IT UNIVERSITY OF COPENHAGEN

Big Data Management – Assignment 2

Adam Hadou Temsamani (ahad@itu.dk)

Big Data Management
Course code: KSBIDMT1KU
Course manager: Zoi Kaoudi (zoka@itu.dk)

Github Repository: `github.itu.dk/Big-Data-Management-2025/ahad_a2`
Hand-in date: 5/12/2025

Contents

1 Introduction

I implemented an ML pipeline that predicts Orkney renewable generation (total MW) from UK Met Office wind forecasts. The report covers data ingestion/extraction, preprocessing and alignment, feature engineering, model training and selections, experiment tracking, and finally deployment. The implementation can be found in `Assignment_2.py`.

1.1 Data description

I have been given two tables. The **power** table contains one-minute frequency measurements with columns `Total` (MW), as well as other columns such as `AMN` and `Non-ANM` that are dropped.

The **weather** table contains 3-hour forecast timestamps and the columns `Speed` (m/s) and `Direction` (encoded as a cardinal string e.g. N or SW); `Source_time`, and `Lead_hours` are dropped due to being irrelevant to the model.

Both files (`power.csv` and `weather.csv`) are read, indexed by their `time` column to produce a time-series Pandas object. The following listing shows how data is extracted and cleaned:

```
power_df = read_csv_with_time_index( data/power.csv )
wind_df = read_csv_with_time_index( data/weather.csv )
cols_to_drop_power = [c for c in ['ANM','Non-ANM'] if c in power_df.
    columns]
cols_to_drop_wind = [c for c in ['Lead_hours','Source_time'] if c in
    wind_df.columns]
power_df.drop(columns=cols_to_drop_power, inplace=True, errors='ignore
    ')
wind_df.drop(columns=cols_to_drop_wind, inplace=True, errors='ignore')
```

2 Preprocessing and time alignment

The resolution of the two tables differs (1 minute vs 3 hours). To evaluate the effects of alignment on model accuracy I produce four merged datasets using the following strategies: *naive inner join*, *up-sampling*, *down-sampling*, and *merge-as-of*. After merging, rows with NaNs in the selected features or target dropped.

2.1 Naive Inner Join

The naive inner join requires exact timestamp matches, and therefore discards non-matching rows. As such *exact* temporal fidelity is preserved; as each forecast is guaranteed to match measured power at the given timestamp. However, by discarding non-matching rows, can substantially reduce training data and potentially also introduce selection bias towards given timestamps.

2.2 Up-sampling and Down-sampling

Up-sampling forward-fills 3-hours forecasts to the 1-minute resolution; matching the power frequency. However, it assumes that forecasts hold between updates and introduce potentially stale forecasts.

Down-sampling reduces noise at the cost of resolution by computing 3-hour averages to align with forecasts. However, this can remove peaks and temporal structure that may be useful for prediction.

2.3 Merge-as-of

Finally, Merge-as-of pairs each power timestamp with the nearest forecast within a tolerance. For the assignment 90-minutes have been chose, however this can be changed and might potentially not be optimal; Having too high of tolerance yields bias and too low reducing matches.

```
 # Naive join
power_weather_naive_join = power_df.join(wind_df, how= inner )

# Upsampling, Downsampling, Merge_asof
wind_upsample_dfs = wind_df.resample('1min').ffill()
wind_upsample_join = wind_upsample_dfs.join(power_df, how= inner )

power_downsample_dfs = power_df.resample('3h').mean()
power_downsample_join = power_downsample_dfs.join(wind_df, how= inner
    )

merged_dfs = pd.merge_asof(power_df, wind_df, on='time', direction='
    nearest', tolerance=pd.Timedelta('90min'))
merged_dfs = merged_dfs.set_index('time')
```

3 Feature Engineering

3.1 Wind Direction as a Feature

Wind direction may be an important predictor power output. I do not explicitly evaluate whether this is the case. However, the most direct way to determinate this is to conduct controlled experiments; train and evaluate models with and without any direction-based features while keeping preprocessing, splits and hyperparameter identical.

If metrics such MAE/RMSE improve when direction is included, then we can conclude that it contributes useful information.

3.2 Encoding Wind Direction

The current way wind direction is encoded is categorical. The cardinal labels give no notion of angular proximity, which creates a 0/360 discontinuity, if converted directly to degree. Therefore, I encode the wind direction with circular information by: converting to degrees, then radians, and finally compute `Sin` and `Cos`.

Circular encoding `sin/cos` preserves angular proximity, so that the model treats nearby directions as similar.

Additionally, using `Sin/Cos` I encode wind direction to wind vectors: `u` and `v` by using the `Speed` (magnitude) and `Sin/Cos`. Which binds magnitude and direction into orthogonal components, which potentially may make this interaction explicit for the model and reduce a dimension (a feature); which may or may not help the model's accuracy.
Both of these encoding are evaluated as separate feature sets `SinCos` and `WindVector`. For `SinCos` wind speed is included as the total power is primarily a function of wind magnitude (which can be described as the poower curve).

The following code shows how I construct the two feature sets:

```
    def add_sin_cos(df):
    df = df.copy()
    df['Radians'] = np.deg2rad(df['Degree'])
    df['Sin'] = np.sin(df['Radians'])
    df['Cos'] = np.cos(df['Radians'])
    df = df.drop(columns=['Radians', 'Direction', 'Degree'], errors='
        ignore')
    return df

def wind_vector(df):
    df = add_sin_cos(df.copy())
    df['u'] = df['Speed'] * df['Sin']
    df['v'] = df['Speed'] * df['Cos']
    df = df.drop(columns=['Sin', 'Cos', 'Radians', 'Direction', '
        Degree'], errors='ignore')
    return df
```

The two feature sets are:

```
# Features and Label
sincos_X = ['Cos', 'Sin', 'Speed'] # SinCos Features
windvector_X = ['u', 'v', 'Speed'] # WindVector Features
y = ['Total']
```

3.3 Normalization

Blindly normalizing the input features such as Wind Direction would generally be incorrect. Wind Direction is encoded between 0-360 degrees, where 0 and 360 are equivalent. Scaling

Min-Max (0-1) would imply that 0 and 1 are far apart, even though they are not.

Standardization/Normalization is applied using StandardScaler within the pipeline. Transforming the features to have mean of 0 and standard deviation of 1. Models sensitive to magnitude would be biased towards features with larger values This is necessary for these models, as there would otherwise be scale imbalance. Sin/Cos features are bound between -1 and 1, while Wind Speed has much higher values (20-30+).

Scale imbalance is detrimental. The model will penalize coefficients to prevent overfitting, preferring smaller coefficients. Without normalization, features with large values (Wind Speed) would naturally have a smaller coefficient, and features like Sin/Cos would require large coefficients to impact prediction.

Trade-Off. Normalization (Min-Max scaling) would compress data if outliers exist. Extreme wind speeds are rare outliers, but critical to detect for accurate power forecasting. This would compress majority of "normal" wind data into a tiny numerical value. Standardization does not impose a fixed upper or lower limit, but is rather scale on variance. Making it more robust to outliers. The choice of which of these two use would depend on the existence of outliers.

Using MLFlow, we can empirically evaluate the impact of these preprocessing steps by tracking and comparing the same model but with different feature scalings.

4 Modeling

I evaluate a range of linear and nonlinear regressors: *LinearRegressor*, *Ridge*, *RandomForestRegressor*, *GradientBoostingRegressor*, *XGBRegressor*, and polynomial feature pipelines (degrees 2 and 3, with and without Ridge).

Using a broad set of models is intention as MLflow makes it straightforward to test multiple candidates, and the additionally compute cost is small compared to the benefit of identifying a superior model.

4.1 Pipelines

Concepts

- **Pipeline**. A sequence of data transformers with an optional final predictor. Allows us to apply a list of transformers to pre-process the data, and if desired a final predictor. The purpose of the pipeline is assemble several steps that can be cross-validated, while setting different parameters.

- **Data Leakage**. Information from the test accidentally influenced the training process; making the model look better than it actually is.

- **Standardization (Z-Score Scaling)**. A technique where features are rescaled to have mean of 0 and a standard deviation of 1. This is distinct from Normalization (Min-Max

scaling). This is critical for algorithms sensitive to the magnitude of input features (Ridge Regression)

To ensure modularity and prevent data leakage; Scikit-Learn Pipeline was used. The pipeline consists of two sequential steps; Preprocessing and Estimator.

Pipeline Structure

- **Transformation**: A `StandardScaler` is applied to standardize all features. This transformation centers the data and scales it to a unit variance. Standardization is necessary for some of the models used in the assignment.
  - Ridge Regression, which prevents overfitting by penalizing the size of model coefficients (weights). If features have different scales, the feature with the smaller scale would require a much larger coefficient to have the same effect on the output.
- **Estimation**: The data is passed directly to the regressor (LinearRegression, RandomForestRegressor)

The primary reason for using the Pipeline API is to prevent data leakage by enforcing a strict separation between training and testing data. It also allows us to tune parameters for preprocessing steps and the model. We can easily swap out the final estimator (changing from Linear Regression to XGBoost).

4.2 Training and testing

Training and testing are split chronologically, reserving the final 20% for the test set. The chronological split preserved temporal order and the underlying serial dependence; ensuring the model is only evaluated on dat that occur after the training samples. This is important as the data is *time-series* based, avoids information leakage, and prevents past training inflating model accuracy.

4.3 Hyperparameters

I do not explicitly test specific hyperparameters for the different models; instead I use conservative defaults, one can argue that I should have spent more time exploring this to see how it affects each *regressor*.

For linear models, I utilized Scikit-Learn defaults as a performance baseline. However, for tree based ensembles (Random Forest, Gradient Boosting, XGBoost) and polynomial, we set the following values: `n_estimators=100`, `max_depth=10` for tree ensembles, and polynomial degrees `2` and `3` for the polynomial pipelines.

Tree Ensembles: Wind power is inherently noisy, unbounded tres would grow until they memorize the noise, leading to overfitting (poor generalization). Increasing the depth would likely decrease training error, but if too large of value is set would increase testing error.

The number of estimators controls the size of the ensemble (number of decision trees). Lowering this number could lead to unstable predictions, while increasing could lead to diminishing returns in accuracy (while increasing computational cost).

Polynomial Degrees: Degrees 2 and 3 were chosen to allow the models to capture non-linearity, without risk of overfitting.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, shuffle=False)

... # pipeline logic

# Fit model on data
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_test)
```

```
def candidate_models():
    models = {
        'linear': LinearRegression(),
        'rf': RandomForestRegressor(n_estimators=100, max_depth=10),
        'gbr': GradientBoostingRegressor(n_estimators=100, max_depth
            =10),
        'xgb': XGBRegressor(tree_method='hist', n_estimators=100,
            max_depth=10),
...
        'polynomial_lr_deg2_with_ridge': Pipeline([
            ('poly', PolynomialFeatures(degree=3, include_bias=False))
                ,
            ('scaler', StandardScaler()),
            ('regressor', Ridge(alpha=1.0))   # recommended to help
                regularize polynomial features
        ]),
        'polynomial_lr_deg2_without_ridge': Pipeline([
            ('poly', PolynomialFeatures(degree=2, include_bias=False))
                ,
            ('scaler', StandardScaler()),
            ('regressor', LinearRegression())
        ])
    }
    return models
```

5 MLflow experiments

All runs are tracked under the MLflow experiment `Assignment 2 - WindPower Prediction`. For each run I log parameters (model type, hyperparameters, dataset variant, and feature set),

metrics (MAE, MSE, RMSE, 2) and artifacts (plots and serialized model).

5.1 Tracking

In MLFlow, we logged specific parameters and metrics:

Dataset. We logged the dataset name (eg.g Inner_Join_SinCos vs. Upsampled_WindVector) to empirically determine which data joining strategy and feature engineering technique yielded better results. This helps isolate whether alignment or feature engineering has a greater impact on performance.

Model. We log the mode class ("Linear", "XGB"). Enabling us to group runs by algorithm family, to access whether more complex non-linear model outperform linear models.

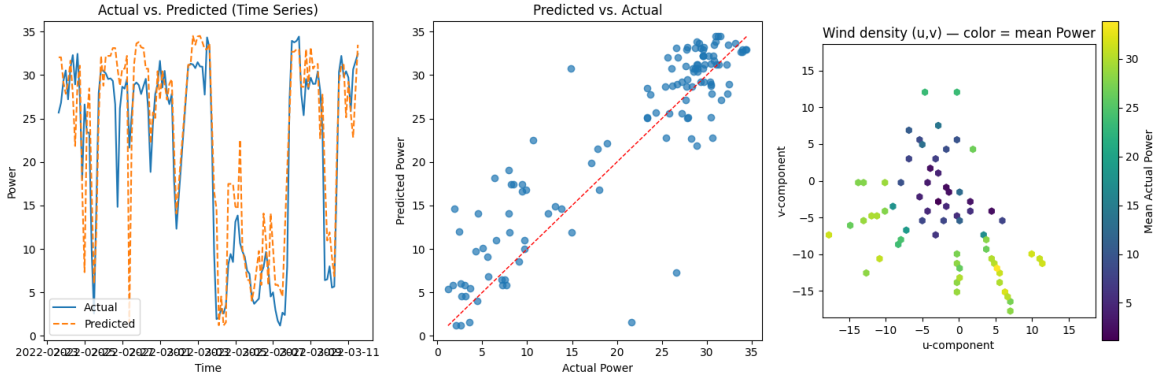
Metrics (MAE, RMSE, MSE, R2). MAE is tracked as it measures the average error, is less sensitive to outliers. RMSE and MSE specifically penalizes larger predicting errors (extreme deviations). R2 was tracked to measure goodness-of-fit.

The pipeline select the best model across both variants by minimum MAE and registers it in MLflow.

```
    def register_model(model_info, model_name):
        Save the model to a local path.
    path = f runs:/{model_info['run_id']}/{model_info['dataset']}_{
        model_info['model']}_model
    mlflow.register_model(path, model_name)
```

6 Evaluation

Performance is measured with MAE, MSE, RMSE and R2. After prediction, the script computes these metrics and stores them in MLflow. Visualization of the model are saved per run: a time-series plot of actual vs predicted, a predicted-vs-actual scatter and a wind-direction visual (polar histrogram for `SinCos` or hexbin of `u/v` colored by mean power for WindVector). The best model is selected by minimum test MAE. These metrics are logged so that runs can be compared in MLflow's UI.




```
# Log metrics
mlflow.log_metric( test_mae , test_mae)
mlflow.log_metric( test_mse , test_mse)
mlflow.log_metric( test_r2 , test_r2)
mlflow.log_metric( test_rmse , test_rmse)

# Log model
signature = infer_signature(X_train, pipeline.predict(X_train))
mlflow.sklearn.log_model(pipeline, f {dataset_name}_{model_name}_model
    , signature=signature)
```

7 MLOps/Deployment

Experiments are executed via MLflow, and models are served through MLflow's REST server on an Azure VM. Example commands:

```
mlflow run . --env-manager=local --experiment-name  Assignment 2 -
   WindPower Prediction

mlflow models serve
-m  /home/ahad/.../mlruns/.../artifacts/Inner_Join_SinCos_linear_model

-h 0.0.0.0 -p 5001 --no-conda
```

The MLflow server is run inside `tmux` to perist after logout. Additionally on Azure, the VM's Network Security has been allowed to permit inbound traffic to the serving port (5001). The server exposes `/invocations` and accepts JSON in following format to make a prediction;

```
curl http://<VM-IP>:5001/invocations
-H  Content-Type: application/json
```

```
-d '{ dataframe_split :{ columns :[ Sin , Cos , Speed ] , data
    :[[0.3826834,0.9238795,7.15264],[0.0,1.0,3.12928]]}}'
```

8 Discussion

The experiments show that models trained on the `Inner_Join` datasets (exact timestamp alignment) produced the strongest performance. The top five runs ranked by MAE all belonged to `Inner_Join` variants using either `WindVector` or `SinCos` encodings. It is important to know that this ranking is based solely on MAE; ordering could differ under RMSE, MSE, or R2 because they weigh error differently. A secondary check using RMSE produced a somewhat near ordering to MAE with only small rank changes.

Table 1: Top 5 models (ranked by MAE). The names have been shortened for readability of table,

Rank	Model	MAE	MSE	RMSE	R^2
1	Inner_Join_WV_polyn_lr_deg3_no_ridge	3.60	25.69	5.07	0.78
2	Inner_Join_SinCos_rf	3.65	28.16	5.31	0.76
3	Inner_Join_WV_rf	3.69	28.82	5.37	0.75
4	Inner_Join_WV_poly_lr_deg3_w_ridge	3.72	27.28	5.22	0.77
5	Inner_Join_WV_polynomial_lr_deg2_w_ridge	3.72	27.28	5.22	0.77

8.1 Data Alignment

The strong performance of `Inner_Join` is potentially due to it maintaining the highest fidelity/resolution mapping between forecast (weather) and power measurements: as when a forecast and a power observation share the exact timestamp, the input most closely represents the conditions that produced that observation. And as such helping the model capture the strongest direct relationship. Other strategies introduce distortions briefly mentioned earlier: Up-sampling can pair power with stale forecasts and down-sampling averages away structure; merge-as-of with wide tolerance (of 90 minutes) can match forecasts that are temporally distant.

8.2 Best models

The repeated appearance of polynomial models among the top-performing runs suggests that the underlying relationship between wind direction and power output is nonlinear, consistent with the turbine power curve. However, high-degree polynomial risk overfitting, especially with limited data.

Random Forests also performs well here as it is an ensemble of decisions trees, which allows it to capture non-linear feature interactions. It is worth noting that very deep forests can also overfit, and such can benefit from exploring a different set of hyperparameters.

8.3 Feature Engineering/Encoding effect on performance

Feature encoding plays an important role. Wind direction is inherently circular, and categorical encoding lose the relationship beteen directions. Both `WindVector` representation and `SinCos` preserve angular information and avoid the 0/360 discontinuity. The `WindVector` encoding goes an extra step combining wind speed with direction, which can make directional effects more interpretable (and remove a nonlinear interaction), making it easier for the model to learn. This may explain why several of the top-performing models in the experiments rely on `WindVector` features.

8.4 MLflow benefits

Systematic Experiment Tracking. Without systematic tracking, comparing the different numerous combinations of data preprocessing strategies and models is error-prone and manual. MLFlow allows to programmatically log every run's parameters and metrics. Using its dashboard enables us to objectively compare the different strategies and algorithms to select the optimal candidate empirically.

Reproducibility. Using MLFlow Projects allows us to encapsulate our code, data, and dependencies into a more portable format. This ensures that a stakeholder can clone the repository and execute `mlflow run` to replicate the exact same training environment and results.

Easier Deployment. Moving a model from a training script to a serving endpoint often requires additional work. MLFlow alleviates this by allowing us to save the trained estimator, and deploy our best-performing model as a REST API using the `mlflow models serve`.

Model Management. By using MLFlow Model Registry to manage the life-cycle of our best models. We can check if a newly trained model outperforms an already existing model, we can register it. Additionally, if a model is better, it is registers as a new model, allowing to roll back (versioning).

8.5 Deployment

Deploying a trained model makes it possible to use it beyond the development environment without re-running or training the model; making it a callable artifact that other system can integrate with. Additionally, deployment improves reproducibility: when served through MLflow, each deployed model corresponds to a specific run ID, parameters, dataset version, and environment snapshot. Ensuring anyone querying the model is guaranteed to use the exact same model weights and preprocessing steps.

8.6 Training window

The pipeline uses the previous 90 days of data. I do not explicitly evaluate whether this is the optimal choice, but in practice this should be validated as it can have an effect on the models performance.

A straightforward approach is to train the same models with the same hyperparameters across different window sizes (e.g., 30, 60, 90, 180 days) and compare their MAE/RMSE. The point at which performance stabilizes or begins to degrade is a sensible training window length. Additionally, more data does not necessarily improve accuracy. Longer windows may introduce outdated or misleading patterns (different seasons), especially in a time dependent dataset. Shorter windows adapt better to recent behavior but can increase variance.

9 Conclusion

The pipeline is effective at exploring and comparing alignment and encoding strategies. The results indicate that exact temporal alignment and wind direction encoding help with prediction.

However, there are several ways the pipeline can be improve. Currently model training is limited to a single epoch and only performed through a coarse exploration of hyperparameters; as such the reported results may not reflect each model's fully optimized performance. The testing could be coupled with k-fold cross-validation and early stopping.

Additionally, when performance is similar, it is better to choose simpler and regularized models as these tend to be more robust on unseen data (future predictions).

10 Potential Exam Questions

10.1 Explain what parameters you logged and why. Show your results

I log three categories of parameters.
First, I log a dataset identifier (e.g, Inner_Join_SinCos or Upsampled_Windvector). This was done to isolate the effects of my preprocessing choice (merging and feature engineering), allowing me to compare whether sampling the weather data outperformed a naive inner join, and whether encoding the data as a Wind Vector, or instead of Wind Direction as Sin/Cos to preserve the cyclic nature of wind direction.
Second, I log the model class (e.g., linear, random forest, xgb, etc.). This enables grouping runs by algorithm family to assess whether non-linear models outperform my linear baseline. Additionally, specific hyperparameters such as number of estimators and max depth, allowing me determine if increasing model complexity leads to overfitting.
Thirdly, I track the following metrics; MAE, MSE, RMSE. MAE (Mean Absolute Error) was prioritized as my primary metric, as it is easily interpretable, and is not overly sensitive to outliers compared to other metrics. MSE and RMSE were also logged to identify and penalize large prediction failures.

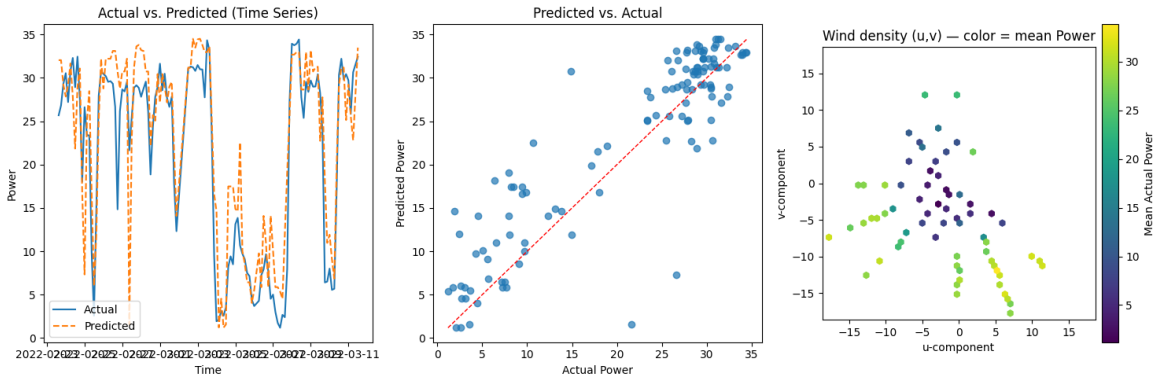
The abundance of parameters (spanning eight dataset variants), multiple models, and various hyperparameter configurations, would be infeasible to track manually. But by using

MLflow, helps us track experiments and sort them by our metrics, and much easier identify which variant was the best among all the candidates.

These are my results and findings:

Table 2: Top 5 models (ranked by MAE). The names have been shortened for readability of table,

Rank	Model	MAE	MSE	RMSE	R^2
1	Inner_Join_WV_polyn_lr_deg3_no_ridge	3.60	25.69	5.07	0.78
2	Inner_Join_SinCos_rf	3.65	28.16	5.31	0.76
3	Inner_Join_WV_rf	3.69	28.82	5.37	0.75
4	Inner_Join_WV_poly_lr_deg3_w_ridge	3.72	27.28	5.22	0.77
5	Inner_Join_WV_polynomial_lr_deg2_w_ridge	3.72	27.28	5.22	0.77



Left Plot: Time Series Comparison (Actual vs. Predicted) What it shows: This plot visualizes temporal fidelity—how well your model follows the changes in the target variable over time. It typically has Time on the x-axis and the Target Variable (e.g., Power, Price, Temperature) on the y-axis.

What you are looking for: You want the dashed line (Prediction) to overlay the solid line (Actual) as closely as possible

Middle Plot: Parity Plot (Predicted vs. Actual) It plots the Ground Truth (Actual) on the x-axis against the Model's Output (Predicted) on the y-axis. A red diagonal line usually represents "perfect prediction

Good Signal: The dots form a tight, narrow "cigar" shape along the diagonal. This means the error is low and consistent.

Cloud/Scatter: If the dots are spread far from the line (a "fat" cloud), the model has high variance (it is inconsistent).

Right Plot: Feature Space / Density Plot

it maps two physical dimensions (u and v wind components) on the x and y axes, using color to represent the target value (Power). Generally, this checks if the input features actually contain useful information.

if the top-right corner is bright yellow (High Power) and the center is dark purple (Low Power), it proves that these specific input features are strong predictors. The model can easily draw a line to separate them.

Confetti" or random noise. If yellow and purple dots are mixed together randomly, it means these features (u and v) do not clearly explain the target variable.

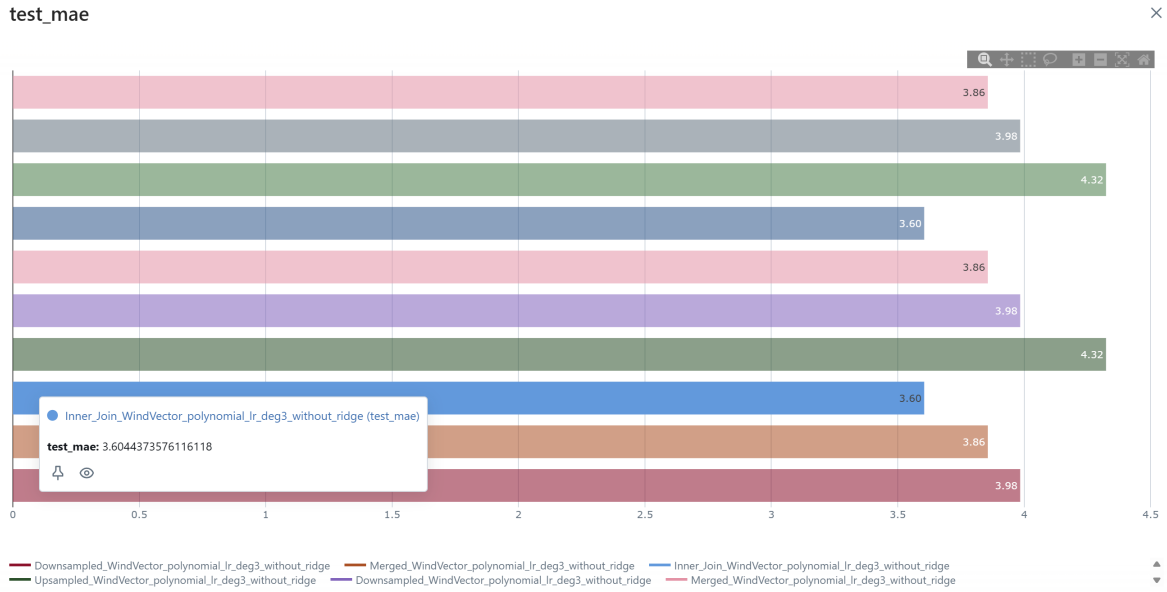


Figure 1: **Best Model Performance:** Polynomial Regression (Degree 3, No Ridge) trained on the Wind Vector dataset.

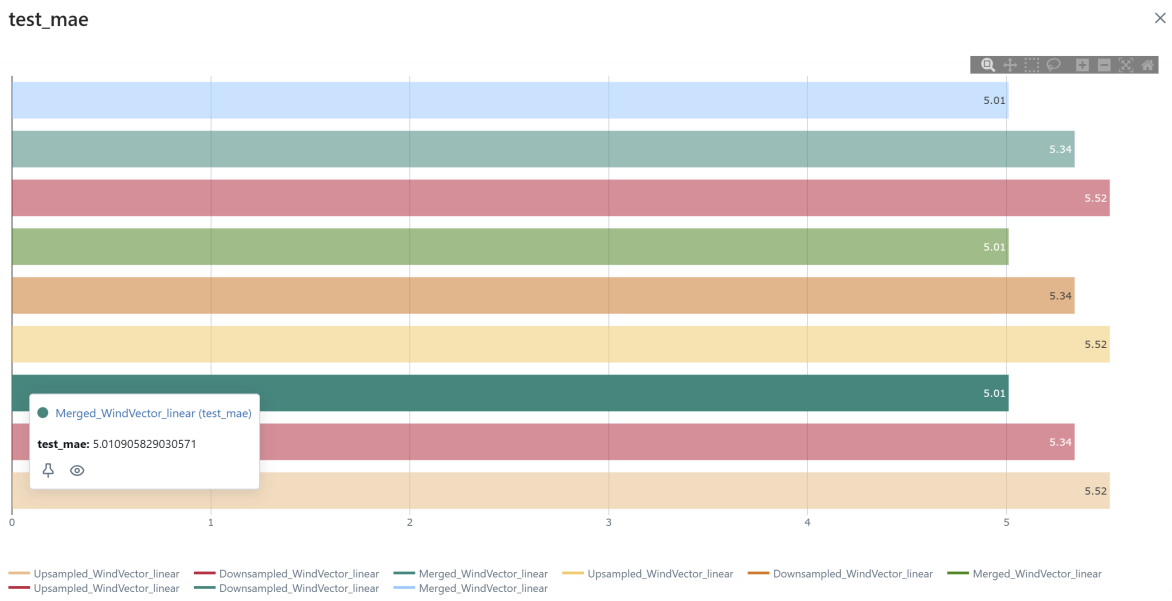


Figure 2: **Baseline Model Performance:** Linear Regression trained on the Wind Vector dataset.

10.2 Which steps does your sklearn pipeline include? Show a pipeline in code and briefly explain each step

```
   models =  candidate_models()
    for model_name, model in models.items():
    # If the candidate model is already a Pipeline, use it as-is.
        if isinstance(model, Pipeline):
            pipeline = model
        else:
            pipeline = Pipeline([
                ( scaler , StandardScaler()), # Transformations
                ( regressor , model) # Estimator
            ])

    def candidate_models():
    models = {
        'linear': LinearRegression(),
        'rf': RandomForestRegressor(n_estimators=100, max_depth=10),
        'gbr': GradientBoostingRegressor(n_estimators=100, max_depth
           =10),
...
        'polynomial_lr_deg2_without_ridge': Pipeline([
```

```
            ('poly', PolynomialFeatures(degree=2, include_bias=False))
                ,
            ('scaler', StandardScaler()),
            ('regressor', LinearRegression())
        ])
    }
    return models

    ...
    pipeline.fit(X_train, y_train)
    y_pred = pipeline.predict(X_test)
```

The pipeline consists of two stages: **Transformations** and **Estimator**.

- **Transformation.** These steps are mainly for pre-processing. In all my pipelines, a StandardScaler is applied to standardize the features (zero mean, unit variance), which ensures input features with magnitudes (such as Wind Speed compared to Cos/Sin) do not dominate the learning process. For polynomial variants, an additional Polynomial Features transformation is done (before Standardization), by creating new features by raising existing ones to a power or combining them. Allowing us to capture complex, and non-linear relationships in data.
- **Estimator.** The final step is the regressor itself. This takes the transformation input features and fits the model to the target data. Done by calling .fit and .predict.

10.3 Show your results for 2 experiments (different models/the same model with different hyperparameters/different preprocessing steps).

These are my results:

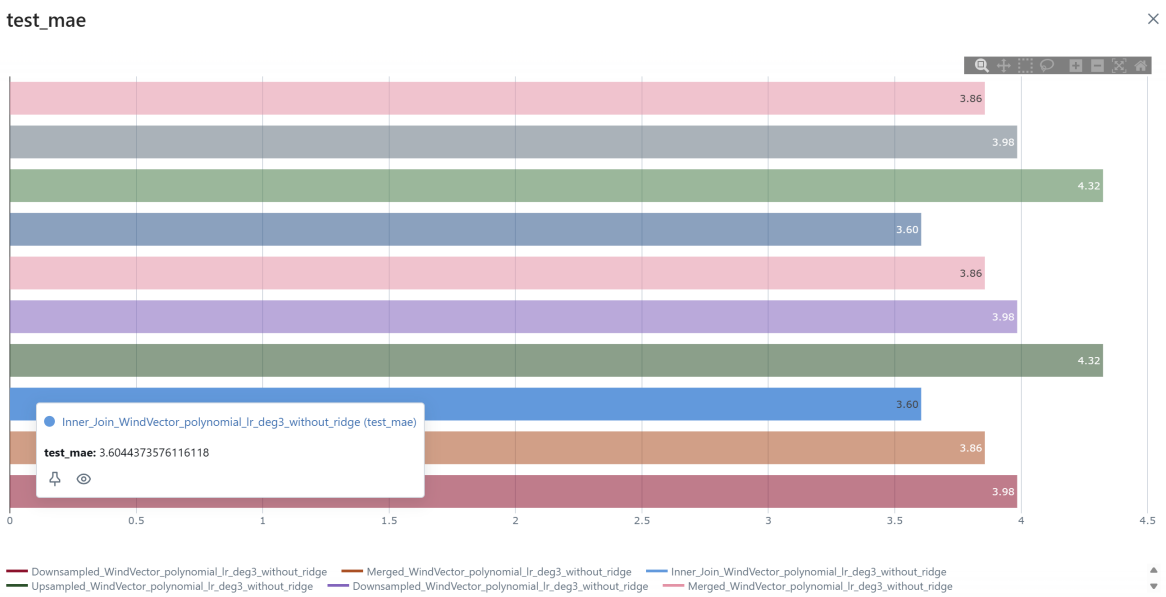


Figure 3: **Best Model Performance:** Polynomial Regression (Degree 3, No Ridge) trained on the Wind Vector dataset.

Figure 4: **Baseline Model Performance:** Linear Regression trained on the Wind Vector dataset.

Table 3: Top 5 models (ranked by MAE). The names have been shortened for readability of table,

Rank	Model	MAE	MSE	RMSE	R^2
1	Inner_Join_WV_polyn_lr_deg3_no_ridge	3.60	25.69	5.07	0.78
2	Inner_Join_SinCos_rf	3.65	28.16	5.31	0.76
3	Inner_Join_WV_rf	3.69	28.82	5.37	0.75
4	Inner_Join_WV_poly_lr_deg3_w_ridge	3.72	27.28	5.22	0.77
5	Inner_Join_WV_polynomial_lr_deg2_w_ridge	3.72	27.28	5.22	0.77

10.4 Imagine that you were developing a wind prediction model for each grid connected turbine in Denmark (6-7000 in total). The turbines have independent datasets and diverse scales and conditions. How would you set up this experiment in a distributed manner using MLflow?

I would deploy a centralized Tracking Server on a cloud (VM) (Azure VM as an example) configured with a database backend to handle the concurrency of potentially 7000 logging streams (metadata such as metrics, parameters, and tags), while offloading heavy files to a remote artifact store like Azure Blob Storage (actual model artifacts).

I would use Apache Spark to parallelize the computation across a cluster, where each worker node trains a model and logs to the central server.

To ensure reproducibility, I would package the training code using an MLproject file, which defines the environment and entry points for the distributed runs. Post-training, we can use the MLflow Registry, a centralized model store, to manage the life-cycle and versioning of the thousands of models. Finally, I would deploy the model (using MLflow) on VM with a public IP address, where by serving it makes predictions available to end users via REST API.

One could potentially also do a model for each area, as the wind conditions are different. It would be paramount to then log the turbine id, and region then using MLflow `tags`.

10.5 Why is it problematic to feed raw angular data (like Wind Direction) into a regression model, and how did you resolve this in your feature engineering?

The current wat wind direction is cardinal labels. These give no notion of angular proximity i.e, a 0/360 discontinuity (if they are converted to degrees). Instead I encode the wind direction with circular information. This is done by converting to degrees, radians, and finally computing Sin/Cos. Additionally, using this Sin/Cos encoding, I also encode win direction to Wind Vectors, by multiplying Speed (wind magnitude) with Sin/Cos. This reduces a dimension (a feature), which potentially may or may not help the model's accuracy.

I evaluate both of these as two separate feature sets.

10.6 In time-series analysis involving multiple sources with different frequencies, how does the choice of merging strategy affect data quality and model bias?

I compared four strategies to handle the frequency mismatch between weather data (3 hour) and power data (1 minute).

Naive Inner Join: The naive inner join requires exact timestamp matches, and therefore discards non-matching rows. As such *exact* temporal fidelity is preserved; as each forecast is guaranteed to match measured power at the given timestamp. However, by discarding non-matching rows, can substantially reduce training data and potentially also introduce selection bias towards given timestamps.

Up-sampling and Down-sampling: Up-sampling forward-fills 3-hours forecasts to the 1-minute resolution; matching the power frequency. However, it assumes that forecasts hold between updates and introduce potentially stale forecasts.

Down-sampling reduces noise at the cost of resolution by computing 3-hour averages to align with forecasts. However, this can remove peaks and temporal structure that may be useful for prediction.

Merge-as-of: Finally, Merge-as-of pairs each power timestamp with the nearest forecast within a tolerance. For the assignment 90-minutes have been chose, however this can be changed and might potentially not be optimal; Having too high of tolerance yields bias and too low reducing matches.

10.7 Which steps (preprocessing, retraining, evaluation) does your pipeline include?

My pipeline includes:

Preprocessing: This involves loading the data, applying a merging strategy (Inner Join, Up-sampling etc.) to align timestamps (1 minute vs. 3 hours), and performing feature engineering (creating Wind Vector or Sin/Cos features). It also includes a StandardScaler step within my Scikit-Learn pipeline to standardize features.

Retraining: My code iterates through a dictionary of models (Linear, Polynomial, XGBoost, Random Forest). For each of these it calls .fit() on the training split. (I am using a 80/20 split).

Evaluation: The trained models predict on the unseen test split. I calculate metrics (MAE, RMSE, MSE) and log them, along with parameters and tags.

10.8 How do you handle missing data?

```
2022-02-22 00:00:00+00:00,SSE,1,1645480800,9.83488
2022-02-22 03:00:00+00:00,SSE,1,1645491600,16.98752
2022-02-22 06:00:00+00:00,WSW,1,1645502400,15.19936
2022-02-22 09:00:00+00:00,W,1,1645513200,15.19936
2022-02-22 21:00:00+00:00,WSW,1,1645556400,15.19936
```

It depends on the merging strategy. While my naive join handled missing data by simply dropping unmatched rows, the up-sampling, and merge-as-of strategy handled filling gaps by using either forward fill, or finding the nearest neighbor. Using time-based interpolation may have helped the model. Merge-as-of based on the example data above would produce NaN values. Therefore I implemented a final cleaning step before training where I explicitly drop NaN values in feature and target column.

```
# Drop rows missing any required column
    needed_cols = features + label
    df_clean = df.dropna(subset=needed_cols).copy()
```