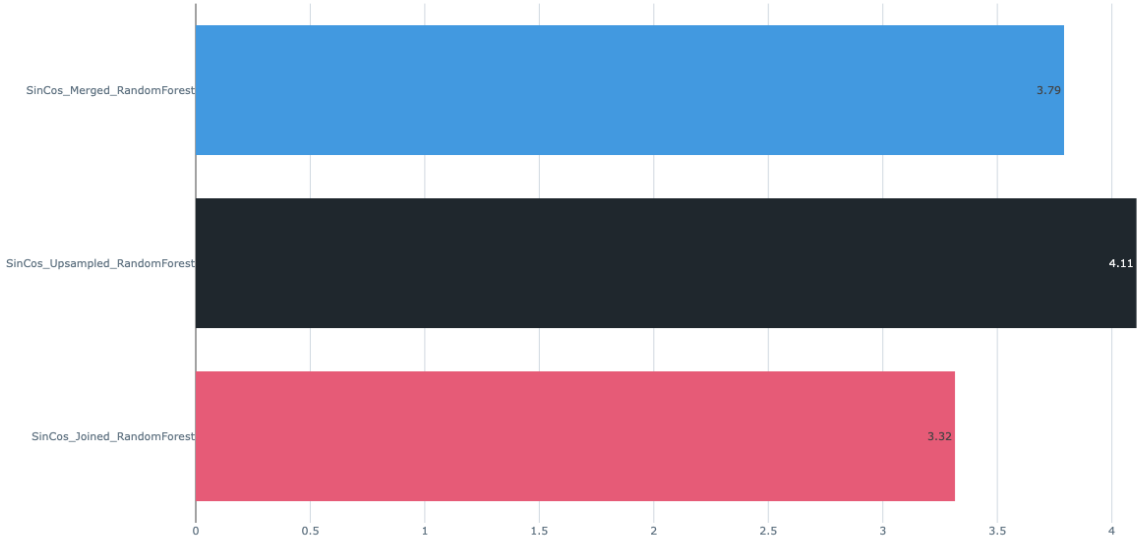

Figure 4: Best model for SinCos encoding

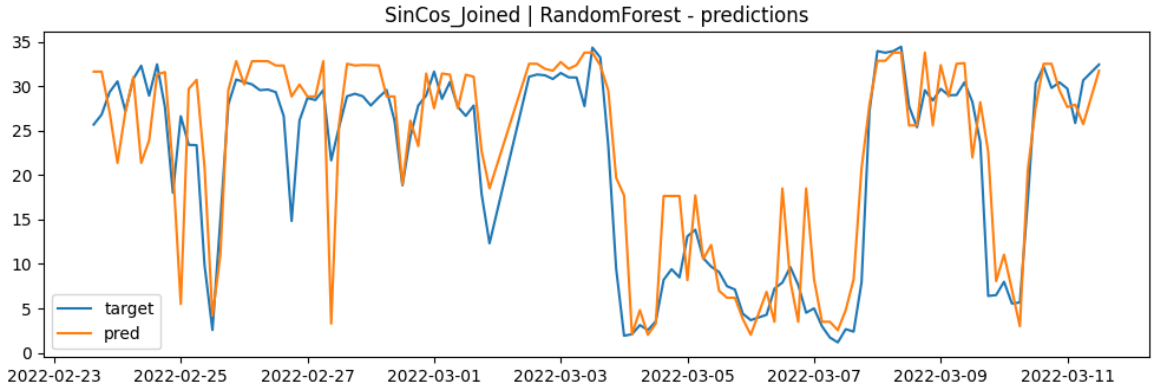

Figure 5: A plot showing the performance of the RandomForest model on the inner joined dataset

Compared to the Polynomial and GradientBoosting models, the Random Forest outperformed both. However, maybe changing he hyperparameters of these models could have impacted the results.

3.2.2 The best encoding strategy

In general, both encoding strategies performed very similar, however, the SinCos encoding slightly outperformed the WindVector encoding.

Therefore, separating direction and speed with the SinCos encoding, and not combining them, helps the model capture directional features more reliably.

3.2.3 The best approach to merging the datasets

From my results, I observe that the best way of merging the two datasets (power generation and weather forecast) is to use a simple inner join compared to the other two approaches (upsampling and ASOF merge), as it avoids unrealistic assumptions about the data and preserves only the timestamps that accurately match each other. Although the inner join produces fewer datapoints, it still retains reliable data that reflects the real behaviour, which is important for building a good model.

In contrast, the upsampling and ASOF merge approaches resulted in a large increase in dataset size, but they did so by making assumptions that do not hold in practice. Upsampling assumes that

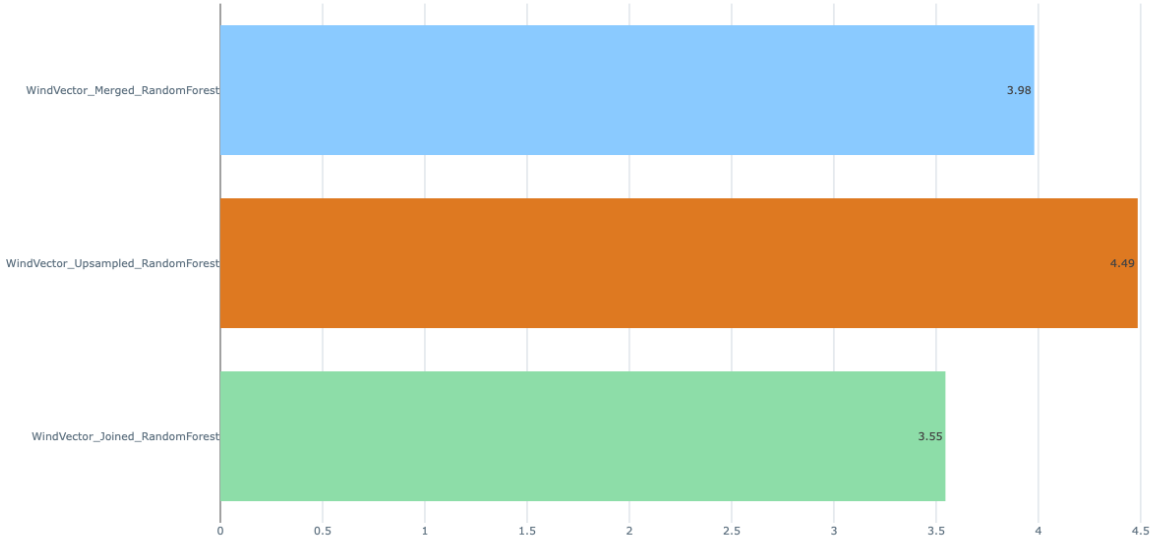

Figure 6: Best model for WindVector Encoding

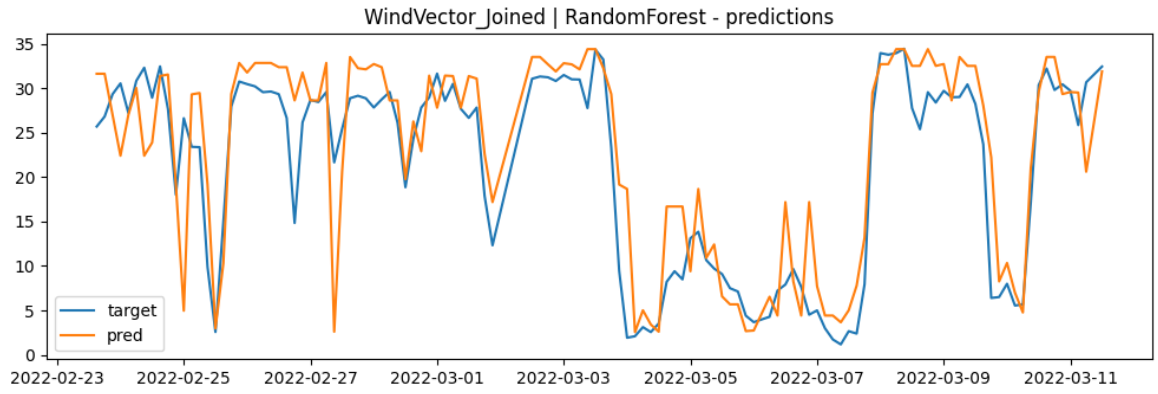

Figure 7: A plot showing the performance of the RandomForest model on the inner joined dataset

wind conditions remain constant between forecasts, which is an unrealistic simplification, and therefore ignores the natural behaviour of wind. Similarly, the ASOF merge assumes that power measurement can be merged to the nearest weather measurements, even when the timestamps are far apart, creating pairings that were never actually observed together. These assumptions introduce systematic biases and misleading patterns in the data.

These issues degrade the model's performance despite the larger dataset. This outcome reflects the principle of "garbage in, garbage out". A model trained on inaccurate data will inevitably produce unreliable results. For this reason, the inner join provides the cleanest and most reliable alignment approach for modelling the relationship between the weather conditions and power generation.

4 Interacting with my model

To interact with my WindVector Random Forest model you can run the following curl command to get a prediction

Listing 9: Curl command to get an prediction from my served model WindVector Randomforest model

```
curl -X POST http://20.251.168.216:5001/invocations \
  -H "Content-Type: application/json" \
```

```
  -d '{
    "dataframe_records": [
      {"Wind_x": -10.325278, "Wind_y": 4.27687}
    ]
  }'
```

This will return:

Listing 10: Result of the curl command

```
{"predictions": [20.595784119966222]}
```

A Appendix

Figure 8: MAE results using Linear Regression with the SinCos encoding

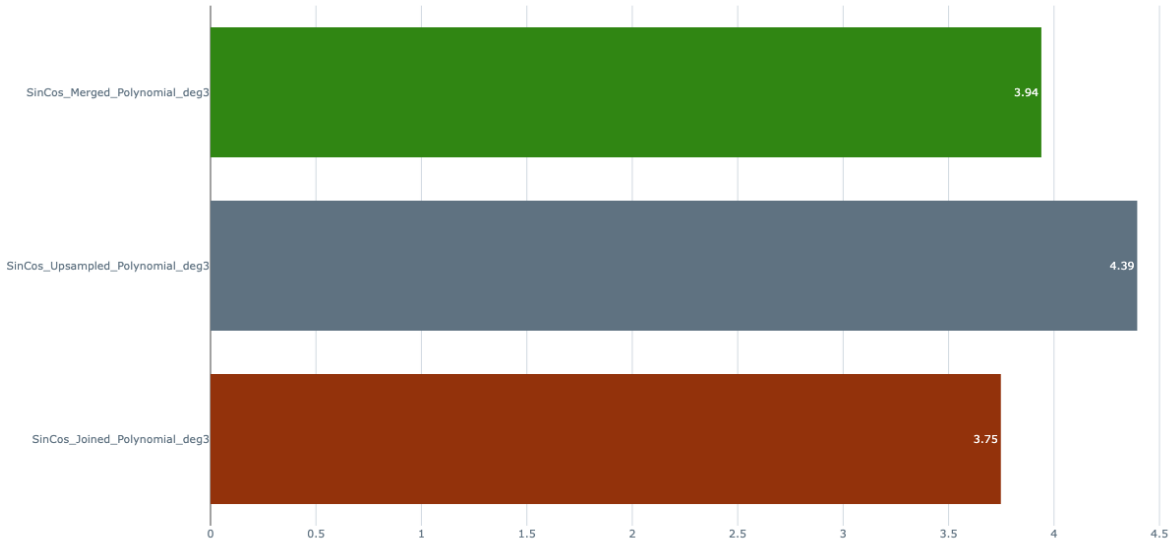

Figure 9: MAE results using Polynomial degree 3 with the SinCos encoding

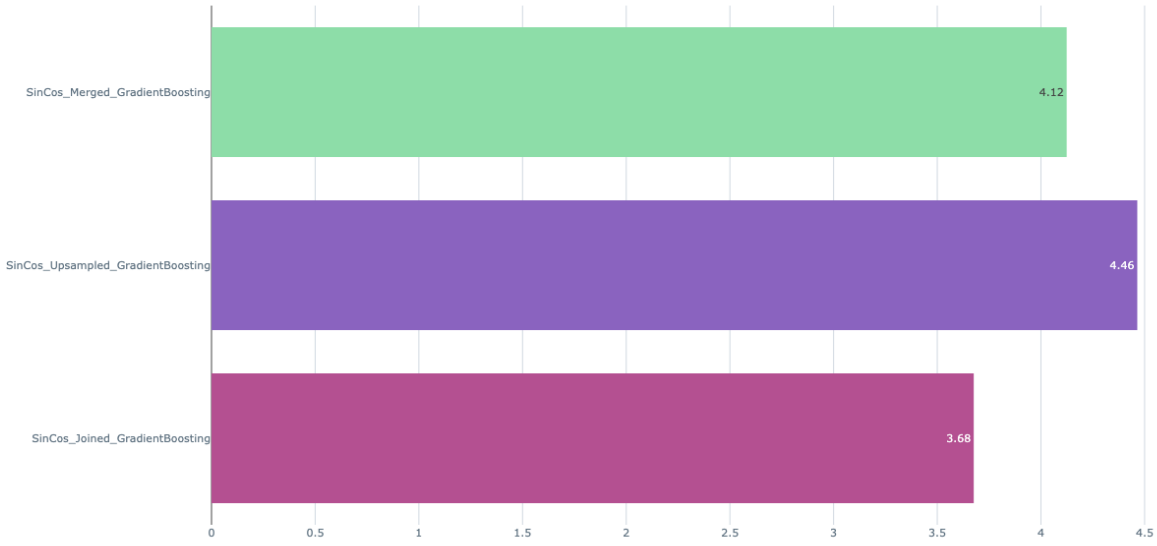

Figure 10: MAE results using GradientBoosting with the SinCos encoding

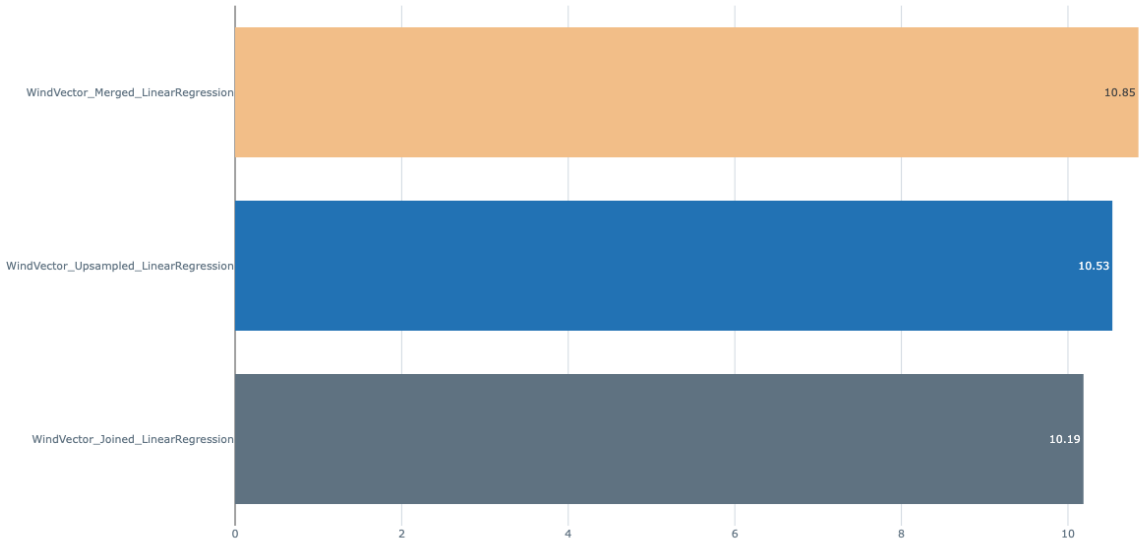

Figure 11: MAE results using Linear Regression with the WindVector encoding

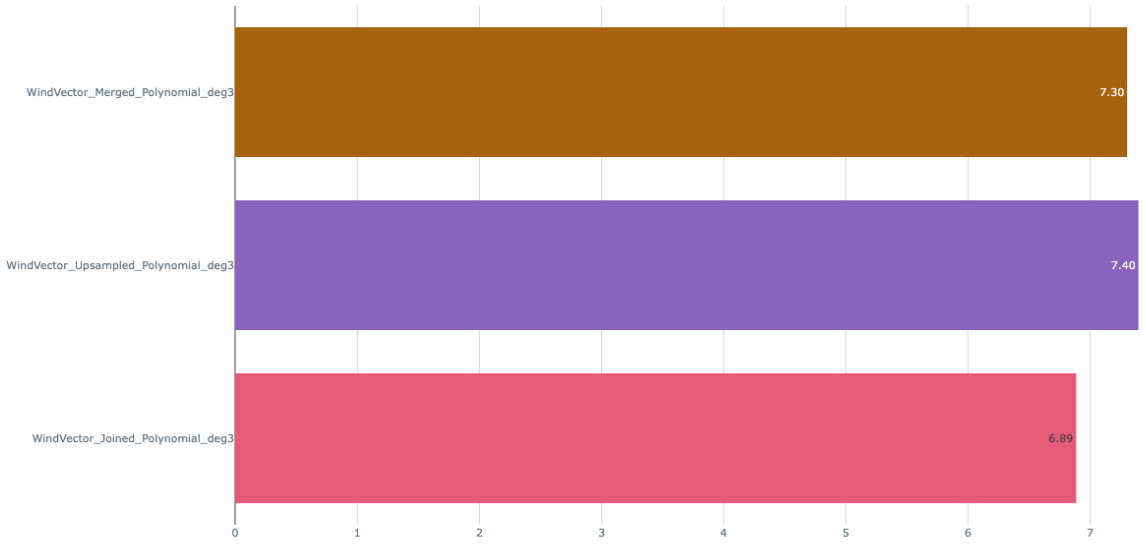

Figure 12: MAE results using Polynomial degree 3 with the WindVector encoding

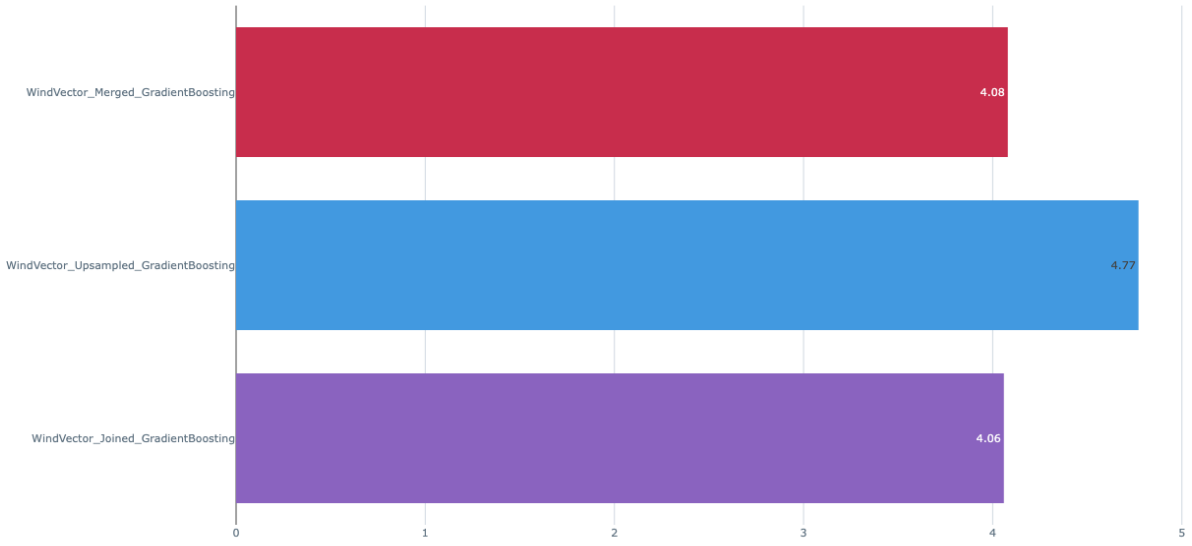

Figure 13: MAE results using GradientBoosting with the WindVector encoding