Intermittent demand forecasting with skforecast

Importing Libraries for Data Processing

```
import numpy as np
import pandas as pd
```

Importing Libraries for Visualisation*

```
import matplotlib.pyplot as plt
import plotly.graph_objects as go
import plotly.express as px
import plotly.io as pio
pio.templates.default = "seaborn"
```

Importing Libraries for Modeling

```
pip install skforecast
```

```
from lightgbm import LGBMRegressor
from sklearn.preprocessing import FunctionTransformer
from sklearn.metrics import mean_absolute_error
from skforecast.ForecasterAutoreg import ForecasterAutoreg
from skforecast.ForecasterAutoregCustom import ForecasterAutoregCustom
from skforecast.model_selection import grid_search_forecaster
from skforecast.model_selection import backtesting_forecaster
```

About the Data

The data used in this example represents the number of users who visited a store during its opening hours from Monday to Friday, between 7:00 and 20:00. Therefore, any predictions outside this period are not useful and can either be ignored or set to 0.

```
url = ('https://raw.githubusercontent.com/JoaquinAmatRodrigo/Estadistica-machine'
       '-learning-python/master/data/intermittent_demand.csv')
data = pd.read_csv(url, sep=',')
```

```
data['date_time'] = pd.to_datetime(data['date_time'], format='%Y-%m-%d %H:%M:%S') #formatting the date
data = data.set_index('date_time')
data = data.asfreq('H')
data = data.sort_index()
data.head(3)
```

	users
date_time	
2011-01-01 00:00:00	0.0
2011-01-01 01:00:00	0.0
2011-01-01 02:00:00	0.0

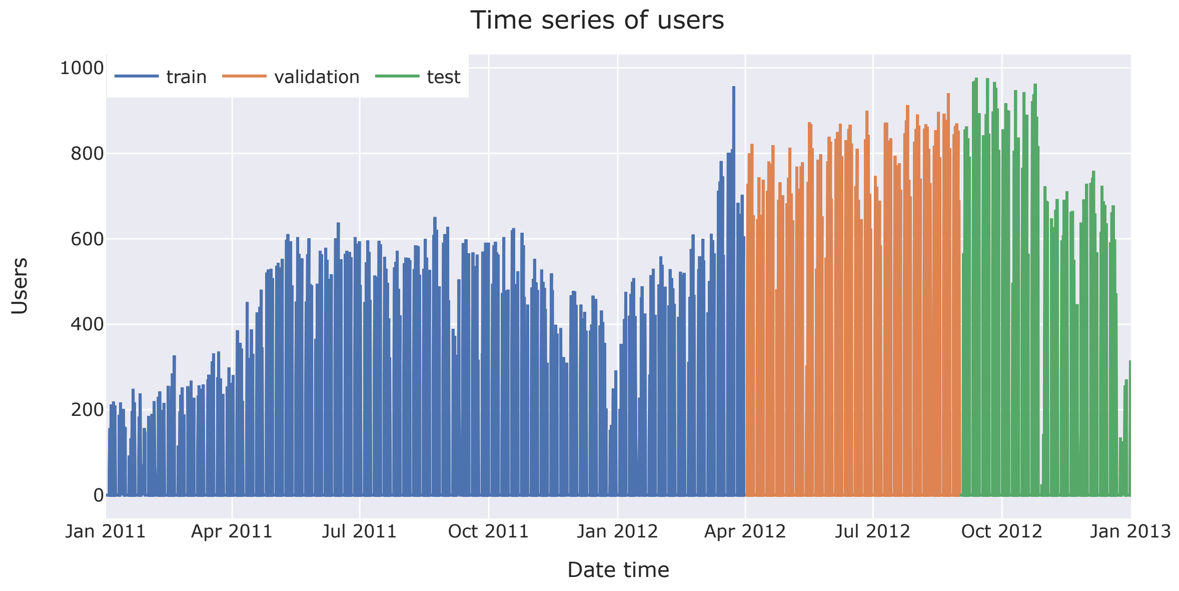
Splitting the data in train-test-val

```
end_train = '2012-03-31 23:59:00'
end_validation = '2012-08-31 23:59:00'
data_train = data.loc[: end_train, :]
data_val   = data.loc[end_train:end_validation, :]
data_test  = data.loc[end_validation:, :]

print(f"Dates train      : {data_train.index.min()} --- {data_train.index.max()}  (n={len(data_train)})")
print(f"Dates validation : {data_val.index.min()} --- {data_val.index.max()}  (n={len(data_val)})")
print(f"Dates test       : {data_test.index.min()} --- {data_test.index.max()}  (n={len(data_test)})")
```

```
Dates train      : 2011-01-01 00:00:00 --- 2012-03-31 23:00:00  (n=10944)
Dates validation : 2012-04-01 00:00:00 --- 2012-08-31 23:00:00  (n=3672)
Dates test       : 2012-09-01 00:00:00 --- 2012-12-31 23:00:00  (n=2928)
```

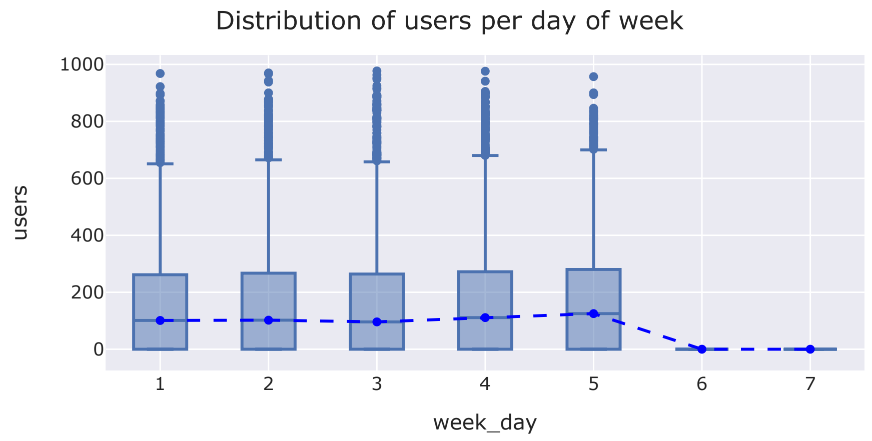
Plot time series

```
fig = go.Figure()
trace1 = go.Scatter(x=data_train.index, y=data_train['users'], name="train", mode="lines")
trace2 = go.Scatter(x=data_val.index, y=data_val['users'], name="validation", mode="lines")
trace3 = go.Scatter(x=data_test.index, y=data_test['users'], name="test", mode="lines")
fig.add_trace(trace1)
fig.add_trace(trace2)
fig.add_trace(trace3)
fig.update_layout(
    title="Time series of users",
    xaxis_title="Date time",
    yaxis_title="Users",
    width  = 800,
    height = 400,
    margin=dict(l=20, r=20, t=35, b=20),
    legend=dict(
        orientation="h",
        yanchor="top",
        y=1,
        xanchor="left",
        x=0.001
    )
)
fig.show()
```



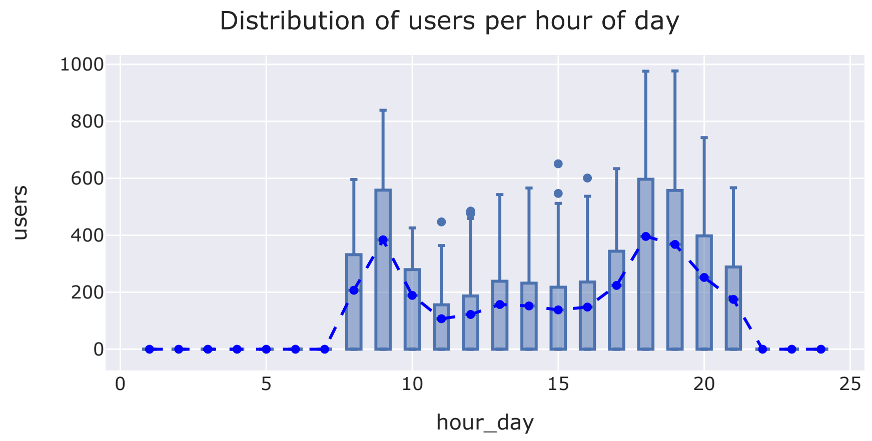
Boxplot for weekly seasonality

```
data['week_day'] = data.index.day_of_week + 1
fig = px.box(
        data,
        x="week_day",
        y="users",
        title = 'Distribution of users per day of week',
        width=600,
        height=300
     )
median_values = data.groupby('week_day')['users'].median()
fig.add_trace(
    go.Scatter(
        x=median_values.index,
        y=median_values.values,
        mode='lines+markers',
        line=dict(color='blue', dash='dash'),
        showlegend=False
    )
)
fig.update_layout(margin=dict(l=20, r=20, t=35, b=20))
fig.show()
```



Boxplot for daily seasonality

```
data['hour_day'] = data.index.hour + 1
fig = px.box(
        data,
        x="hour_day",
        y="users",
        title = 'Distribution of users per hour of day',
        width=600,
        height=300
     )
median_values = data.groupby('hour_day')['users'].median()
fig.add_trace(
    go.Scatter(
        x=median_values.index,
        y=median_values.values,
        mode='lines+markers',
        line=dict(color='blue', dash='dash'),
        showlegend=False
    )
)
fig.update_layout(margin=dict(l=20, r=20, t=35, b=20))
fig.show()
```



Define a custom metric to evaluate the model

To accurately evaluate the performance of the model, it is crucial to define a metric that closely reflects the business scenario in which the model will be used. Specifically, in this case, the model's performance should be optimized during weekdays from 9:00 to 20:00.

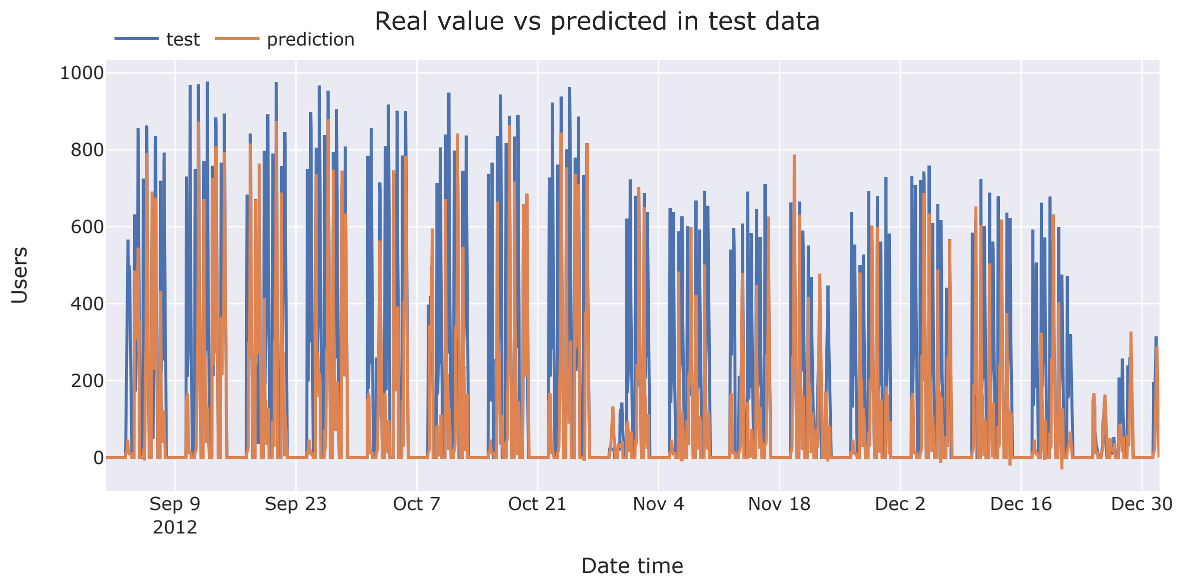
```
def custom_metric(y_true, y_pred):
    """
    Calculate the mean absolute error using only the predicted values for weekdays
    from 9:00 AM to 8:00 PM
    """

    day_of_week = y_true.index.day_of_week
    hour_of_day = y_true.index.hour
    mask = day_of_week.isin([0, 1, 2, 3, 4]) | ((hour_of_day > 7) | (hour_of_day < 20))
    metric = mean_absolute_error(y_true[mask], y_pred[mask])

    return metric
```

Forecasting

```
# Create forecaster
forecaster = ForecasterAutoreg(
                 regressor = LGBMRegressor(
                                 learning_rate = 0.1,
                                 max_depth     = 5,
                                 n_estimators  = 500,
                                 random_state  = 123,
                             ),
                 lags = 24
             )
forecaster
```

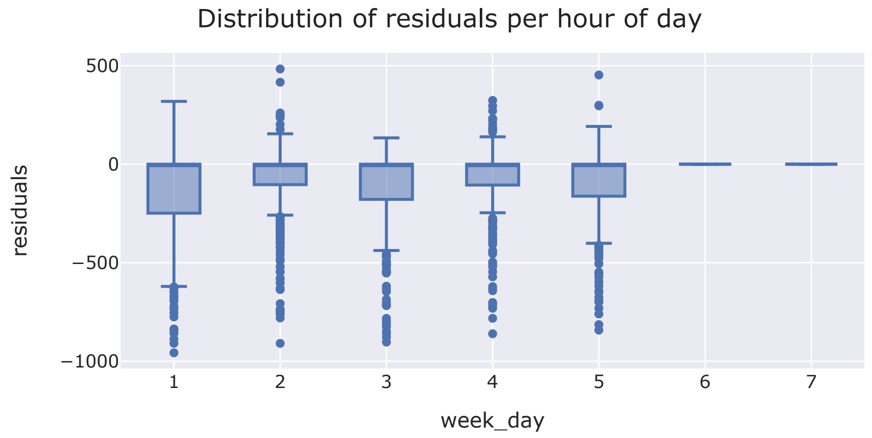
```
=================
ForecasterAutoreg
=================
Regressor: LGBMRegressor(max_depth=5, n_estimators=500, random_state=123)
Lags: [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
Transformer for y: None
Transformer for exog: None
Window size: 24
Weight function included: False
Differentiation order: None
Exogenous included: False
Type of exogenous variable: None
Exogenous variables names: None
Training range: None
Training index type: None
Training index frequency: None
Regressor parameters: {'boosting_type': 'gbdt', 'class_weight': None, 'colsample_bytree': 1.0, 'importance_type': 'split', 'learning_rate': 0.1, 'max_depth': 5,
'min_child_samples': 20, 'min_child_weight': 0.001, 'min_split_gain': 0.0, 'n_estimators': 500, 'n_jobs': None, 'num_leaves': 31, 'objective': None, 'random_state': 123,
'reg_alpha': 0.0, 'reg_lambda': 0.0, 'subsample': 1.0, 'subsample_for_bin': 200000, 'subsample_freq': 0}
fit_kwargs: {}
Creation date: 2024-04-16 16:31:24
Last fit date: None
Skforecast version: 0.11.0
Python version: 3.10.12
Forecaster id: None
```

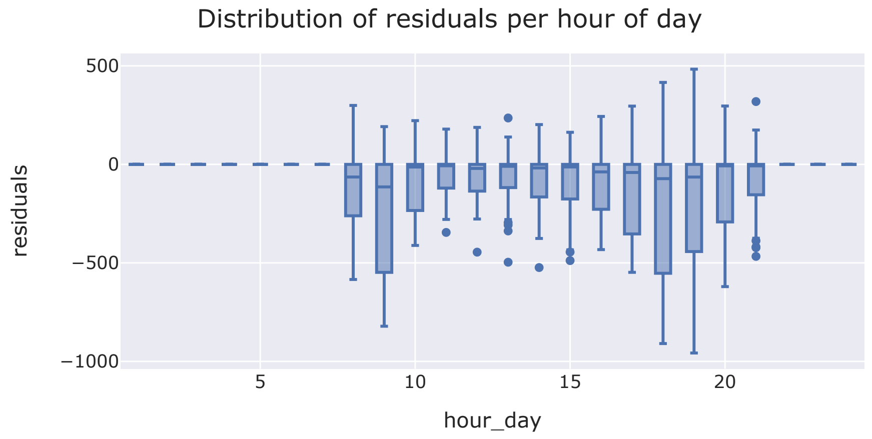
```
# Backtesting test period
metric, predictions = backtesting_forecaster(
                          forecaster         = forecaster,
                          y                  = data['users'],
                          initial_train_size = len(data.loc[:end_validation]),
                          fixed_train_size   = False,
                          steps              = 36,
                          refit              = False,
                          metric             = custom_metric,
                          verbose            = True
                      )

print(f"Backtest error: {metric}")


# Set to zero predictions for closed hours
hour = data_test.index.hour
day_of_week = data_test.index.day_of_week
closed_hours = (hour < 7) | (hour > 20)
closed_days = day_of_week.isin([5, 6])
is_closed = (closed_hours) | (closed_days)
predictions[is_closed] = 0


# Plot predictions vs real value

fig = go.Figure()
trace1 = go.Scatter(x=data_test.index, y=data_test['users'], name="test", mode="lines")
trace2 = go.Scatter(x=predictions.index, y=predictions['pred'], name="prediction", mode="lines")
fig.add_trace(trace1)
fig.add_trace(trace2)
fig.update_layout(
    title="Real value vs predicted in test data",
    xaxis_title="Date time",
    yaxis_title="Users",
    width  = 800,
    height = 400,
    margin=dict(l=20, r=20, t=35, b=20),
    legend=dict(
        orientation="h",
        yanchor="top",
        y=1.1,
        xanchor="left",
        x=0.001
    )
)
fig.show()
```



When analysing the estimated predictions, it is clear that the model struggles to accurately capture the patterns of store opening and closing times. In addition, the influence of lags leads to an underestimation of the first few hours of the day and the days following the closing days.

```
# Distribution of residuals by day of week and hour of day
residuals = (predictions['pred'] - data_test['users']).to_frame('residuals')
residuals['week_day'] = residuals.index.day_of_week + 1
residuals['hour_day'] = residuals.index.hour + 1

fig = px.box(
        residuals,
        x="week_day",
        y="residuals",
        title = 'Distribution of residuals per hour of day',
        width=600,
        height=300
     )
fig.update_layout(margin=dict(l=20, r=20, t=35, b=20))
fig.show()

fig = px.box(
        residuals,
        x="hour_day",
        y="residuals",
        title = 'Distribution of residuals per hour of day',
        width=600,
        height=300
     )
fig.update_layout(margin=dict(l=20, r=20, t=35, b=20))
fig.show()
```





Inform the model when the store is closed

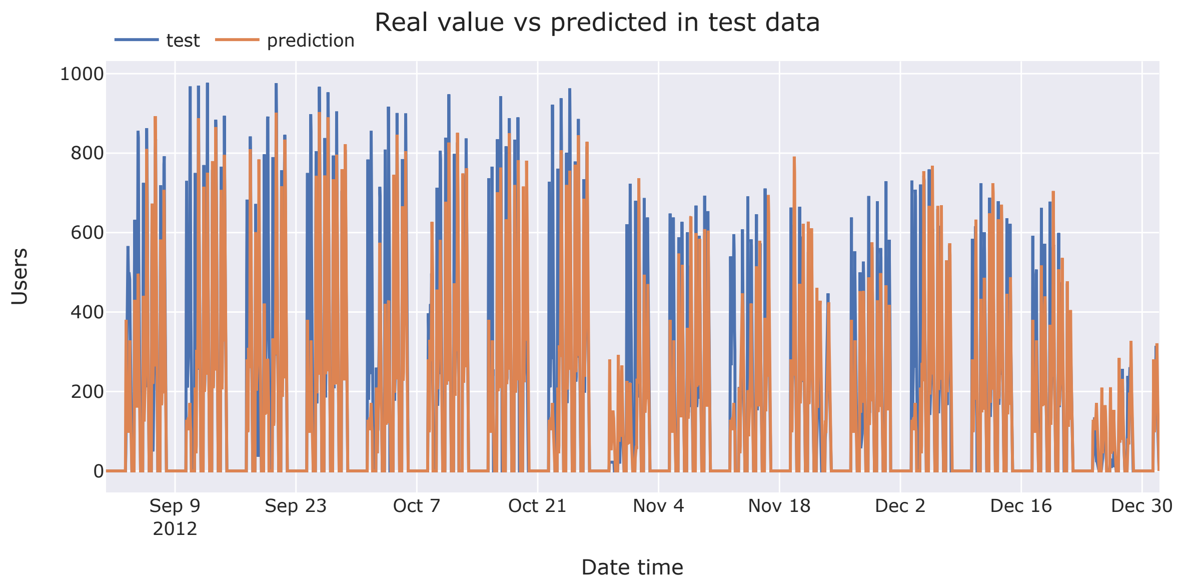
Exogenous variables can be used in a forecasting model to provide additional information and improve the model's ability to detect patterns. This approach offers the advantage of incorporating external factors that could influence the accuracy of the forecast, leading to a more reliable and accurate forecasting model.

```
# Create exogenous variable
hour = data.index.hour
day_of_week = data.index.day_of_week
closed_hours = (hour < 7) | (hour > 20)
closed_days = day_of_week.isin([5, 6])
is_closed = (closed_hours) | (closed_days)
data['is_closed'] = is_closed

end_train = '2012-03-31 23:59:00'
end_validation = '2012-08-31 23:59:00'
data_train = data.loc[: end_train, :]
data_val   = data.loc[end_train:end_validation, :]
data_test  = data.loc[end_validation:, :]
```

```
# Backtesting test period
metric, predictions = backtesting_forecaster(
    forecaster         = forecaster,
    y                  = data['users'],
    exog               = data['is_closed'],
    initial_train_size = len(data.loc[:end_validation]),
    fixed_train_size   = False,
    steps              = 36,
    refit              = False,
    metric             = custom_metric,
    verbose            = False
)

print(f"Backtest error: {metric}")
```

```
# Set to zero predictions for closed hours
hour = data_test.index.hour
day_of_week = data_test.index.day_of_week
closed_hours = (hour < 7) | (hour > 20)
closed_days = day_of_week.isin([5, 6])
is_closed = (closed_hours) | (closed_days)
predictions[is_closed] = 0
```

```
# Plot predictions vs real value
fig = go.Figure()
trace1 = go.Scatter(x=data_test.index, y=data_test['users'], name="test", mode="lines")
trace2 = go.Scatter(x=predictions.index, y=predictions['pred'], name="prediction", mode="lines")
fig.add_trace(trace1)
fig.add_trace(trace2)
fig.update_layout(
    title="Real value vs predicted in test data",
    xaxis_title="Date time",
    yaxis_title="Users",
    width  = 800,
    height = 400,
    margin=dict(l=20, r=20, t=35, b=20),
    legend=dict(
        orientation="h",
        yanchor="top",
        y=1.1,
        xanchor="left",
        x=0.001
    )
)
fig.show()
```



Incorporating an exogenous variable into the model led to a significant improvement in forecasting accuracy, with the error being halved. This highlights the importance of using external factors to enhance the performance of the forecasting model.

Model tunning

Hyperparameter tuning is the process of selecting the best hyperparameters for a machine learning model to optimize its performance.In the context of forecasting models, the lags used in the model can also be considered as hyperparameters.

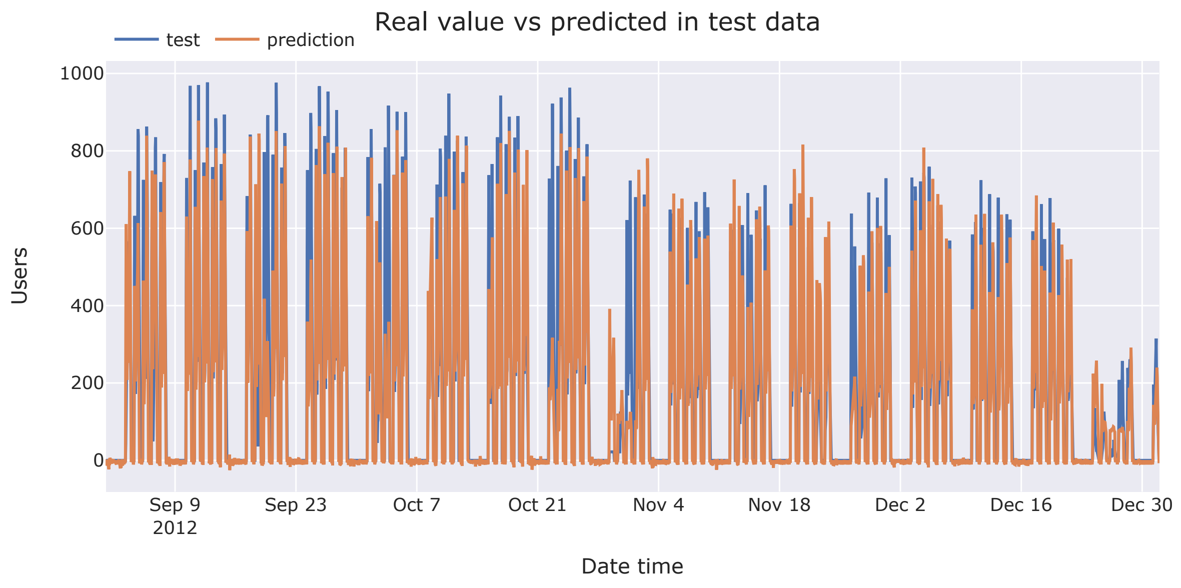
The skforecast library provides strategies to find the optimal combination of hyperparameters for forecasting models. These strategies include:

- Grid Search: Grid search involves exhaustively testing all possible combinations of hyperparameters within a specified grid. This approach can be computationally expensive, especially for models with a large number of hyperparameters, but it ensures that all possible combinations are evaluated.
- Random Search: Random search randomly samples hyperparameters from predefined distributions and evaluates their performance. This approach is less computationally expensive than grid search but still explores a wide range of hyperparameter values.
- Bayesian Search: Bayesian search uses probabilistic models to iteratively update the search space based on the performance of previously evaluated hyperparameters. This approach is more efficient than grid search and random search as it focuses on promising regions of the hyperparameter space.

By systematically testing different combinations of hyperparameters, including lags for forecasting models, these tuning strategies help identify the optimal configuration that maximizes the model's performance on a validation dataset. This optimized model can then be used for accurate forecasting on new data.

```
# Grid search hyperparameters and lags

forecaster = ForecasterAutoreg(
                 regressor = LGBMRegressor(
                                 learning_rate = 0.1,
                                 max_depth     = 5,
                                 n_estimators  = 500,
                                 random_state  = 123,
                             ),
                 lags = 24
             )

# Lags used as predictors
lags_grid = [24, 48, 72]

# Regressor hyperparameters
param_grid = {'n_estimators': [50, 100, 500],
              'max_depth': [5, 10, 15],
              'learning_rate': [0.01, 0.1]}

results_grid = grid_search_forecaster(
                   forecaster         = forecaster,
                   y                  = data.loc[:end_validation, 'users'],
                   exog               = data.loc[:end_validation, 'is_closed'],
                   param_grid         = param_grid,
                   lags_grid          = lags_grid,
                   steps              = 36,
                   refit              = False,
                   metric             = custom_metric,
                   initial_train_size = len(data.loc[:end_train]),
                   fixed_train_size   = False,
                   return_best        = True,
                   verbose            = False
               )


# Backtesting test period

metric, predictions = backtesting_forecaster(
                          forecaster         = forecaster,
                          y                  = data['users'],
                          exog               = data['is_closed'],
                          initial_train_size = len(data.loc[:end_validation]),
                          fixed_train_size   = False,
                          steps              = 36,
                          refit              = False,
                          metric             = custom_metric,
                          verbose            = False
                      )

print(f"Backtest error: {metric}")


# Plot predictions vs real value
```

```
fig = go.Figure()
trace1 = go.Scatter(x=data_test.index, y=data_test['users'], name="test", mode="lines")
trace2 = go.Scatter(x=predictions.index, y=predictions['pred'], name="prediction", mode="lines")
fig.add_trace(trace1)
fig.add_trace(trace2)
fig.update_layout(
    title="Real value vs predicted in test data",
    xaxis_title="Date time",
    yaxis_title="Users",
    width  = 800,
    height = 400,
    margin=dict(l=20, r=20, t=35, b=20),
    legend=dict(
        orientation="h",
        yanchor="top",
        y=1.1,
        xanchor="left",
        x=0.001
    )
)
fig.show()
```



An improved set of hyperparameters has been discovered through grid search, resulting in a further reduction of forecast errors.

```
Start coding or generate with AI.
```