**Build Complex Time Series Regression Pipelines with sktime**

**How to forecast with scikit-learn and XGBoost models with sktime**



Photo by [Markus Winkler](https://unsplash.com/@markuswinkler?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com?utm_source=medium&utm_medium=referral)

Stop using *scikit-learn* for forecasting. Just because you can use your existing regression pipeline doesn’t mean you should. Alternatively, aren’t you bored of forecasting using the same old techniques, such as exponential smoothing and ARIMA? Wouldn’t it be more fun to use a more advanced algorithm such as a gradient boosted tree?

Using *sktime*, you can do this (and more). *Sktime* is a library that lets you safely use any *scikit-learn* compatible regression model for time series forecasting. This tutorial will discuss how we can convert a time series forecasting problem to a regression problem using *sktime*. I will also show you how to build a complex time series forecaster with the popular library, *XGBoost*.

**Background and motivation**

**What is time series forecasting?**

Time series forecasting is a technique to predict one or more future values. Like regression modelling, a data practitioner can fit a model based on historical data and use this model to predict future observations.

Some of the most popular models used in forecasting are univariate. Univariate models forecast using only the previous observations. [Exponential smoothing](https://otexts.com/fpp2/expsmooth.html) models use a weighted average of past observations to predict future values, with more recent data points given more weight. [ARIMA](https://otexts.com/fpp2/arima.html)models base their predictions on the [autocorrelations](https://otexts.com/fpp2/autocorrelation.html) in the data. Autocorrelation measures the relationship between time-shifted (i.e. lagged) values of a time series.

If time series forecasting sounds similar to regression modelling, you’re correct. We can use regression models for our time-series predictions.

**Why would we want to use regression for time series forecasting?**

One of the reasons you should use regression models is for improved model performance. Regression techniques are flexible, and you can go beyond univariate modelling of past observations. Including time-step features, such as the day of the week or holidays, can enrich your data and potentially uncover hidden trends in your data.

Secondly, you likely already use popular machine learning frameworks such as *scikit-learn* and *XGBoost*. Converting your time series forecasting task into a regression problem can save you time if you’re already familiar with these libraries.

However, modern forecasting libraries, such as Facebook’s [*Prophet*](https://facebook.github.io/prophet/), also provide the flexibility for multivariate analysis and a simple API. Nonetheless, using regression techniques for forecasting should be another tool in your data science toolkit.

**What do we need to watch out for in time series regression?**

Time series regression isn’t without issues. Autocorrelated observations violate the assumptions of linear regression models. However, more complex regression models, such as gradient tree boosting, are generally robust to multicollinearity.

Additionally, we can introduce data leakage if we blindly fit and tune our regression model. [Data leakage](https://towardsdatascience.com/avoiding-data-leakage-in-timeseries-101-25ea13fcb15f) is the use of information in the model training and validation phase that would not otherwise be available for forecasting. For example, randomly shuffling your data using K-fold cross-validation allows your model to peek ahead into the future. Instead, you should use [temporal cross-validation](https://towardsdatascience.com/dont-use-k-fold-validation-for-time-series-forecasting-30b724aaea64).

Rather than implementing time-series regression models from scratch, you can use the [*sktime*](https://www.sktime.org/en/stable/) framework.

**Why should we use sktime?**

[*Sktime*](https://www.sktime.org/en/stable/) is an open-source framework for various machine learning tasks for modelling time series, including time-series regression, classification, clustering and annotation. The framework combines features from several libraries with a user experience similar to *scikit-learn*.

We will be using *sktime* for our forecasting task because it provides [functionality](https://www.sktime.org/en/stable/examples/01_forecasting.html#3.-Advanced-composition-patterns---pipelines,-reduction,-autoML,-and-more) such as:

* **Reduction**: building a time series forecaster using estimators compatible with the *scikit-learn* API
* **Tuning**: determining the value of hyperparameters using grid search strategies with temporal cross-validation
* **Evaluation**: *sktime* includes several performance metrics (e.g. MAPE, MASE) and provides an easy implementation for custom scorers and backtesting
* **Pipelining:** an extension of the *scikit-learn* [*Pipeline*](https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html), which is a series of concatenated transformers to build a single forecaster
* **AutoML**: automated tuning strategies to determine the best forecaster across a range of models and hyperparameters

*Sktime* also provides [several other classes](https://www.sktime.org/en/stable/examples/01_forecasting.html#2.-Forecasters-in-sktime---lookup,-properties,-main-families) of forecasting models, including exponential smoothing, ARIMA, BATS and Prophet. We will not discuss these forecasting models in this tutorial, but I encourage you to implement these forecasters during your model selection phase.

**Code Examples**

**Dataset**

The dataset we will be using in this tutorial is an [hourly count of pedestrians](https://data.melbourne.vic.gov.au/Transport/Pedestrian-Counting-System-Monthly-counts-per-hour/b2ak-trbp) in the city of Melbourne, Australia [1]. We will restrict our analysis from 1st January 2010 to 31st December 2019. I’ve aggregated the data weekly.

If we visualise our data, we can see a general upwards trend from about mid-2013. We can also see more pedestrians in the city in December, likely due to Christmas shopping, and in July for end-of-financial-year sales. Our data appears non-stationary, confirmed when we run an Augmented Dickey-Fuller test.

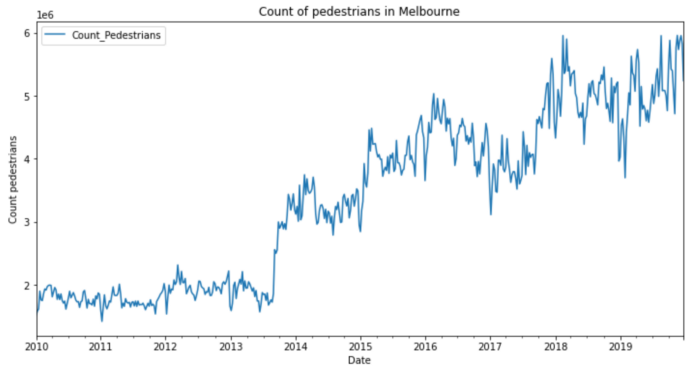


Image by author

**A simple forecaster with linear regression**

The first example we’ll run is a simple forecaster using linear regression. We first take the final 26 weeks of data as our test set using *sktime’s* [*temporal\_train\_test\_split*](http://temporal_train_test_split). The function does not shuffle the data. Therefore it is suitable for forecasting.

from sktime.forecasting.model\_selection import temporal\_train\_test\_splity\_train, y\_test = temporal\_train\_test\_split(y, test\_size=26)

We also specify our forecasting horizon, the 26 weeks of test data that we will predict. We achieve this using the [*ForecastingHorizon*](https://www.sktime.org/en/stable/api_reference/auto_generated/sktime.forecasting.base.ForecastingHorizon.html) object.

from sktime.forecasting.base import ForecastingHorizonfh = ForecastingHorizon(y\_test.index, is\_relative=False)

With the preliminaries complete, we can move on to instantiating our *scikit-learn* [*LinearRegression*](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html) object that will be the estimator for our forecaster. Alternatively, you can use other *scikit-learn* estimators or *scikit-learn* API compatible estimators, such as *XGBoost*. I will demonstrate this in the following example.

from sklearn.linear\_model import LinearRegressionregressor = LinearRegression()

*Sktime’s* [*make\_reduction*](https://www.sktime.org/en/stable/api_reference/auto_generated/sktime.forecasting.compose.make_reduction.html?highlight=make_reduction) function transforms the time series into tabular data compatible with our *scikit-learn* estimator. The parameter, ‘window\_length’, controls the number of lags in our sliding window transformation.

Consider the simple time series, denoted *‘y’*, below. If we apply our *make\_reduction* function, with a window length equal to 3, we produce a tabular dataset with three input variables, denoted *‘lag\_1*’, *‘lag\_2*’ and *‘lag\_3*’. The function transforms our one-dimensional time series dataset into a compatible format with our regression estimator.

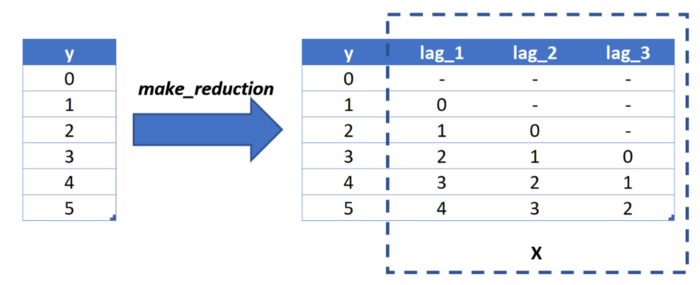


Image by author

Furthermore, the *make\_reduction* function has an input ‘strategy’, which controls how the forecaster generates its predictions. The parameter is relevant when we are making more than one-step ahead forecasts. In our example, we are making predictions for the next 26 weeks of pedestrian counts (multi-step) rather than just the following week (one-step).

We have the choice of three options for our multi-step forecasting strategy:

* **Direct**: we create a separate model for each period we are forecasting. In our example, we fit 26 models, each making a single prediction.
* **Recursive**: we fit a single one-step ahead model. However, we use the previous time step’s output for the following input. For example, we use next week’s prediction as an input for the prediction two weeks away, and so on.
* **Multiple outputs**: one model is used to predict the entire time series horizon in a single forecast. The use of this option is dependent on having a model capable of predicting sequences.

In our example, we will use the recursive strategy.

from sktime.forecasting.compose import make\_reductionforecaster = make\_reduction(regressor, window\_length=52, strategy="recursive")

We can now fit our linear regression forecaster and predict the 26 weeks of pedestrian counts on our test data.

forecaster.fit(y\_train)  
y\_pred = forecaster.predict(fh)

We then plot our predictions relative to the train and test data to determine the model’s suitability. *Sktime* makes this easy using the [*plot\_series*](https://www.sktime.org/en/stable/api_reference/auto_generated/sktime.utils.plotting.plot_series.html?highlight=plot_series) utility function.

from sktime.utils.plotting import plot\_seriesplot\_series(y\_train['2018-07-01':], y\_test, y\_pred, labels=["y\_train", "y\_test", "y\_pred"], x\_label='Date', y\_label='Count pedestrians');

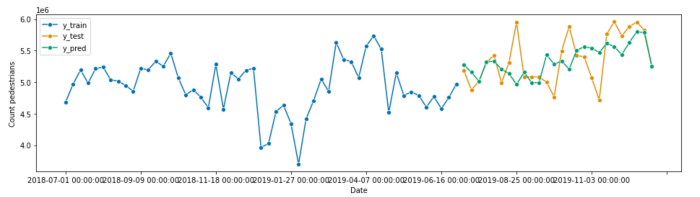


Image by author

Our linear regression forecaster appears to give us a reasonable fit. However, it is a conservative prediction and misses the peaks and troughs of the test data.

We evaluate our model using the [*mean\_absolute\_percentage\_error*](https://www.sktime.org/en/stable/api_reference/auto_generated/sktime.performance_metrics.forecasting.mean_absolute_percentage_error.html)(MAPE),which yields a result of 4.8%.

from sktime.performance\_metrics.forecasting import mean\_absolute\_percentage\_errorprint('MAPE: %.4f' % mean\_absolute\_percentage\_error(y\_test, y\_pred, symmetric=False))

If we put all this together, our code looks like this:

Let’s see if we can beat this using a more complex algorithm such as *XGBoost*.

**Time series forecasting with XGBoost *and exogenous inputs***

[*XGBoost*](https://xgboost.readthedocs.io/en/stable/) is an implementation of a gradient boosting machine, popular for tabular machine learning tasks because of its speed and performance. We can use *XGBoost* for time series forecasting because it has a [*scikit-learn* wrapper](https://xgboost.readthedocs.io/en/stable/python/python_api.html#module-xgboost.sklearn) compatible with *sktime’s* *make\_reduction* function.

from xgboost import XGBRegressorregressor = XGBRegressor(objective='reg:squarederror', random\_state=42)  
forecaster = make\_reduction(regressor, window\_length=52, strategy="recursive")

We will also include exogenous data, another time helpful series for the prediction that is not forecasted. Let’s include several dummy variables that indicate the year’s month, split into our train and test sets.

# Create an exogenous dataframe indicating the month  
X = pd.DataFrame({'month': y.index.month}, index=y.index)  
X = pd.get\_dummies(X.astype(str), drop\_first=True)# Split into train and test  
X\_train, X\_test = temporal\_train\_test\_split(X, test\_size=26)

We include our exogenous data when we call the fit and predict methods. You can find more information in the [*RecursiveTabularRegressionForecaster*](https://www.sktime.org/en/stable/api_reference/auto_generated/sktime.forecasting.compose.RecursiveTabularRegressionForecaster.html#sktime.forecasting.compose.RecursiveTabularRegressionForecaster.fit) documentation.

forecaster.fit(y=y\_train, X=X\_train)  
y\_pred = forecaster.predict(fh=fh, X=X\_test)

Visually, our predictions appear to be worse than our linear regression forecaster. Our expectation is confirmed when we look at the MAPE, which has increased to 7.1%. The *XGBoost* forecaster is likely under-fitted because we’ve used the default hyperparameter values.

The following example will demonstrate how we can tune the hyperparameters for our *XGBoost* forecaster.

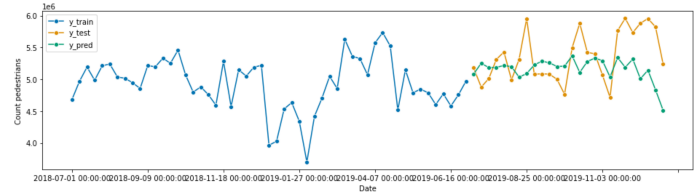


Image by author

You can find the code for this example below.

**Tuning the hyperparameters of our forecaster**

Our [*XGBRegressor*](https://xgboost.readthedocs.io/en/stable/python/python_api.html#module-xgboost.sklearn) has several hyperparameters that we can tune, as described in this [article](https://towardsdatascience.com/xgboost-fine-tune-and-optimize-your-model-23d996fab663). We want to adjust the hyperparameters of our forecaster to see if we can improve the performance.

Before tuning our hyperparameters, we must add a validation set to our data. We have several strategies implemented by *sktime*, including a [*SingleWindowSplitter*](https://www.sktime.org/en/stable/api_reference/auto_generated/sktime.forecasting.model_selection.SingleWindowSplitter.html), [*SlidingWindowSplitter*](https://www.sktime.org/en/stable/api_reference/auto_generated/sktime.forecasting.model_selection.SlidingWindowSplitter.html), and [*ExpandingWindowSplitter*](https://www.sktime.org/en/stable/api_reference/auto_generated/sktime.forecasting.model_selection.ExpandingWindowSplitter.html). For simplicity, we will create a single validation set, the same size as our test set. However, this [article](https://towardsdatascience.com/dont-use-k-fold-validation-for-time-series-forecasting-30b724aaea64) explains the differences between the various strategies.

from sktime.forecasting.model\_selection import SingleWindowSplittervalidation\_size = 26  
cv = SingleWindowSplitter(window\_length=len(y)-validation\_size, fh=validation\_size)

*Sktime* has implemented two strategies for hyperparameter tuning: [randomised search](https://www.sktime.org/en/stable/api_reference/auto_generated/sktime.forecasting.model_selection.ForecastingRandomizedSearchCV.html) and [grid search](https://www.sktime.org/en/stable/api_reference/auto_generated/sktime.forecasting.model_selection.ForecastingGridSearchCV.html). We will use a randomised search with 100 iterations.

from sktime.forecasting.model\_selection import ForecastingRandomizedSearchCVparam\_grid = {  
 'estimator\_\_max\_depth': [3, 5, 6, 10, 15, 20],  
 'estimator\_\_learning\_rate': [0.01, 0.1, 0.2, 0.3],  
 'estimator\_\_subsample': np.arange(0.5, 1.0, 0.1),  
 'estimator\_\_colsample\_bytree': np.arange(0.4, 1.0, 0.1),  
 'estimator\_\_colsample\_bylevel': np.arange(0.4, 1.0, 0.1),  
 'estimator\_\_n\_estimators': [100, 500, 1000]  
}regressor = XGBRegressor(objective='reg:squarederror', random\_state=42)  
forecaster = make\_reduction(regressor, window\_length=52, strategy="recursive")gscv = ForecastingRandomizedSearchCV(forecaster, cv=cv, param\_distributions=param\_grid, n\_iter=100, random\_state=42)

Again, we fit and predict.

gscv.fit(y=y\_train, X=X\_train)  
y\_pred = gscv.predict(fh=fh, X=X\_test)

But this time, we can inspect our random search object to see how the forecaster performs using different combinations of hyperparameters.

gscv.cv\_results\_



Image by author

Interestingly, our forecaster performs worse on our test data, as shown by our visualisation and the MAPE increasing to 7.8%.

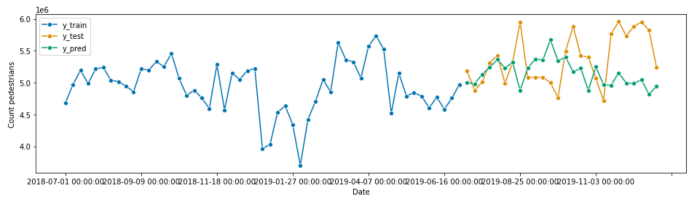


Image by author

You can find the complete code for the example below.

**Adding components to our forecasting pipeline**

We previously trained our model on non-stationary data resulting in a poor forecast. Using [*statsmodels*](https://www.statsmodels.org/dev/generated/statsmodels.tsa.seasonal.seasonal_decompose.html), we can decompose our pedestrian time series to observe the trend and seasonality.

from statsmodels.tsa.seasonal import seasonal\_decomposeresult = seasonal\_decompose(y\_train, model='multiplicative')  
result.plot()  
plt.show()

We assume our time series is multiplicative rather than additive because it appears the oscillation amplitude of our time series increases over time. Looking at the decomposition below, we see that our trend (second subplot) has increased since mid-2013 and that there is a seasonal pattern, with increased pedestrian traffic around Christmas and mid-year and decreased traffic in the first week of January.

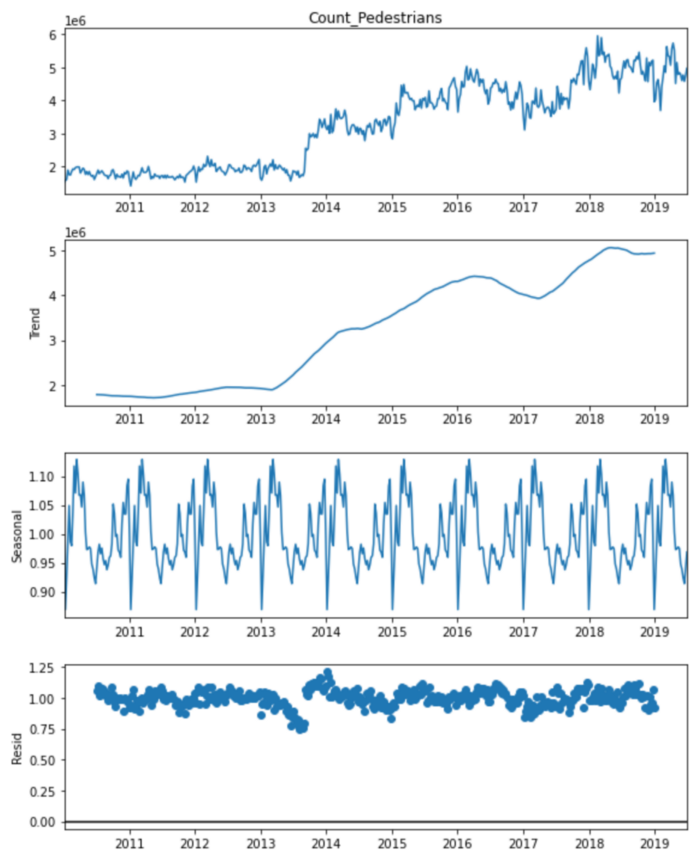


Image by author

Therefore, our forecaster’s performance may be improved by removing the seasonality and trend of the time series, producing a time series closer to being stationary. *Sktime* includes two classes, [*Deseasonalizer*](https://www.sktime.org/en/stable/api_reference/auto_generated/sktime.transformations.series.detrend.Deseasonalizer.html?highlight=deseasonalizer) and [*Detrender*](https://www.sktime.org/en/stable/api_reference/auto_generated/sktime.transformations.series.detrend.Detrender.html)*,* that we can incorporate into our forecasting pipeline.

from sktime.forecasting.compose import TransformedTargetForecaster  
from sktime.transformations.series.detrend import Deseasonalizer, Detrender  
from sktime.forecasting.trend import PolynomialTrendForecasterregressor = XGBRegressor(objective='reg:squarederror', random\_state=42)forecaster = TransformedTargetForecaster(  
 [  
 ("deseasonalize", Deseasonalizer(model="multiplicative", sp=52)),  
 ("detrend", Detrender(forecaster=PolynomialTrendForecaster(degree=1))),  
 ("forecast", make\_reduction(regressor, window\_length=52, strategy="recursive"),  
 ),  
 ]  
)

We can tune the parameters for our *Deaseasonalizer* and *Detrender* using a randomised grid search. For example, we can see if an additive or multiplicative model is best or what degree of the polynomial we want to use to model the trend.

param\_grid = {  
 'deseasonalize\_\_model': ['multiplicative', 'additive'],  
 'detrend\_\_forecaster\_\_degree': [1, 2, 3],  
 'forecast\_\_estimator\_\_max\_depth': [3, 5, 6, 10, 15, 20],  
 'forecast\_\_estimator\_\_learning\_rate': [0.01, 0.1, 0.2, 0.3],  
 'forecast\_\_estimator\_\_subsample': np.arange(0.5, 1.0, 0.1),  
 'forecast\_\_estimator\_\_colsample\_bytree': np.arange(0.4, 1.0, 0.1),  
 'forecast\_\_estimator\_\_colsample\_bylevel': np.arange(0.4, 1.0, 0.1),  
 'forecast\_\_estimator\_\_n\_estimators': [100, 500, 1000]  
}gscv = ForecastingRandomizedSearchCV(forecaster, cv=cv, param\_distributions=param\_grid, n\_iter=100, random\_state=42)  
gscv.fit(y=y\_train, X=X\_train)  
y\_pred = gscv.predict(fh=fh, X=X\_test)

After tuning our forecaster pipeline, we can visualise our predictions. We observe our predictions are much closer to our trend data, indicating that removing the seasonality and trend from our time series has improved the performance of our model. The MAPE has decreased to 4.7%, the best performance of all our models tested.

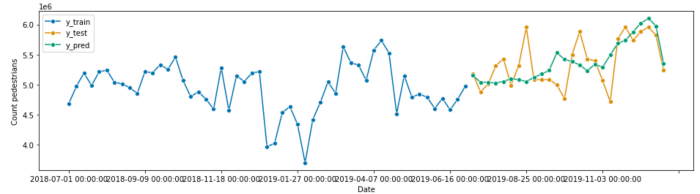


Image by author

The complete code for this example is below.

*Sktime* is a versatile library that lets you use your *scikit-learn* compatible regression model for time series forecasting. You can build complex multi-step pipelines that rival even the most advanced forecasting algorithms. The next time you’re doing a forecasting project, don’t just use ARIMA. Give *sktime* a go.

Liked what you read? Follow me on [Medium](https://medium.com/@rtkilian). Otherwise, [tweet me](https://twitter.com/rtkilian) or add me on [LinkedIn](https://www.linkedin.com/in/rtkilian/).

You can find all the code used in this post on [GitHub](https://github.com/rtkilian/data-science-blogging/blob/main/sktime_regression_forecasting.ipynb).

[1] City of Melbourne, [Pedestrian Counting System — Monthly (counts per hour)](https://data.melbourne.vic.gov.au/Transport/Pedestrian-Counting-System-Monthly-counts-per-hour/b2ak-trbp) (2022), City of Melbourne Open Data published under Creative Commons Attribution 3.0 Australia (CC BY 3.0 AU)