Highly Integrated System Project

Assignment 4

Aaliya Javed

Zain Hanif

November 16, 2024

Ensemble Kalman Filter (EnKF) Description

The Ensemble Kalman Filter (EnKF) is a recursive data assimilation method used to estimate the state of dynamic systems (such as weather or ocean currents) by combining predictions from a model with real-world observations. Instead of directly calculating the error covariance, it uses an ensemble (or group) of model states to approximate it, making the approach computationally efficient for large, nonlinear systems.

By adjusting each ensemble member with observational data at each time step, the EnKF continuously refines the system's state estimate, allowing for more accurate predictions over time.

Key Differences Between EnKF and KF

- **Error Covariance Representation:**
  - **KF:** Uses a single covariance matrix to represent state uncertainty, which can be computationally expensive for high-dimensional systems.
  - **EnKF:** Approximates the error covariance using an ensemble of state estimates, making it more efficient for large systems.
- **Handling Nonlinearity:**
  - **KF:** Assumes linearity in the system. For nonlinear systems, the Extended Kalman Filter (EKF) is used, requiring linearization of the model.
  - **EnKF:** Directly handles nonlinear models by using an ensemble, avoiding the need for explicit linearization.
- **Computational Efficiency:**
  - **KF:** Can become computationally expensive for high-dimensional systems due to the need to update a large covariance matrix.
  - **EnKF:** More computationally efficient, especially for large, complex systems, as it uses an ensemble instead of a full covariance matrix.

Ensemble Kalman Filter (EnKF) Steps

Forecast Step

In the forecast step, each ensemble member (representing a possible state of the system) is propagated forward in time using the system's model dynamics.

Ensemble Propagation

Each ensemble member x_k^i is updated using the model forecast function f:

$$x_{k+1}^i = f(x_k^i) + \eta_k^i$$

where η_k^i is a random noise term representing model uncertainty, and i represents the ensemble index.

Ensemble Mean and Covariance

The ensemble mean and covariance matrix are calculated as:

$$\bar{x}_{k+1} = \frac{1}{N}\sum_{i=1}^{N} x_{k+1}^i$$

$$P_{k+1} = \frac{1}{N-1}\sum_{i=1}^{N}(x_{k+1}^i - \bar{x}_{k+1})(x_{k+1}^i - \bar{x}_{k+1})^T$$

where N is the number of ensemble members.

Analysis (Update) Step

In the analysis step, the forecasted ensemble is corrected by incorporating the observations, leading to an updated state estimate.

Calculate the Kalman Gain

The Kalman Gain K is computed to weigh the forecast error and observational error:

$$K = P_{k+1}H^T(HP_{k+1}H^T + R)^{-1}$$

where H is the observation operator matrix, and R is the observation error covariance matrix.

Update Each Ensemble Member

Each ensemble member is updated using the Kalman Gain and the difference between the observed data y_k and the model prediction Hx_{k+1}^i:

$$x_{k+1}^i = x_{k+1}^i + K(y_k + \epsilon_k^i - Hx_{k+1}^i)$$

where ϵ_k^i is a random perturbation added to the observational error.

These steps are repeated for each time step, allowing the EnKF to continuously refine the system's state estimate based on new observations.

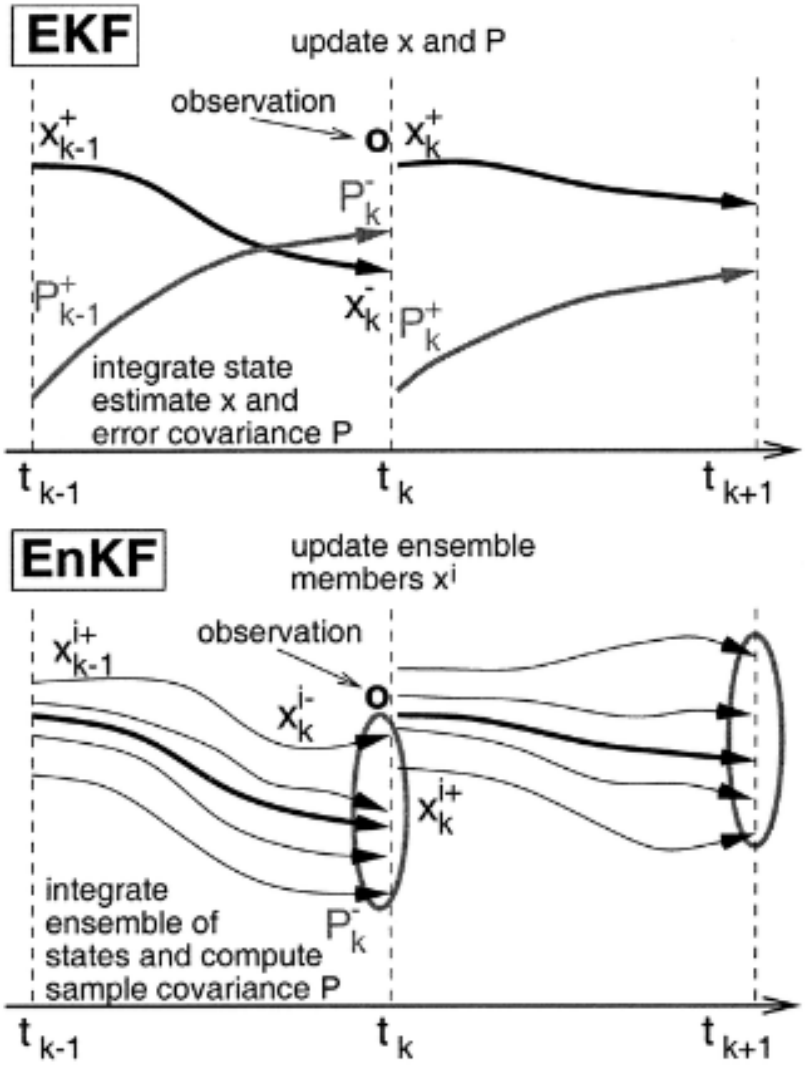

Figure 1: Difference Between EKF and EnKf

Real-Life Applications of Ensemble Kalman Filter (EnKF)

Weather Forecasting

Challenge: Weather prediction involves millions of variables, such as temperature, wind speed, and pressure, spread across a global grid. Standard Kalman Filter cannot handle the enormous data due to high computational requirements for storing and inverting large matrices.

Why EnKF? EnKF uses smaller ensembles (e.g., 100-500 members) to approximate the covariance matrix, making it computationally efficient for such large-scale systems.

Impact: EnKF is used by organizations like ECMWF (European Centre for Medium-Range Weather Forecasts) to improve weather prediction accuracy and efficiency.

Oil Reservoir Modeling

Challenge: Estimating subsurface parameters such as porosity and permeability from limited data is highly nonlinear and involves solving complex equations for fluid flow. Traditional methods struggle due to the high dimensionality and nonlinearity of the system.

Why EnKF? EnKF handles nonlinear dynamics effectively by using ensemble approximations, avoiding the need for linearization or direct inversion of large matrices.

Impact: EnKF is a standard tool in the oil and gas industry for real-time optimization and history matching of reservoir models.

Groundwater Monitoring

Challenge: Groundwater systems are complex and involve irregular observations of contaminant concentrations. KF and EKF cannot efficiently handle the high-dimensional and nonlinear nature of these systems.

Why EnKF? EnKF processes sparse data efficiently and updates groundwater models to provide accurate predictions.

Impact: EnKF helps in managing and mitigating groundwater contamination effectively, aiding environmental efforts.

Epidemiological Modeling

Challenge: In disease spread models like SEIR (Susceptible-Exposed-Infected-Recovered), nonlinearity and high uncertainty make standard KF impractical. EKF requires extensive computations for parameter estimation across regions.

Why EnKF? EnKF efficiently updates disease spread models using real-time data like daily case counts, enabling accurate predictions without extensive computations.

Impact: EnKF has been used for COVID-19 modeling to predict disease spread and evaluate interventions, providing actionable insights.

EnKF As Noise Handler

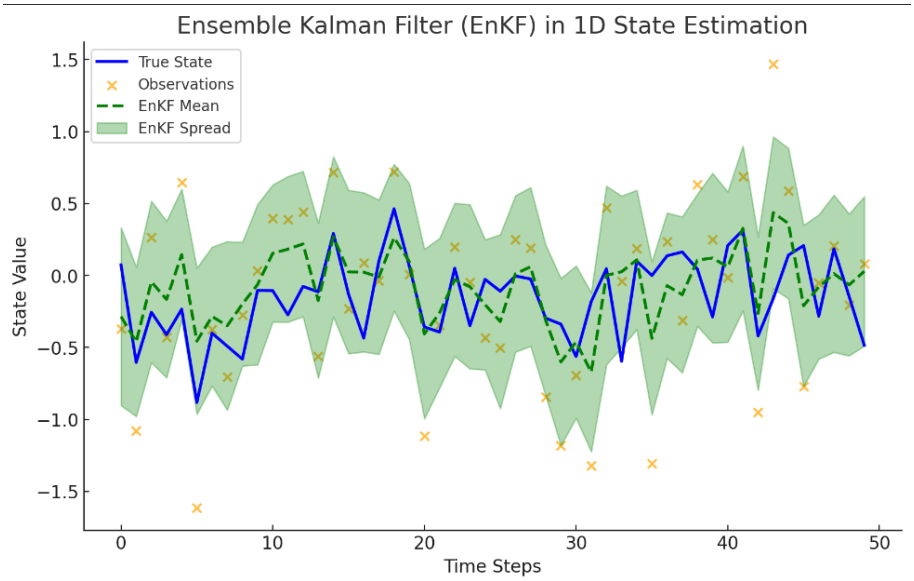

Figure 2: Noise Observation

Visualization of the Ensemble Kalman Filter (EnKF)

This visualization illustrates the Ensemble Kalman Filter (EnKF) in action:

- **True State (Blue Line):** The actual state trajectory over time.
- **Noisy Observations (Orange Points):** Observed data with noise added.
- **EnKF Mean (Green Dashed Line):** The mean of the ensemble, representing the estimated state.
- **EnKF Spread (Green Shaded Area):** The ensemble's spread (± 2 standard deviations), showing the uncertainty in the state estimate.

Advantages of EnKF

- **Scalability:** Efficient in handling high-dimensional systems.
- **Nonlinearity:** Does not require explicit linearization, unlike EKF.
- **Computational Efficiency:** Avoids direct inversion of large covariance matrices.
- **Flexibility:** Adaptable to a wide range of problems by tuning ensemble size and configurations.

Limitations of the Ensemble Kalman Filter (EnKF)

1. **Assumes Data Follows Normal Distribution**

   The EnKF usually assumes that errors in the model and observations follow a bell-curve (normal) distribution. If the errors are irregular or don't fit this assumption, the filter's results might not be as accurate.

2. **Needs a Large Number of Particles**

   For the EnKF to work well, it often needs a large number of particles to represent the possible states accurately. Using a small number of particles can lead to errors, but using a large number requires more computing power, which can be costly.

3. **Risk of Losing Diversity Among Particles**

   Over time, the particles can become too similar to each other, reducing the filter's ability to represent different possibilities. When this happens, it becomes harder for the EnKF to accurately track the true system state.

The EnKF is valued for its ability to handle large and complex problems, adapt to new data, and account for uncertainties, but it requires careful setup and enough computational resources to avoid potential issues with accuracy and particle diversity.

Forecasting Time Series with Machine Learning Models

In the previous chapter, we started exploring machine learning as a tool to address the problem of time series forecasting. We introduced a few techniques such as time delay embedding and temporal embedding, which transform a time series forecasting problem into a classical regression problem within the machine learning paradigm.

In this chapter, we will delve deeper into these techniques and apply them practically using the London Smart Meters dataset, which we have been working with throughout this book.

In this chapter, we will cover the following topics:

- Training and predicting with machine learning models
- Generating single-step forecast baselines
- Standardized code to train and evaluate machine learning models
- Training and predicting for multiple households

Data Preprocessing Steps

Data preprocessing is a critical step in preparing raw data for analysis and modeling. The typical steps include:

1. **Data Cleaning:**
    - Handle missing values (e.g., imputation, removal, or interpolation).
    - Remove duplicate entries.
    - Identify and handle outliers.
2. **Feature Selection:**
    - Identify and retain relevant features for the analysis.
    - Remove redundant or irrelevant features.
3. **Data Transformation:**
    - Normalize or standardize numerical features to ensure uniform scaling.
    - Encode categorical variables using methods like one-hot encoding or label encoding.
4. **Time Series Specific Steps:**
    - Ensure the data is sorted in chronological order.
    - Create lag features or rolling statistics as necessary.
    - Handle seasonality by extracting seasonal components.
5. **Splitting the Dataset:**

- Divide the dataset into training, validation, and test sets.
- Use appropriate splitting techniques, such as time-based splits for time series data.

6. **Handling Imbalanced Data (if applicable):**
    - Apply resampling techniques (oversampling, undersampling) or synthetic data generation methods (e.g., SMOTE).

Training and Predicting with Machine Learning Models

In Chapter 5, we introduced the concept of supervised machine learning, where the goal is to approximate an ideal function using a model. This is expressed as:

$$\hat{y} = h(X, \phi), \quad \text{where } h \in \mathcal{H}$$

Here, X represents the input features, ϕ the model parameters, and h the approximation function from a family of functions $\mathcal{H}$. For example, in linear regression, $\mathcal{H}$ is the set of all functions generated by different coefficient values.

This chapter focuses on applying machine learning models for time series forecasting, rather than general machine learning. While many models exist, we will review a few commonly used and effective ones. Before exploring these models, we first establish baseline methods for comparison.

Generating Single-Step Forecast Baselines

In Chapter 4, we reviewed and generated baseline models, but these were multi-step forecasts. For machine learning models, we focus on single-step forecasts, where only one target is predicted at a time. To adapt baseline algorithms like ARIMA or ETS for single-step forecasting, we must fit the model on history, predict one step ahead, and refit iteratively for each subsequent step. This process is computationally intensive, especially for large datasets (e.g., 1,440 iterations for 30 days and 150 households).

Instead, we use simpler baseline methods like the naïve and seasonal naïve approaches, which can be implemented directly using pandas. These methods provide strong baselines for single-step forecasts. The results of these models are as

Naive and Seasonal Naive Forecasting

The **naive method** predicts the next value as the most recent observed value. It is simple yet highly effective for single-step forecasts.

The **seasonal naive method** predicts the next value based on the last observed value from the same season (e.g., the same hour of the previous day). This method is particularly effective for time series with strong seasonal patterns.

Both methods serve as strong baseline models for comparison in forecasting tasks.

Examples of Naïve and Seasonal Naïve Forecasting

Naïve Forecasting Example: Suppose the last observed value in the time series is $y_t = 50$. The naïve forecast for the next time step (y_{t+1}) is simply:

$$\hat{y}_{t+1} = y_t = 50$$

Seasonal Naïve Forecasting Example: Assume a daily seasonal pattern with observations every hour. If the observation at the same hour on the previous day was $y_{t-24} = 60$, the seasonal naïve forecast for the next time step is:

$$\hat{y}_{t+1} = y_{t-24} = 60$$

	MAE	MSE	meanMASE	Forecast Bias
Naive	0.086	0.045	1.050	0.02%
Seasonal Naive	0.122	0.072	1.487	4.07%

Figure 3: Aggregate metrics for a single-step baseline

To streamline model training and evaluation, we have followed a standardized structure, which we will review briefly to ensure you can follow the accompanying notebooks.

Standardized Code to Train and Evaluate Machine Learning Models

Training a machine learning model involves two primary components: the data and the model. To streamline and standardize the training pipeline, we define the following key configuration classes and a wrapper class:

1. **FeatureConfig:** Handles feature engineering, including the selection and transformation of input features.
2. **MissingValueConfig:** Manages missing data handling strategies, such as imputation or removal of incomplete records.
3. **ModelConfig:** Defines model-specific configurations, such as hyperparameters, model type, and evaluation metrics.
4. **MLForecast:** A wrapper class built over scikit-learn-style estimators (with `.fit` and `.predict` methods) to ensure a consistent and smooth training process.

These components work together to standardize the pipeline for training and evaluating machine learning models, enabling efficient experimentation and reproducibility.

FeatureConfig Overview

`FeatureConfig` is a Python dataclass that defines attributes and functions essential for processing data prior to feeding it into a machine learning model. It provides separate handling for continuous, categorical, and Boolean features. The key attributes include:

- **date:** The name of the column containing dates (mandatory).
- **target:** The name of the column containing the target variable (mandatory).
- **original_target:** The column with the untransformed target. Essential for metrics like MASE. Defaults to `target` if unspecified.
- **continuous_features:** A list of continuous features.
- **categorical_features:** A list of categorical features.
- **boolean_features:** A list of Boolean features (categorical with two unique values).
- **index_cols:** Columns used as the DataFrame index, typically datetime and/or unique identifiers.
- **exogenous_features:** Optional. A list of external features, such as temperature, that must be a subset of continuous, categorical, or Boolean features.

The class includes a validation mechanism and a method, `get_X_y`, with parameters for flexible preprocessing:

- **df:** A DataFrame containing all necessary columns, including the target.
- **categorical:** A flag to include or exclude categorical features.
- **exogenous:** A flag to include or exclude exogenous features.

The `FeatureConfig` class provides a convenient method, `get_X_y`, which returns a tuple: To use the class, initialize it with the feature names categorized into appropriate parameters (e.g., continuous, categorical, or Boolean features). Once initialized, you can pass a DataFrame to the `get_X_y` function to extract the features, target, and original target.

MissingValueConfig Overview

The `MissingValueConfig` class handles missing values in the dataset. In time series data, missing values can arise during feature engineering, such as when creating lag features, where early dates may lack sufficient data. While some machine learning models handle missing values naturally, others require preprocessing to address them.

`MissingValueConfig` is a Python dataclass that defines how to handle missing values based on different strategies. The key attributes of `MissingValueConfig` include:

- **bfill_columns:** A list of column names that require a backward fill strategy to handle missing values.

- **ffill_columns:** A list of column names that require a forward fill strategy to handle missing values. If a column appears in both `bfill_columns` and `ffill_columns`, it will be filled using backward fill first, followed by forward fill.
- **zero_fill_columns:** A list of column names that should be filled with zeros.

The default behavior for missing values is to use the column mean to fill any missing data, ensuring that no column is left unfilled if no specific strategy is defined. The class includes a method, textttimpute_missing_values, which accepts a DataFrame and fills the missing values based on the specified strategies.

- `bfill_columns` first
- `ffill_columns` second
- `zero_fill_columns` last

ModelConfig Overview

The `ModelConfig` class is a Python dataclass that holds configuration details related to the machine learning model, such as normalization, handling of missing values, and categorical encoding. The key attributes of `ModelConfig` are as follows:

- **model:** A mandatory parameter that defines the machine learning model, which must be a scikit-learn-style estimator.
- **name:** A string identifier for the model. If not provided, the class name of the model is used as the default.
- **normalize:** A Boolean flag that indicates whether to apply `StandardScaler` to the input features (default: False).
- **fill_missing:** A Boolean flag that indicates whether to fill missing values in the data before training (default: False). Some models can handle NaN values natively, while others cannot.
- **encode_categorical:** A Boolean flag that indicates whether categorical columns should be encoded as part of the fitting procedure (default: False). If set to False, categorical encoding should be handled separately before passing them as continuous features.
- **categorical_encoder:** If `encode_categorical` is set to True, this defines the scikit-learn-style encoder to use for encoding categorical features.

MLForecast Overview

The `MLForecast` class is a wrapper around a scikit-learn-style model. It utilizes the configurations we discussed earlier to streamline the training and prediction processes. The key parameters used during initialization of the `MLForecast` model are as follows:

- **model_config:** An instance of the `ModelConfig` class, which contains configuration details for the machine learning model.

- **feature_config:** An instance of the `FeatureConfig` class, which defines how the features are extracted and preprocessed.
- **missing_config:** An instance of the `MissingValueConfig` class, which specifies how missing values should be handled during training.
- **target_transformer:** An instance of the target transformers from `src.transforms`. This should support the methods `fit`, `transform`, and `inverse_transform`. It should return a `pandas.Series` with a datetime index to ensure proper handling during prediction. This transformer is also used during prediction to reverse the transformation of the target.

The `fit` Function

The `fit` function is akin to the scikit-learn `fit` method, with additional capabilities. It handles standardization, categorical encoding, and target transformations using the configuration classes. The parameters of the `fit` function are as follows:

- **X:** The pandas DataFrame containing the features to be used for training the model.
- **y:** The target variable, which can be a pandas DataFrame, pandas Series, or a numpy array.
- **is_transformed:** A Boolean flag indicating whether the target has already been transformed. If `True`, the target will not be transformed again, even if a `target_transformer` was initialized.

The `predict` Function

The `predict` function handles the inferencing process. It wraps around the `predict` function of the scikit-learn estimator, but it also includes additional functionality such as standardization, categorical encoding, and reversing the target transformation. The parameter for this function is as follows:

- **X:** The pandas DataFrame containing the features to be used in the model for prediction. The index of the DataFrame is passed along to the prediction process.

The `feature_importance` Function

The `feature_importance` function retrieves the feature importance from the trained model, if available. For linear models, it extracts the coefficients, and for tree-based models, it extracts the built-in feature importance. The function returns this information in a sorted pandas DataFrame.

Helper Functions for Evaluating Models

In addition to the standard configurations and wrapper classes, we have defined a couple of helper functions for evaluating different machine learning models. These functions are available in the accompanying notebook and assist in measuring the performance of the models.

```
def evaluate_model(
    model_config,
    feature_config,
    missing_config,
    train_features,
    train_target,
    test_features,
    test_target,
):
    ml_model = MLForecast(
        model_config=model_config,
        feature_config=feat_config,
        missing_config=missing_value_config,
    )
    ml_model.fit(train_features, train_target)
    y_pred = ml_model.predict(test_features)
    feat_df = ml_model.feature_importance()
    metrics = calculate_metrics(test_target, y_pred, model_
config.name, train_target)
    return y_pred, metrics, feat_df
```

Figure 4: Helper Functions for Evaluating Models

Decision Trees

Decision trees are a type of model that can make predictions by dividing data into different groups. These groups are based on the features (like characteristics or input variables) and each group gets its own simple prediction. For example, if you were predicting house prices based on the size of the house, a decision tree might break up the data into groups based on size, and then give an average price for each group.

The main reason decision trees are used is because they can model non-linear patterns that other models, like linear ones, can't capture. This flexibility makes them effective for both classification (predicting categories) and regression (predicting continuous values), allowing them to handle complex relationships in the data.

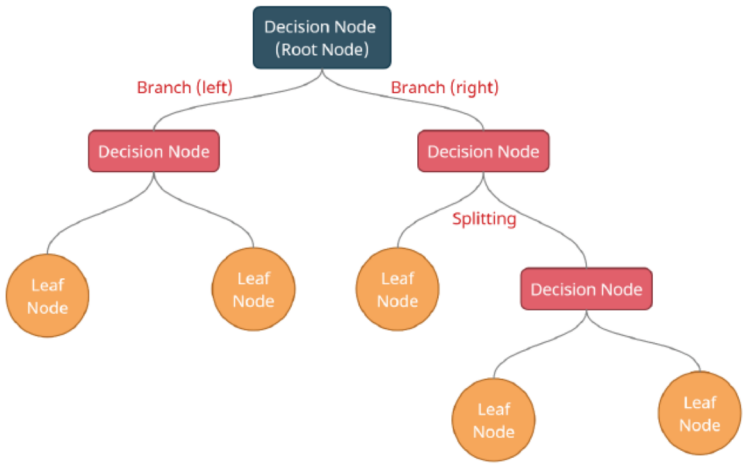

Figure 8.11 – Anatomy of a decision tree

Figure 5: Decision Tree

Splitting Criteria

The decision tree uses a *greedy algorithm* to decide how to split the data at each decision node. The algorithm selects the feature and split point that minimizes the error in predictions. Specifically, for regression tasks, the error is minimized by reducing the sum of squared errors for each partition.

$$\text{Error} = \sum_{i \in P_1} (Y_i - C_1)^2 + \sum_{i \in P_2} (Y_i - C_2)^2$$

where C_1 and C_2 represent the average values of the target variable Y in the respective partitions P_1 and P_2.

Stopping Criteria

As the decision tree grows by adding more splits, there is a risk of overfitting, where the model captures noise in the data, or underfitting, where it fails to capture important patterns. To avoid overfitting,

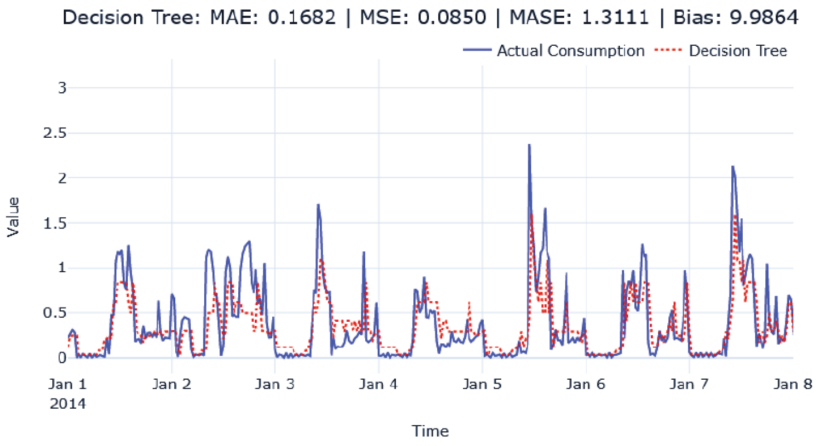

Figure 8.12 – Decision tree forecast

several strategies are used:

- Limiting the *maximum depth* of the tree.
- Setting a *minimum number of samples required* to split a node.

These stopping criteria help balance between overfitting and underfitting.

CART Algorithm

The *CART* (Classification and Regression Trees) algorithm is widely used to construct decision trees. It supports both *classification* and *regression* tasks. The method builds a tree by repeatedly splitting the data based on the best features, using the criteria mentioned above, and continues until stopping conditions are met.

Feature Importance

- Decision trees estimate *feature importance* using the **mean decrease in the loss function** during tree construction.
- In Scikit-learn, feature importance can be accessed via the `feature_importances_` attribute.
- **Best Practice:**
  - The default feature importance can be misleading for continuous features or high-cardinality categorical features.
  - Use *permutation importance* (`sklearn.inspection.permutation_importance`) for a more accurate assessment.

Overfitting and Underfitting

- **Overfitting (High Variance):**

Figure 8.13 – Feature importance of a decision tree (top 15)

- Decision trees are prone to overfitting as they can partition the feature space too finely, memorizing the training data.

- **Underfitting (High Bias):**
  - Insufficient splits or shallow trees result in underfitting, failing to capture patterns effectively.
- **Inability to Extrapolate:**
  - Decision trees cannot extrapolate beyond the range of training data. For example, if a feature f increases linearly with the target y, values of $f > f_{\max}$ will still yield predictions limited to $y \leq y_{\max}$.

Next Steps

- Ensemble models, which combine multiple decision trees, address the limitations of overfitting and improve overall performance.

Random Forest

- Random Forest is an **ensemble learning algorithm** that combines multiple decision trees to improve predictions.
- It uses the "wisdom of the crowd" concept, where combining many models (base learners) creates a stronger overall model.
- It modifies a technique called **bagging** by ensuring the decision trees are decorrelated, enhancing their combined performance.

How Does It Work?

1. **Tree Building:**

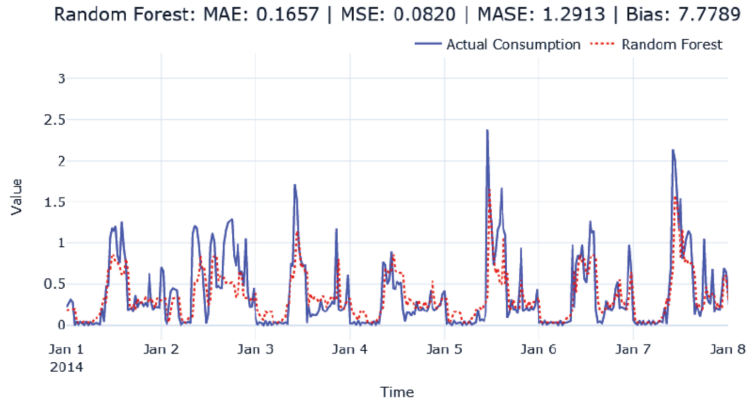

Figure 8.14 – Random Forest forecast

- Create M trees:
  (a) Draw a bootstrap sample (random sampling with replacement) from the dataset.
  (b) Randomly select f features from all features.
  (c) Choose the best split based only on these f features.
  (d) Repeat until the stopping criteria are met.
- This randomness reduces the correlation between trees and makes the ensemble more robust.

2. **Prediction:**
   - For regression: Average the predictions from all trees.
   - For classification: Use majority voting.

Key Advantages

- Reduces **overfitting** compared to a single decision tree.
- Handles variance well while maintaining low bias.
- Performs well on many datasets without much tuning.
- Feature importance can be estimated, similar to decision trees.

Key Limitations

- Inherits the **inability to extrapolate** from decision trees (e.g., it cannot predict outside the range of training data).
- Can be computationally slow for large datasets or many trees.

Hyperparameters

- Key parameters to tune:
  – **n_estimators**: Number of trees in the forest.

Figure 8.15 – Feature importance of a decision tree (top 15)

  - **max_features**: Number of features to consider for each split.

- Other decision tree parameters like `max_depth` also apply.

Best Practices

- Use tools like **Scikit-learn's** `RandomForestRegressor` for implementation.
- For larger datasets, consider alternatives like `XGBRFRegressor` from XGBoost for faster computations and additional benefits (e.g., handling missing values).

Performance

- Random Forest typically performs better than decision trees and is less prone to overfitting.
- Tuning hyperparameters can further enhance performance.
- Still, Random Forest may lag behind models like gradient boosting on certain datasets.

Random Forest builds an ensemble of decorrelated decision trees using randomness, making it a robust and widely used algorithm. It strikes a balance between variance and bias, but like decision trees, it cannot extrapolate beyond the training data.

Gradient Boosting Decision Trees

Gradient Boosting?

Gradient boosting is a machine learning technique that builds a powerful model by combining several weak models (usually decision trees). Unlike methods such as Random Forest, which create models in parallel, gradient boosting builds models sequentially, learning from the mistakes of the previous models.

How Does it Work?

1. **Start Simple:** Begin with a simple model that predicts a constant value (e.g., the average of the target values).
2. **Learn Residuals:** For each step:
   - Identify where the current model is making mistakes (residuals or pseudo-residuals).
   - Build a new decision tree to predict these residuals.
3. **Update the Model:** Add the new tree to the existing model, scaling its contribution using a parameter called the *learning rate*.
4. **Repeat:** Continue adding trees, each focusing on the errors of the combined model so far, until the desired number of trees is reached or improvement stops.

Key Features of Gradient Boosting

- **Learning Rate:** Controls how much each new tree contributes to the overall model. Smaller values improve accuracy but require more trees.
- **Number of Trees:** Increasing the number of trees can improve the model but requires careful monitoring to avoid overfitting. Early stopping is used to halt training when validation performance stops improving.
- **Regularization:** Techniques like row/column sampling and penalties (e.g., L1/L2 regularization) reduce overfitting and improve generalization.

Why is Gradient Boosting Powerful?

- It learns iteratively, correcting errors at each step.
- Regularization techniques, early stopping, and sampling make it robust and flexible for various tasks.

Feature Importance in Gradient Boosting

Gradient boosting provides insights into feature importance:

- **Split Importance:** Measures how often a feature is used to split nodes in the trees.
- **Gain Importance:** Evaluates the reduction in error attributed to a feature's split.

Popular Implementations

- **LightGBM:** Fast and efficient, supports categorical features, and handles missing data natively.
- **XGBoost:** Robust and widely used for a variety of tasks.
- **CatBoost:** Designed for categorical features, created by Yandex.
- **Scikit-learn:** Offers simpler implementations like `GradientBoostingRegressor`.

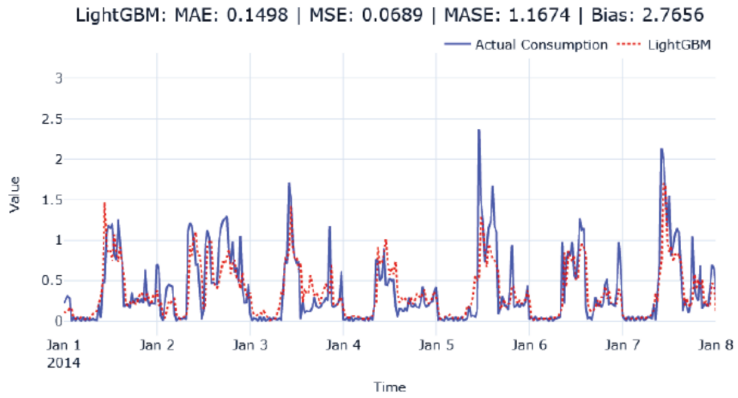

Figure 8.16 – LightGBM forecast

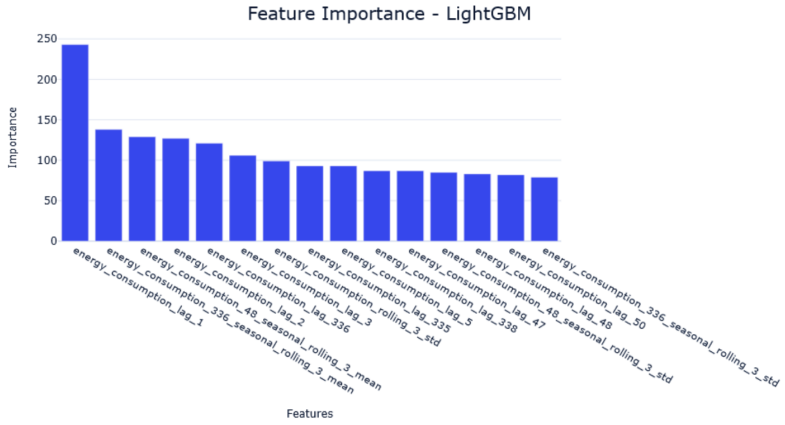

Figure 8.17 – Feature importance of LightGBM (top 15)

Gradient boosting builds trees sequentially, focusing on correcting errors at each step. It is powerful and versatile, with tools like LightGBM and XGBoost enhancing its speed and ease of use.

Gradient Boosted Decision Trees (GBDTs)

Strengths and Limitations of GBDTs

- **Strengths:**
  - GBDTs perform exceptionally well on tabular data and time series tasks, including regression.
  - They are commonly part of winning solutions in Kaggle competitions for time series forecasting.
- **Limitations:**
  - **Overfitting:** GBDTs are high-variance algorithms, requiring robust regularization to avoid overfitting.
  - **Training Time:** GBDTs take longer to train compared to Random Forests because of their sequential nature.
    - Random Forests train trees in parallel, whereas GBDTs must build trees sequentially.

Training and predicting for multiple households

We have picked a few models (`LassoCV`, `XGBRFRegressor`, and `LGBMRegressor`) that are doing better in terms of metrics, as well as runtime, to run on all the selected households in our validation dataset. The process is straightforward: loop over all the unique combinations, inner loop over the different models to run, and then train, predict, and evaluate. The code is available in the `01-Forecasting with ML.ipynb` notebook in `chapter08`, under the *Running an ML Forecast For All Consumers* heading. You can run the code and take a break because this is going to take a little less than an hour. The notebook also calculates the metrics and contains a summary table that will be ready for you when you're back. Let's look at the summary now:

Algorithm	MAE	MSE	meanMASE	Forecast Bias
Naive	0.0882	0.0450	1.1014	-0.00%
Seasonal Naive	0.1292	0.0777	1.6004	-1.00%
Lasso Regression	0.0802	0.0271	1.0052	-0.29%
XGB Random Forest	0.0808	0.0306	1.0177	-2.43%
LightGBM	0.0772	0.0275	0.9781	0.05%

Figure 8.19 – Aggregate metrics on all the households in the validation dataset

Here, we can see that even at the aggregated level, the different models we used perform as expected. The notebook also saves the predictions for the validation set on disk.

    * Tools like LightGBM enable parallelization using strategies like feature parallelism, data parallelism, and voting parallelism.

  - **Extrapolation:** GBDTs struggle to extrapolate trends in time series data, making detrending or switching to other models necessary.

    * Detrending can be done using techniques like `AutoStationaryTransformer`.

Performance Evaluation

- **Linear Models:** Performed well on MAE, MASE, and MSE metrics with slightly longer runtimes for regularized models.
- **Decision Trees:** Underperformed due to overfitting but can improve with better tuning.
- **Random Forests:** Performed better than decision trees.
  - XGBoost's Random Forest implementation was approximately six times faster than scikit-learn's.
- **LightGBM:** Achieved the best performance across all metrics and had the fastest runtime.

The GBDT models showed strong performance across most metrics. However, further evaluation on all selected households is necessary to generalize the results.

The aggregate metrics for the test dataset are as follows (from the notebook):

Algorithm	MAE	MSE	meanMASE	Forecast Bias
Naive	0.086	0.045	1.050	0.02%
Seasonal Naive	0.122	0.072	1.487	4.07%
Lasso Regression	0.077	0.026	0.946	0.99%
XGB Random Forest	0.078	0.030	0.966	-0.18%
LightGBM	0.075	0.027	0.914	2.57%

Figure 8.20 – Aggregate metrics on all the households in the test dataset

In *Chapter 6, Feature Engineering for Time Series Forecasting*, we used `AutoStationaryTransformer` on all the households and saved the transformed dataset.

Using AutoStationaryTransformer

The process is really similar to what we did earlier in this chapter, but with small changes. We read in the transformed targets and joined them to our regular dataset in such a way that the original target is named `energy_consumption` and the transformed target is named `energy_consumption_auto_stat`:

```
#Reading the missing value imputed and train test split data
train_df = pd.read_parquet(preprocessed/"block_0-7_train_
missing_imputed_feature_engg.parquet")
auto_stat_target = pd.read_parquet(preprocessed/"block_0-7_
train_auto_stat_target.parquet")
transformer_pipelines = joblib.load(preprocessed/"auto_
transformer_pipelines_train.pkl")
#Reading in validation as test
test_df = pd.read_parquet(preprocessed/"block_0-7_val_missing_
imputed_feature_engg.parquet")
# Joining the transformed target
train_df = train_df.set_index(['LCLid','timestamp']).join(auto_
stat_target).reset_index()
```

And while defining `FeatureConfig`, we used `energy_consumption_auto_stat` as `target` and `energy_consumption` as `original_target`.

Let's look at the summary metrics that were generated by these notebooks on the transformed data:

Algorithm	MAE	MSE	meanMASE	Forecast Bias
Naive	0.088	0.045	1.101	-0.00%
Seasonal Naive	0.129	0.078	1.600	-1.00%
Lasso Regression	0.080	0.027	1.005	-0.29%
XGB Random Forest	0.081	0.031	1.018	-2.43%
LightGBM	0.077	0.028	0.978	0.05%
Lasso Regression_auto_stat	0.083	0.030	1.055	-3.50%
XGB Random Forest_auto_stat	0.086	0.033	1.098	-8.33%
LightGBM_auto_stat	0.079	0.029	1.002	-4.41%

Figure 8.21 – Aggregate metrics on all the households with transformed targets in the validation dataset

The target transformed models are not performing as well as the original ones. This might be because the dataset doesn't have any strong trends.

Congratulations on making it through a very heavy and packed chapter full of theory as well as practice. We hope this has enhanced your understanding of machine learning and skills in applying these modern techniques to time series data.

Reference

- Breiman, L. *Random Forests*, Machine Learning 45, 5–32 (2001). https://doi.org/10.1023/A:1010933404324.
- Chen, Tianqi and Guestrin, Carlos. (2016). *XGBoost: A Scalable Tree Boosting System*. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16). Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785.
- Ke, Guolin et al. (2017). *LightGBM: A Highly Efficient Gradient Boosting Decision Tree*. Advances in Neural Information Processing Systems, pages 3149-3157. https://dl.acm.org/doi/pdf/10.5555/3294996.3295074.
- Prokhorenkova, Liudmila, Gusev, Gleb et al. (2018). *CatBoost: Unbiased Boosting with Categorical Features*. Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18). https://dl.acm.org/doi/abs/10.5555/3327757.3327770.
- GitHub repository: https://github.com/AaliyaJaved/TimeSeriesAnalysis_Group1.git