**A Step-by-Step Guide to Feature Engineering for Multivariate Time Series**

**Adding new features based on summary statistics using Python**



Photo by [Artem Maltsev](https://unsplash.com/@art_maltsev?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com?utm_source=medium&utm_medium=referral)

This is a step-by-step guide to feature engineering for [multivariate time series forecasting](https://towardsdatascience.com/machine-learning-for-forecasting-supervised-learning-with-multivariate-time-series-b5b5044fe068). You’ll learn how to compute several rolling statistics. Adding these to the explanatory variables often leads to better forecasting performance.

**Introduction**

**Auto-regression**

A multivariate time series contains two or more variables. See below for an example. Often, these data sets are studied with the goal of forecasting one or more of these variables.

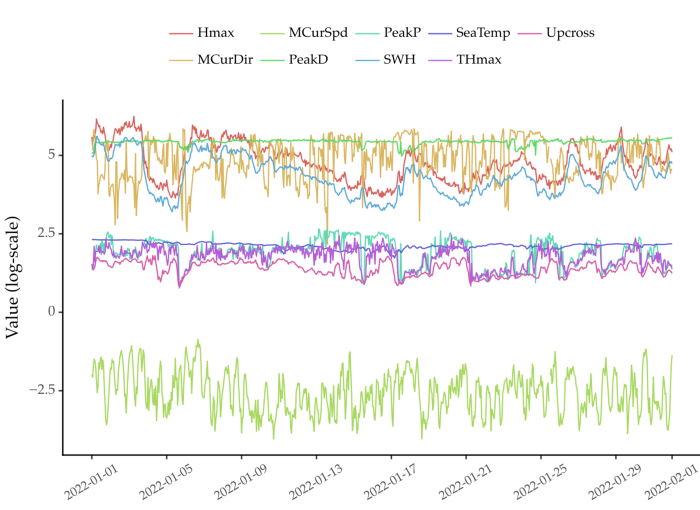


Figure 1: A multivariate time series that contains 9 variables. These represent ocean conditions captured by a smart buoy. [Data is publicly available [1] (CC BY 4.0 License)](https://erddap.marine.ie/erddap/tabledap/IWaveBNetwork.html). Image by Author.

Most forecasting models are based on auto-regression. This amounts to solving a supervised learning regression task. [The future values of the series are the target variables. The input explanatory variables are the recent past values of each variable.](https://towardsdatascience.com/machine-learning-for-forecasting-supervised-learning-with-multivariate-time-series-b5b5044fe068)

Auto-regression works under one main assumption. That recent past values contain enough information about the future. But this might not be true.

You can try to extract more information from recent data. For example, rolling summary statistics are helpful in describing recent dynamics.

**Automated Feature Engineering**

Feature engineering involves extracting and curating explanatory variables. It is a key stage in any data science project. The quality of the features is a central aspect of the performance of models. Because of this, data scientists spend a lot of time in this process.

Yet, feature engineering is often an ad-hoc process. Data scientists create features based on their domain knowledge and expertise. So, automating part of this process is desirable for practitioners.

Let’s see how you can do this for multivariate time series.

**Feature Engineering for Multivariate Time Series**

**Reading the data**

We’ll use a multivariate time series [collected from a smart buoy](https://erddap.marine.ie/erddap/tabledap/IWaveBNetwork.html) as a case study [1]. This buoy is placed on the coast of Ireland. It captures 9 variables related to ocean conditions. These include [sea temperature, the height of waves, and sea water velocity](https://erddap.marine.ie/erddap/info/IWaveBNetwork/index.html), among others. Figure 1 above shows the plot for the first month of 2022.

Here’s how you can read this data using pandas:

import pandas as pd  
  
# skipping second row, setting time column as a datetime column  
# dataset available here: https://github.com/vcerqueira/blog/tree/main/data  
buoy = pd.read\_csv('data/smart\_buoy.csv',   
 skiprows=[1],   
 parse\_dates=['time'])  
  
# setting time as index  
buoy.set\_index('time', inplace=True)  
# resampling to hourly data  
buoy = buoy.resample('H').mean()  
# simplifying column names  
buoy.columns = [  
 'PeakP', 'PeakD', 'Upcross',  
 'SWH', 'SeaTemp', 'Hmax', 'THmax',  
 'MCurDir', 'MCurSpd'  
]

Figure 1 above shows the plot for the first month of 2022.

The goal is to forecast the future values of the SWH (significant wave height) variable. This variable is often used to quantify the height of ocean waves. [One of the use cases of this problem is to estimate the energy produced from ocean waves](https://medium.com/towards-data-science/an-introduction-to-exceedance-probability-forecasting-4c96c0e7772c). This energy source is an increasingly popular alternative to non-renewable energy sources.

**Auto-regressive model**

The time series is multivariate. So, you can use an ARDL (Auto-regressive distributed lags) approach to solve this task. You can learn more about this method in [my previous post](https://towardsdatascience.com/machine-learning-for-forecasting-supervised-learning-with-multivariate-time-series-b5b5044fe068).

Here’s how you’d implement this method.

import pandas as pd  
  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import mean\_absolute\_percentage\_error as mape  
from sklearn.multioutput import MultiOutputRegressor  
from lightgbm import LGBMRegressor  
  
# https://github.com/vcerqueira/blog/blob/main/src/tde.py  
from src.tde import time\_delay\_embedding  
  
target\_var = 'SWH'  
  
colnames = buoy.columns.tolist()  
  
# create data set with lagged features using time delay embedding  
buoy\_ds = []  
for col in buoy:  
 col\_df = time\_delay\_embedding(buoy[col], n\_lags=24, horizon=12)  
 buoy\_ds.append(col\_df)  
  
# concatenating all variables  
buoy\_df = pd.concat(buoy\_ds, axis=1).dropna()  
  
# defining target (Y) and explanatory variables (X)  
predictor\_variables = buoy\_df.columns.str.contains('\(t\-')  
target\_variables = buoy\_df.columns.str.contains(f'{target\_var}\(t\+')  
X = buoy\_df.iloc[:, predictor\_variables]  
Y = buoy\_df.iloc[:, target\_variables]  
  
# train/test split  
X\_tr, X\_ts, Y\_tr, Y\_ts = train\_test\_split(X, Y, test\_size=0.3, shuffle=False)  
  
# fitting a lgbm model without feature engineering  
model\_wo\_fe = MultiOutputRegressor(LGBMRegressor())  
model\_wo\_fe.fit(X\_tr, Y\_tr)  
  
# getting forecasts for the test set  
preds\_wo\_fe = model\_wo\_fe.predict(X\_ts)  
  
# computing the MAPE error  
mape(Y\_ts, preds\_wo\_fe)  
# 0.238

First, the time series is transformed into an auto-regressive problem. This is done with the function *time\_delay\_embedding*. The predictive goal is to forecast the next 12 values of SWH (*horizon=12*). The explanatory variables are the past 24 values of each variable in the series (*n\_lags=24*).

A LightGBM is trained for each forecasting horizon using a direct approach. [The *direct* method is a popular approach for multi-step ahead forecasting](https://medium.com/towards-data-science/6-methods-for-multi-step-forecasting-823cbde4127a). It is implemented in scikit-learn with the name *MultiOutputRegressor.*

The code above builds and tests an auto-regressive model. The explanatory variables include only the recent past values of each variable. This leads to a mean absolute percentage error of *0.238*. Let’s see if this score can be improved with feature engineering.

This guide includes two approaches to extract features from multivariate time series:

* **Univariate feature extraction.** Computing rolling statistics of each variable. For example, a rolling average can be used to smooth out spurious observations;
* **Bivariate feature extraction.** Computing rolling statistics of pairs of variables to summarise their interaction. For example, the rolling covariance between two variables.

**Univariate Feature Extraction**

You can summarise the recent past values of each variable. For example, computing a rolling average to summarise the recent level. Or a rolling dispersion to get a sense of the recent degree of dispersion.

import numpy as np  
  
SUMMARY\_STATS = {  
 'mean': np.mean,  
 'sdev': np.std,  
}  
  
univariate\_features = {}  
# for each column in the data  
for col in colnames:  
 # get lags for that column  
 X\_col = X.iloc[:, X.columns.str.startswith(col)]  
  
 # for each summary stat  
 for feat, func in SUMMARY\_STATS.items():  
 # compute that stat along the rows  
 univariate\_features[f'{col}\_{feat}'] = X\_col.apply(func, axis=1)  
  
# concatenate features into a pd.DF  
univariate\_features\_df = pd.concat(univariate\_features, axis=1)

You may want to add a few more statistics. You can do this by adding functions to the SUMMARY\_STATS dictionary. Having these functions in a single dictionary keeps the code tidy.

**Bivariate Feature Extraction**

Univariate statistics miss potential interactions between different variables. You can capture this information using a bivariate feature extraction process.

The idea is to compute features for different pairs of variables. So, you summarise the joint dynamics of these pairs with binary statistics.

There are two ways you can do this:

* **Rolling binary statistics.** Compute statistics that take pairs of variables as input. For example, the rolling covariance or rolling correlation;
* **Rolling binary transformation followed by univariate statistics.** Transform a pair of variables into a single variable, and summarise this variable. For example, computing the elementwise cross-correlation and then taking its average value.

Examples of rolling binary statistics include covariance, correlation, or relative entropy.

There are many binary transformation possibilities. For example, the percentage difference, cross-correlation, or linear convolution between pairs of variables. Then, these transformations are summarised with statistics such as mean or standard deviation.

Here’s the script used to do these two extraction processes in one go.

import itertools  
  
import pandas as pd  
  
from scipy.spatial.distance import jensenshannon  
from scipy import signal  
from scipy.special import rel\_entr  
  
from src.feature\_extraction import covariance, co\_integration  
  
BIVARIATE\_STATS = {  
 'covariance': covariance,  
 'co\_integration': co\_integration,  
 'js\_div': jensenshannon,  
}  
  
BIVARIATE\_TRANSFORMATIONS = {  
 'corr': signal.correlate,  
 'conv': signal.convolve,  
 'rel\_entr': rel\_entr,  
}  
  
# get all pairs of variables  
col\_combs = list(itertools.combinations(colnames, 2))  
  
bivariate\_features = []  
# for each row  
for i, \_ in X.iterrows():  
 # feature set in the i-th time-step  
 feature\_set\_i = {}  
 for col1, col2 in col\_combs:  
 # features for pair of columns col1, col2  
  
 # getting the i-th instance for each column  
 x1 = X.loc[i, X.columns.str.startswith(col1)]  
 x2 = X.loc[i, X.columns.str.startswith(col2)]  
  
 # compute each summary stat  
 for feat, func in BIVARIATE\_SUMMARY\_STATS.items():  
 feature\_set\_i[f'{col1}|{col2}\_{feat}'] = func(x1, x2)  
  
 # for each transformation  
 for trans\_f, t\_func in BIVARIATE\_TRANSFORMATIONS.items():  
  
 # apply transformation  
 xt = t\_func(x1, x2)  
  
 # compute summary stat  
 for feat, s\_func in SUMMARY\_STATS.items():  
 feature\_set\_i[f'{col1}|{col2}\_{trans\_f}\_{feat}'] = s\_func(xt)  
  
 bivariate\_features.append(feature\_set\_i)  
  
bivariate\_features\_df = pd.DataFrame(bivariate\_features, index=X.index)

Again, you can add extra transformations or statistics. This is done by including them in the dictionaries BIVARIATE\_TRANSFORMATIONS or BIVARIATE\_STATS.

After extracting all the features, you concatenate them to the original explanatory variables. Then, the training and testing cycle is like what you did before.

# concatenating all features with lags  
X\_with\_features = pd.concat([X, univariate\_features\_df, bivariate\_features\_df], axis=1)  
  
# train/test split  
X\_tr, X\_ts, Y\_tr, Y\_ts = train\_test\_split(X\_with\_features, Y, test\_size=0.3, shuffle=False)  
  
# fitting a lgbm model with feature engineering  
model\_w\_fe = MultiOutputRegressor(LGBMRegressor())  
model\_w\_fe.fit(X\_tr, Y\_tr)  
  
# getting forecasts for the test set  
preds\_w\_fe = model\_w\_fe.predict(X\_ts)  
  
# computing MAPE error  
print(mape(Y\_ts, preds\_w\_fe))  
# 0.227

This leads to a *0.227* mean absolute percentage error, which is an improvement. The approach that doesn’t use feature engineering suffered a larger loss (*0.238*).

**Feature Selection**

The extraction process above results in a total of 558 explanatory variables.

Depending on the number of variables and summary statistics, this may lead to a high-dimensional problem. So, it is important to remove poor or redundant features from the data set.

One way you can do this is to grab the most important features and repeat the training process with these.

# getting the importance of each feature in each horizon  
avg\_imp = pd.DataFrame([x.feature\_importances\_  
 for x in model\_w\_fe.estimators\_]).mean()  
  
# getting the top 100 features  
n\_top\_features = 100  
  
importance\_scores = pd.Series(dict(zip(X\_tr.columns, avg\_imp)))  
top\_features = importance\_scores.sort\_values(ascending=False)[:n\_top\_features]  
top\_features\_nm = top\_features.index  
  
# subsetting training and testing sets by those features  
X\_tr\_top = X\_tr[top\_features\_nm]  
X\_ts\_top = X\_ts[top\_features\_nm]  
  
# re-fitting the lgbm model  
model\_top\_features = MultiOutputRegressor(LGBMRegressor())  
model\_top\_features.fit(X\_tr\_top, Y\_tr)  
  
# getting forecasts for the test set  
preds\_top\_feats = model\_top\_features.predict(X\_ts\_top)  
  
# computing MAE error  
mape(Y\_ts, preds\_top\_feats)  
# 0.229

The top 100 features lead to a similar performance as the complete 558 features.

Here’s the importance of the top 15 features (others omitted for conciseness):

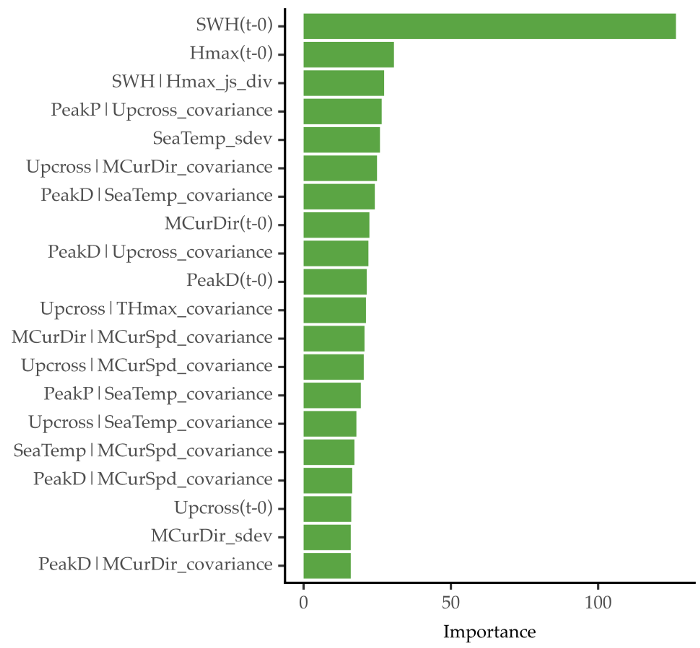


Figure 2: Top 15 features according to the forecasting model. Image by Author.

The most important feature is the first lag of the target variable. But, some of the extracted features are present in this top 15. For example, the third best feature in SWH|Hmax\_js\_div. This represents the Jensen-Shannon divergence between the lags of the target variable and the lags of Hmax. The fifth best feature is SeaTemp\_sdev: the standard deviation sea temperature lags. The covariance is also a relevant statistic for different pairs of variables.

Another way to remove redundant features is to apply a correlation filter. You remove highly correlated features to reduce the dimension of the data.

**Summarising the Whole Time Series**

This guide focuses on forecasting problems with multivariate time series. So, the feature extraction process is applied to many subsequences of a time series. At each time step, you summarised the past 24 hours of data with a set of statistics.

But, you can also apply these statistics to characterize the whole time series in one go. This might be useful if your goal is to cluster a set of time series. First, you summarise each time series with feature extraction. Then, you apply a clustering algorithm to the resulting features.

**Key Take-Aways**

* Multivariate time series forecasting is usually an auto-regressive process
* Feature engineering is a key step in data science projects.
* You can improve multivariate time series data sets with feature engineering. This includes computing univariate and bivariate transformations and summary statistics.
* Extracting too many features results in a high-dimensional problem. You can use feature selection methods to remove unwanted features.

Thanks for reading, and see you in the next story!

**References**

[1] Data source: <https://erddap.marine.ie/erddap/tabledap/IWaveBNetwork.html>. License: Creative Commons Attribution 4.0 (<https://erddap.marine.ie/erddap/info/IWaveBNetwork/index.html>)

[2] Cerqueira, Vitor, Nuno Moniz, and Carlos Soares. “Vest: Automatic feature engineering for forecasting.” *Machine Learning* (2021): 1–23.