<https://neptune.ai/blog/select-model-for-time-series-prediction-task>

### Joos Korstanje

How to Select a Model For Your Time Series Prediction Task [Guide]

Working with time series data? Here’s a guide for you. In this article, you will learn how to compare and select time series models based on predictive performance.

In the first part, you will be introduced to numerous models for time series. This part is divided into three parts:

* classical time series models,
* supervised models,
* and deep learning-based models.

In the second part, you will see an application to a use case in which you will build a couple of time series models for stock market prediction and you will get to know a few time series modeling techniques. The models will be benchmarked against each other to select the best-performing one.

[Jump right to the hands-on part ->](https://neptune.ai/blog/select-model-for-time-series-prediction-task##times-series-model-selection)

Introduction to time series datasets and forecasting

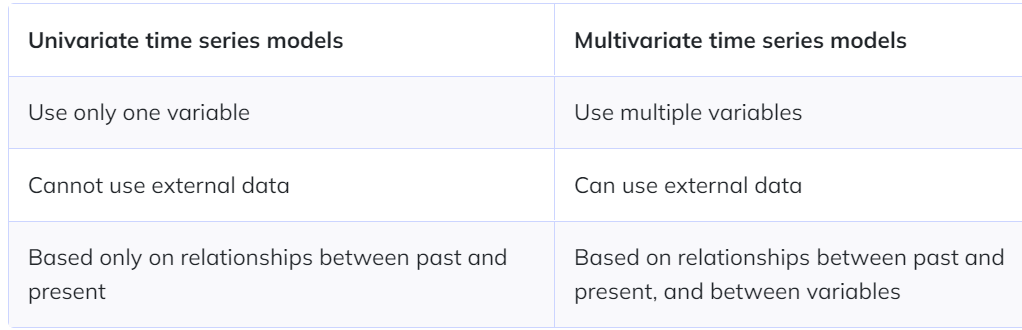
**Time series is a special type of data set in which one or more variables are measured over time.**

|  |  |
| --- | --- |
| **Not time series** | **Time Series** |
| Each observation is independent | Each observation IS DEPENDENT |
|  | Measure over time, each point correspond to a specific time |
| A visitor, with ID, name, independent from other visitors. | Sales according to season, according to specific month |

**Time series models specifics**

**Univariate vs multivariate time series models**

The first specificity of time series is that the timestamp that identifies the data has intrinsic meaning. **Univariate time series models** are forecasting models that use only one variable (the target variable) and its temporal variation to forecast the future. Univariate models are specific to time series.



In other situations, you may have additional explanatory data about the future. For example, imagine that you want to factor the weather forecast into your product demand forecast, or that you have some other data that will influence your predictions. In this case, you can use **Multivariate time series models**. Multivariate Time Series models are the Univariate Time Series models that are adapted to integrate external variables. You can also use supervised machine learning for this task.

If you want to use a temporal variation on your time series data, you will first need to understand the different types of temporal variations that you can expect.

Time series decomposition

**Time Series Decomposition** is a technique to extract multiple types of variation from your dataset. There are 3 important components in the temporal data of a time series:

* Seasonality,
* Trend,
* and Noise.

Time series decomposition example in Python

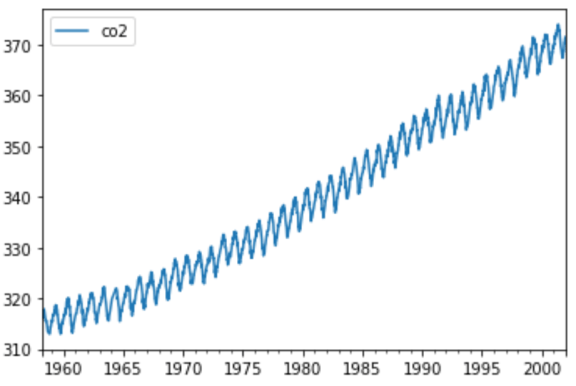
Let’s see a short example to understand how to decompose a time series in Python, using the [CO2 dataset from the statsmodels library](https://www.statsmodels.org/dev/datasets/generated/co2.html).

**import** statsmodels.datasets.co2 **as** co2

co2\_data = co2.load(as\_pandas=**True**).data

co2\_data = co2\_data.fillna(co2\_data.interpolate())

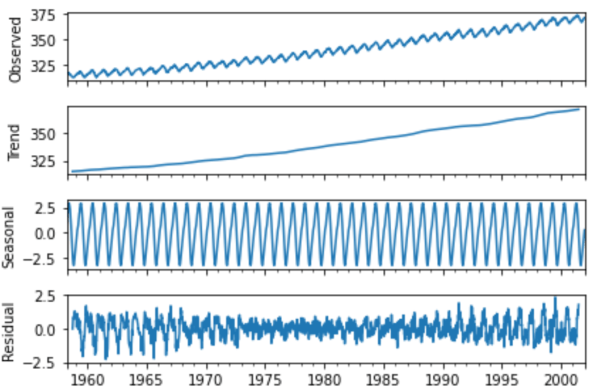
co2\_data.plot()



**from** statsmodels.tsa.seasonal **import** seasonal\_decompose

result = seasonal\_decompose(co2\_data)

result.plot()



The decomposition of the CO2 data shows an upward trend and a strong seasonality.

**CHECK ALSO**

[Time Series Forecasting: Data, Analysis, and Practice](https://neptune.ai/blog/time-series-forecasting)

Autocorrelation

Let’s move on to the second type of temporal information that can be present in time series data: **autocorrelation**.

Autocorrelation is the correlation between a time series’ current value with past values. If this is the case, you can use present values to better predict future values.

Autocorrelation can be positive or negative:

Two famous graphs can help you detect autocorrelation in your dataset:

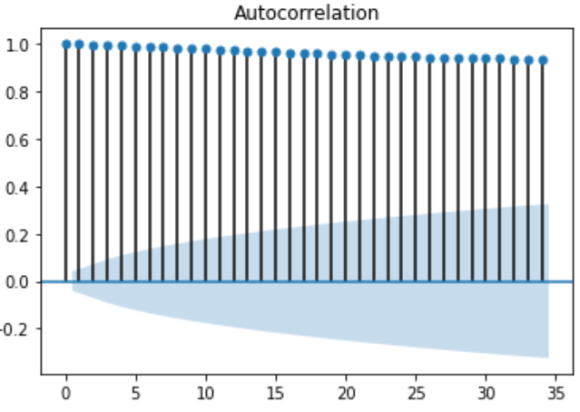
* ACF plot
* PACF plot.

**ACF: the autocorrelation function**

The autocorrelation function is a tool that helps identify whether autocorrelation exists in your time series.

**from** statsmodels.graphics.tsaplots **import** plot\_acf

plot\_acf(co2\_data)



On the x-axis, you can see the time steps (back in time). This is also called the **number of lags**. On the y-axis you can see the amount of correlation of every time step with ‘present’ time. Its obvious that there is significant autocorrelation in this plot.

**PACF: the autocorrelation function**

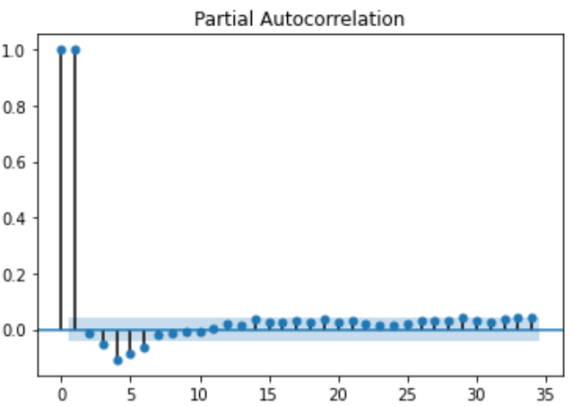
The PACF is an alternative to the ACF. Rather than giving the autocorrelations, it gives you the partial autocorrelation. This autocorrelation is called partial, because with each step back in the past, only additional autocorrelation is listed. This is different from the ACF, as the ACF contains duplicate correlations when variability can be explained by multiple points in time.

For example, If the value of today is the same as the value of yesterday, but also the same as the day-before-yesterday, the ACF would show two highly correlated steps. The PACF would only show yesterday and remove the day before yesterday.

You can compute a PACF plot using Python as follows:

**from** statsmodels.graphics.tsaplots **import** plot\_pacf

plot\_pacf(co2\_data)



You can see below that this PACF plot gives a much better representation of the autocorrelation in the CO2 data. There is a strong positive autocorrelation with lag 1: a high value now means that you are very likely to observe a high value in the next step. Since the autocorrelation shown here is partial, you do not see any duplicate effects with earlier lags, making the PACF plot neater and clearer.

**Stationarity**

Another important definition in time series is stationarity. A stationary time series is a time series that has no trend. Some time series models are not able to deal with trends (more on this later). You can detect non-stationarity using the Dickey-Fuller Test and you can remove non-stationarity using differencing.

**Dickey-Fuller test**

The Dickey-Fuller test is a statistical hypothesis test that allows you to detect non-stationarity. You can use the following Python code to apply a Dickey-Fuller test to the CO2 data:

**from** statsmodels.tsa.stattools **import** adfuller

adf, pval, usedlag, nobs, crit\_vals, icbest = adfuller(co2\_data.co2.values)

print('ADF test statistic:', adf)

print('ADF p-values:', pval)

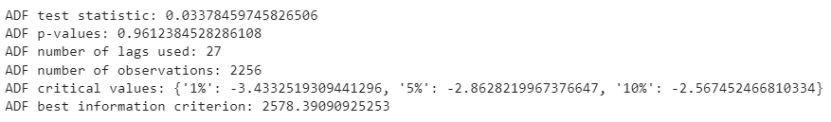
print('ADF number of lags used:', usedlag)

print('ADF number of observations:', nobs)

print('ADF critical values:', crit\_vals)

print('ADF best information criterion:', icbest)

The result looks as follows:



The null hypothesis of the ADF test is that a unit root is present in the time series. The alternative hypothesis is that the data is stationary.

The second value is the p-value. If this p-value is smaller than 0.05 you can reject the null hypothesis (reject non-stationarity) and accept the alternative hypothesis (stationarity). In this case, we cannot reject the null hypothesis and will have to assume that the data is non-stationary. As you have seen the data, you know that there is a trend, so this also confirms the result we obtained.

**Differencing**

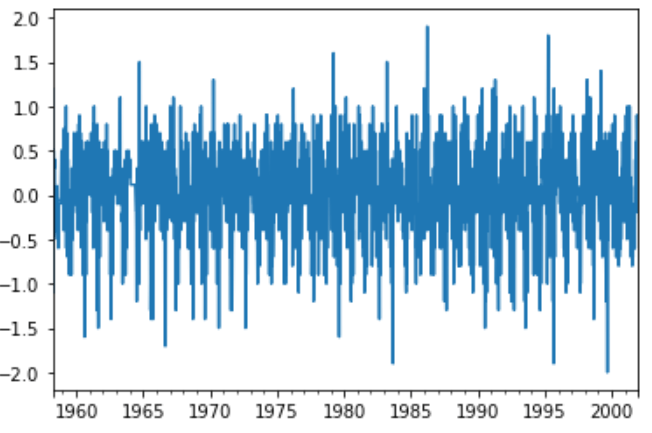
You can remove the trend from your time series. The goal is to have only seasonal variation: this can be a way to use certain models that work with seasonality but not with trends.

prev\_co2\_value = co2\_data.co2.shift()

differenced\_co2 = co2\_data.co2 - prev\_co2\_value

differenced\_co2.plot()

The differenced CO2 data looks as follows:



If you redo the ADF test on the differenced data, you will confirm that this data is now indeed stationary:

adf, pval, usedlag, nobs, crit\_vals, icbest = adfuller(differenced\_co2.dropna())

print('ADF test statistic:', adf)

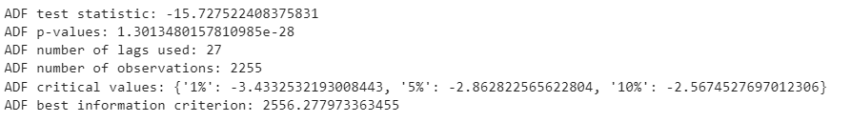
print('ADF p-values:', pval)

print('ADF number of lags used:', usedlag)

print('ADF number of observations:', nobs)

print('ADF critical values:', crit\_vals)

print('ADF best information criterion:', icbest)



The p-value is very small, indicating that the alternative hypothesis (stationarity) is true.

**One-step vs multi-step time series models**

The last concept that is important to understand before going into modeling is the concept of one-step models versus multi-step models.

Some models work great for predicting the next step for a time series, but do not have the capacity to predict multiple steps at once. These models are one-step models. You can make multi-step models with them by windowing over your predictions, but there is a risk: when using predicted values to make predictions, your errors can quickly add up and become very large.

Multistep models are models that have the intrinsic capacity to predict multiple steps at once. They are generally the better choice for long-term forecasts, and sometimes for one-step forecasts as well. It is key that you decide on the number of steps that you want to predict before starting to build models. This purely depends on your use case.

**One-step forecasts**

**Multi-step forecasts**

Designed to only forecast 1 step into the future

Designed to forecast multiple steps into the future

Can generate multi-step forecasts by windowing over predictions

No need to window over predictions

Can be less performant for multi-step forecasts

More appropriate for multi step forecasts

**Types of time series models**

Now that you have seen the main specificities of time series data, it is time to look into the types of models that can be used for predicting time series. This task is generally referred to as forecasting.

**Classical time series models**

Classical time series models are a family of models that have been traditionally used a lot in many domains of forecasting. They are strongly based on temporal variation inside a time series and they work well with univariate time series. Some advanced options exist to add external variables into the models as well. These models are generally only applicable to time series and are not useful for other types of machine learning.

**Supervised models**

Supervised models are a family of models that are used for many machine learning task. A machine learning model is supervised when it uses clearly defined input variables and one or more output (target) variables.

Supervised models can be used for time series, as long as you have a way to extract seasonality and put it into a variable. Examples include creating a variable for a year, a month, or a day of the week, etc. These are then used as the X variables in your supervised model and the ‘y’ is the actual value of the time series. You can also include lagged versions of y (the past value of y) into the X data, in order to add autocorrelation effects.

**Deep learning and recent models**

The increasing popularity of deep learning over the past years has opened new doors for forecasting as well, as specific deep learning architectures have been invented that work very well on sequence data.

Cloud computing and the popularization of AI as a service have also provided a number of new inventions in the domain. Facebook, Amazon, and other big tech companies are open-sourcing their forecasting products, or making them available on their cloud platforms. The availability of those new “black- box” models gives forecasting practitioners new tools to try and test, and can sometimes even beat previous models.

**Going deeper into classical time series models**

In this part, you will discover classical time series models in depth.

**ARIMA family**

The ARIMA family of models is a set of smaller models that can be combined. Each part of the ARMIA model can be used as a stand-alone component, or the different building blocks can be combined. When all of the individual components are put together, you obtain the SARIMAX model. You will now see each of the building blocks separately.

1. Autoregression (AR)

Autoregression is the first building block of the SARIMAX family. You can see the AR model as a regression model that explains a variable’s future value using its past (lagged) values.

The order of an AR model is denoted as p, and it represents the number of lagged values to include in the model. The simplest model is the AR(1) model: it uses only the value of the previous timestep to predict the current value. The maximum number of values that you can use is the total length of the time series (i.e. you use all previous time steps).

2. Moving average (MA)

The Moving Average is the second building block of the larger SARIMAX model. It works in a comparable way to the AR model: it uses past values to predict the current value of the variable.

The past values that the Moving Average model uses are not the values of the variable. Rather, the Moving Average uses the prediction error in previous time steps to predict the future.

This sounds counter-intuitive, but there is a logic behind it. When a model has some unknown but regular external perturbations, your model may have a seasonality or other pattern in the error of the model. The MA model is a method to capture this pattern without even having to identify where it comes from.

The MA model can use multiple steps back in time as well. This is represented in the order parameter called q. For example, an MA(1) model has an order of 1 and uses only one time step back.

3. Autoregressive moving average (ARMA)

The Autoregressive Moving Average, or ARMA, model combines the two previous building blocks into one model. ARMA can therefore use both the value and the prediction errors from the past.

ARMA can have different values for the lag of the AR and MA processes. For example an ARMA(1, 0) model has an AR order of 1 ( p = 1) and an MA order of 0 (q=0). This is actually just an AR(1) model. The MA(1) model is the same as the ARMA(0, 1) model. Other combinations are possible: ARMA(3, 1) for example has an AR order of 3 lagged values and uses 1 lagged value for the MA.

4. Autoregressive integrated moving average (ARIMA)

The ARMA model needs a stationary time series. As you have seen earlier on, stationarity means that a time series remains stable. You can use the Augmented Dickey-Fuller test to test whether your time series is stable and apply differencing if it is not the case.

The ARIMA model adds automatic differencing to the ARMA model. It has an additional parameter that you can set to the number of times that the time series needs to be differenced. For example, an ARMA(1,1) that needs to be differenced one time would result in the following notation: ARIMA(1, 1, 1). The first 1 is for the AR order, the second one is for the differencing, and the third 1 is for the MA order. ARIMA(1, 0, 1) would be the same as ARMA(1, 1).

5. Seasonal autoregressive integrated moving-average (SARIMA)

SARIMA adds seasonal effects into the ARIMA model. If seasonality is present in your time series, it is very important to use it in your forecast.

SARIMA notation is quite a bit more complex than ARIMA, as each of the components receives a seasonal parameter on top of the regular parameter.

For example, let’s consider the ARIMA(p, d, q) as seen before. In SARIMA notation, this becomes SARIMA(p, d, q)(P, D, Q)m.

m is simply the number of observations per year: monthly data has m=12, quarterly data has m=4 etc. The small letters (p, d, q) represent the non-seasonal orders. The capital letters (P, D, Q) represent the seasonal orders.

**READ ALSO**

[ARIMA & SARIMA: Real-World Time Series Forecasting (Advanced Guide)](https://neptune.ai/blog/arima-sarima-real-world-time-series-forecasting-guide)

6. Seasonal autoregressive integrated moving-average with exogenous regressors (SARIMAX)

The most complex variant is the SARIMAX model. It regroups AR, MA, differencing, and seasonal effects. On top of that, it adds the X: external variables. If you have any variables that could help your model to improve, you could add them with SARIMAX.

An example with Auto Arima in Python on CO2

Now that you have seen all the individual building blocks of the ARIMA family, it is time to apply this to an example. Let’s see if we can use the model to build a predictive model for the CO2 data.

The difficult thing with ARIMA or SARIMAX models is that you have many parameters (pp, d, q) or even (p, d, q)(P, D, Q) that you need to choose.

In some cases, you can inspect autocorrelation graphs and identify logical choices for the parameters. You can use the [statsmodels](https://www.statsmodels.org/dev/generated/statsmodels.tsa.statespace.sarimax.SARIMAX.html" \t "_blank) implementation of SARIMAX and try out performances with the parameters of your choice.

Another approach is to use an auto-arima function that automatically optimizes the hyperparameters for you. The Pyramid Python library does exactly this: it tries out different combinations and selects the one that has the best performance.

You can install Pyramid as follows:

**import** pmdarima **as** pm

**from** pmdarima.model\_selection **import** train\_test\_split

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

train, test = train\_test\_split(co2\_data.co2.values, train\_size=2200)

You then fit the model on the CO2 training data and make predictions with the best-selected model.

model = pm.auto\_arima(train, seasonal=**True**, m=52)

preds = model.predict(test.shape[0])

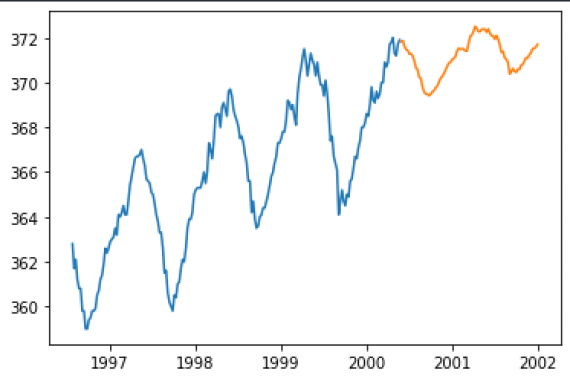
You can show them with the plot that is created here:

x = np.arange(y.shape[0])

plt.plot(co2\_data.co2.values[:2200], train)

plt.plot(co2\_data.co2.values[2200:], preds)

plt.show()



In this plot, the blue line is the actuals (training data) and the orange line is the forecast.

For more information and example on Pyramid, you can check out their [documentation](https://pypi.org/project/pmdarima/" \t "_blank).

Vector autoregression (VAR) and its derivatives VARMA and VARMAX

You could see Vector Autoregression, or VAR as a multivariate alternative to Arima. Rather than predicting one dependent variable, you predict multiple time series at the same time. This can be especially useful when there are strong relationships between your different time series. The Vector Autoregression, like the standard AR model, contains only an autoregressive component.

The**VARMA model** is the multivariate equivalent of the ARMA model. VARMA is to ARMA what VAR is to AR: it adds a Moving Average component to the model.

If you want to go a step further, you can use **VARMAX**. The X represents **external (exogenous) variables**. Exogenous variables are variables that can help your model to make better forecasts, but that do not need to be forecasted themselves. The statsmodels [VARMAX](https://www.statsmodels.org/dev/examples/notebooks/generated/statespace_varmax.html) implementation is a good way to get started with implementing VARMAX models.

More advanced versions like seasonal VARMAX (SVARMAX) exist, but they become so complex and specific that it will be hard to find implementations that do this easily and efficiently. Once models become so complex, it gets hard to understand what is happening inside the model and is often better to start looking into other familiar models.

Smoothing

Exponential Smoothing is a basic statistical technique that can be used to smoothen out time series. Time series patterns often have a lot of long term variability, but also short term (noisy) variability. Smoothening allows you to make your curve smoother so that long term variability becomes more evident and short term (noisy) patterns are removed.

This smooth version of the time series can then be used for analysis.

1. Simple moving average

The **simple moving average** is the simplest smoothing technique. It consists of replacing the current value by the average of the current and a few  past values. The exact number of past values to take into account is a parameter. The more values you use, the smoother the curve will become. At the same time, you will lose more and more variation.

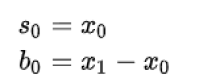
2. Simple exponential smoothing (SES)

**Exponential smoothing** is an adaptation of this simple moving average. Rather than taking the average, it takes a weighted average of past values. A value that is further back will count less and a more recent value will count more.

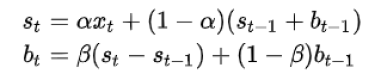
3. Double exponential smoothing

When trends are present in your time series data, you should avoid using Simple Exponential Smoothing: it does not work well in this case, as the model cannot make the distinction between variation and trend correctly. However, you can use **double exponential smoothing**.

In DES, there is a recursive application of an exponential filter. This allows you to remove trend problems. This works using the following formulas for time zero:



and the following formulas for subsequent time steps:



In which alpha is the data smoothing factor and beta is the trend smoothing factor.

4. Holt Winter’s exponential smoothing (HWES)

If you want to go even further, you can use Triple Exponential Smoothing, which is also called **Holt Winter’s exponential smoothing**. You should use this only when there are three important signals in your time series data. For example, one signal could be the trend, another one could be a weekly seasonality and a third one could be a monthly seasonality.

An example of exponential smoothing in Python

In the following example, you see how to apply a Simple Exponential Smoothing to the CO2 data. The smoothing level indicates how smooth your curve should become. In the example it is set very low, indicating a very smooth curve. Feel free to play around with this parameter and see what less smooth versions look like.

**from** statsmodels.tsa.api **import** SimpleExpSmoothing

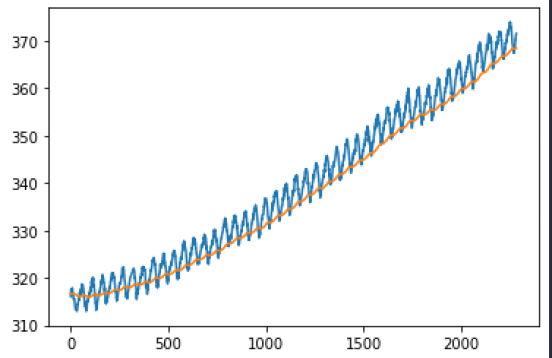
es = SimpleExpSmoothing(co2\_data.co2.values)

es.fit(smoothing\_level=0.01)

plt.plot(co2\_data.co2.values)

plt.plot(es.predict(es.params, start=0, end=**None**))

plt.show()



The blue line represents the original data and the orange line represents the smoothed curve. As it is a Simple Exponential Smoothing, it could capture only one signal: the trend.

Going deeper into supervised machine learning models

Supervised Machine Learning models work very differently than classical machine learning models. The main difference is that they consider that variables are either dependent variables or independent variables. Dependent variables, or target variables, are the variables that you want to predict. Independent variables are the variables that help you to predict.

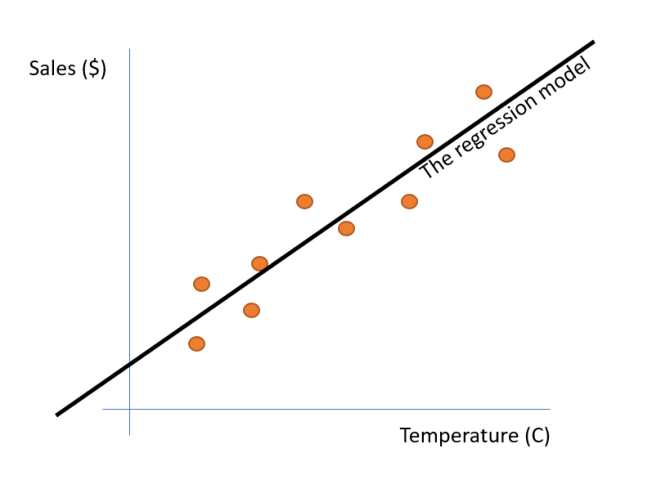
Supervised Machine Learning models are not made especially for time series data. After all, there are often no independent variables in time series data. Yet it is fairly simple to adapt them to time series by converting the seasonality (based on your time stamps for example) into independent variables.

Linear regression

Linear Regression is arguably the simplest supervised machine learning model. Linear Regression estimates linear relationships: each independent variable has a coefficient that indicates how this variable affects the target variable.

Simple Linear Regression is a Linear Regression in which there is only one independent variable. An example of a Simple Linear Regression model in non-time series data could be the following: hot chocolate sales that depend on the outside temperature (degrees Celsius).

The colder the temperature, the higher the hot chocolate sales. Visually, this could look like the graph below.



In Multiple Linear Regression, rather than using only one independent variable, you use multiple independent variables. You could imagine the 2d graph converting into a 3d graph, where the third axis represents the variable Price. In this case, you would build a Linear Model that explains the sales using temperature and price. You can add as many variables as you need.

Now, of course, this is not a time series data set: there is no time variable present. So, how could you use this technique for time series? The answer is fairly straightforward. Rather than only using temperature and price in this data set, you could add the variables year, month, day of the week, etc.

If you build a supervised model on time series, you have the disadvantage that you need to do a little bit of feature engineering to extract seasonality into variables in a way or another. An advantage is, however, that adding exogenous variables becomes much easier.

Let’s now see how to apply a Linear Regression on the CO2 dataset. You can prepare the CO2 data as follows:

**import** numpy **as** np

*# extract the seasonality data*

months = [x.month **for** x **in** co2\_data.index]

years = [x.year **for** x **in** co2\_data.index]

day = [x.day **for** x **in** co2\_data.index]

*# convert into one matrix*

X = np.array([day, months, years]).T

You then have three independent variables: day, month, and week. You could also think of other seasonality variables like the day of the week, the week number, etc., but for now, let’s go with this.

You can then use scikit-learn to build a Linear Regression model and make predictions to see what the model has learned:

**from** sklearn.linear\_model **import** LinearRegression

*# fit the model*

my\_lr = LinearRegression()

my\_lr.fit(X, co2\_data.co2.values)

*# predict on the same period*

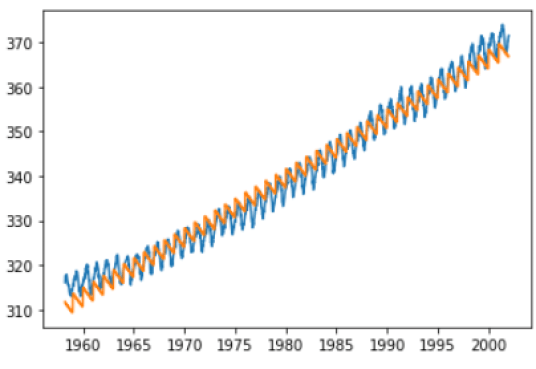
preds = my\_lr.predict(X)

*# plot what has been learned*

plt.plot(co2\_data.index, co2\_data.co2.values)

plt.plot(co2\_data.index, preds)

When using this code, you will obtain the following plot that shows a relatively good fit with the data:



Random forest

The Linear model is very limited: it can only fit linear relationships. Sometimes this will be enough, but in most cases, it is better to use more performant models. Random Forest is a much-used model that allows fitting nonlinear relationships. It is still very easy to use.

The scikit-learn library has the RandomForestRegressor that you can simply use to replace the LinearRegression in the previous code.

**from** sklearn.ensemble **import** RandomForestRegressor

*# fit the model*

my\_rf = RandomForestRegressor()

my\_rf.fit(X, co2\_data.co2.values)

*# predict on the same period*

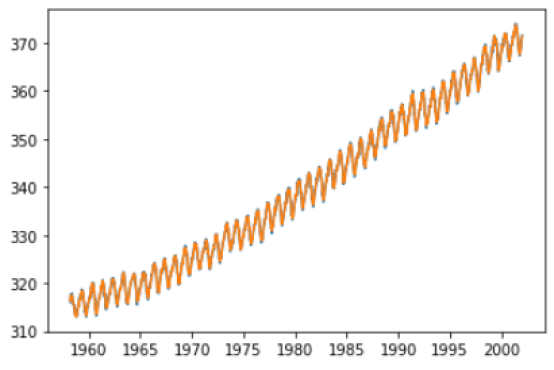
preds = my\_rf.predict(X)

*# plot what has been learned*

plt.plot(co2\_data.index, co2\_data.co2.values)

plt.plot(co2\_data.index, preds)

The fit on the training data is now even better than before:



For now, it is enough to understand that this Random Forest has been able to learn the training data better. In a later part of this article, you will see more quantitative methods for model evaluation.

**RECOMMENDED FOR YOU**

[Random Forest Regression: When Does It Fail and Why?](https://neptune.ai/blog/random-forest-regression-when-does-it-fail-and-why" \t "_blank)

XGBoost

The XGBoost model is a third model that you should absolutely know. There are many other models out there, but Random Forests and XGBoost are considered absolute classics among the supervised machine learning family.

XGBoost is a machine learning model that is based on the gradient boosting framework. This model is an ensemble model of weak learners just like the Random Forest but with an interesting advantage. In standard gradient boosting, the individual trees are fit in sequence and each consecutive decision tree is fit in such a way to minimize the error of previous trees. XGBoost obtains the same result but is still able to do parallel learning.

You can use the XGBoost package as follows:

**import** xgboost **as** xgb

my\_xgb = xgb.XGBRegressor()

my\_xgb.fit(X, co2\_data.co2.values)

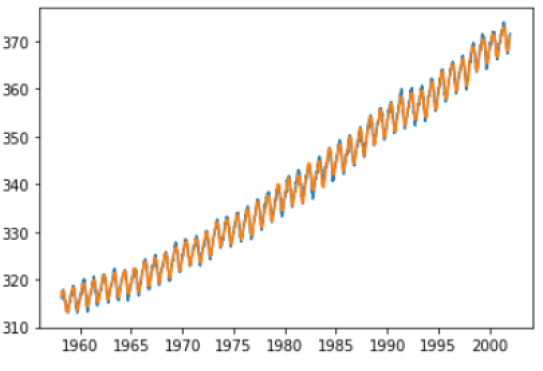
*# predict on the same period*

preds = my\_xgb.predict(X)

*# plot what has been learned*

plt.plot(co2\_data.index, co2\_data.co2.values)

plt.plot(co2\_data.index, preds)



As you can see, this model fits the data quite well too. You will learn how to do the model evaluation in the later part of this article.

**RELATED**

[How to Organize Your XGBoost Machine Learning (ML) Model Development Process: Best Practices](https://neptune.ai/blog/how-to-organize-your-xgboost-machine-learning-ml-model-development-process" \t "_blank)

Going deeper into advanced and specific time series models

In this part, you will discover two more advanced and specific time series models called GARCH and TBATS.

GARCH

GARCH stands for Generalized Autoregressive Conditional Heteroskedasticity. It is an approach to estimating the volatility of financial markets and is generally used for this use case. It is seldom used for other use cases.

The model works well for this, as it assumes an ARMA model for the error variance of the time series rather than for the actual data. In this way, you can predict variability rather than actual values.

There exist a number of variants to the GARCH family of models, for example, check [this](https://en.wikipedia.org/wiki/Autoregressive_conditional_heteroskedasticity" \t "_blank) out. This model is great to know, but should only be used when forecasting variability is the need and it is therefore relatively different from the other models that are presented in this article.

TBATS

TBATS stands for the combination of the following components:

* Trigonometric seasonality
* Box-Cox transformation
* ARMA errors
* Trend
* Seasonal components

The model was created in 2011 as a solution to forecast time series with multiple seasonal periods. As it is relatively new and relatively advanced, it is less widespread and not as much used as the models in the ARIMA family.

A useful Python implementation of TBATS can be found in Pythons [sktime](https://www.sktime.org/en/latest/api_reference/auto_generated/sktime.forecasting.tbats.TBATS.html" \t "_blank) package.

Going deeper into deep learning-based time series models

You have now seen two relatively different model families, each of them with its specific ways of fitting the models. Classical time series models are focused on relations between the past and the present. Supervised machine learning models are focused on relations between cause and effect.

You will now see three more recent models that can be used for forecasting as well. They are even more complex to apprehend and master and may (or may not) produce better results, depending on the data and the specifics of the use case.

LSTM (Long Short-Term Memory)

LSTMs are Recurrent Neural Networks. Neural Networks are very complex machine learning models that pass input data through a network. Each node in the network learns a very simple operation. The neural network consists of many such nodes. The fact that the model can use a large number of simple nodes makes the overall prediction very complex. Neural Networks can therefore fit very complex and nonlinear data sets.

RNNs are a special type of Neural network, in which the network can learn from sequence data. This can be useful for multiple use cases, including understanding time series (which are clearly sequences of values over time), but also text (sentences are sequences of words).

LSTMs are a specific type of RNNs. They have proven useful for time series forecasting on multiple occasions. They require some data and are more complicated to learn than supervised models. Once you master them, they can prove to be very powerful depending on your data and your specific use case.

To go into LSTMs, the [Keras](https://keras.io/api/layers/recurrent_layers/lstm/" \t "_blank) library in Python is a great starting point.

Prophet

Prophet is a time series library that was open-sourced by Facebook. It is a black-box model, as it will generate forecasts without much user specification. This can be an advantage, as you can almost automatically generate forecasting models without much knowledge or effort.

On the other hand, there is a risk here as well: if you do not pay close enough attention, you may very well be producing a model that seems good to the automated model building tool, but that in reality does not work well.

Extensive model validation and evaluation are recommended when using such black-box models, yet if you find that it works well on your specific use case, you may find a lot of added value here.

You can find a lot of resources on Facebook’s [GitHub](https://facebook.github.io/prophet/).

DeepAR

DeepAR is another such black-box model developed by Amazon. The functioning deep down is different, but in terms of user experience, it is relatively equal to Prophet. The idea is again to have a Python library that does all the heavy lifting for you.

Again, caution is needed, as you can never just expect any black-box model to be perfectly reliable. In the next part, you will see more on model evaluation and benchmarking, which is extremely important with such complex models. The more complex a model, the more wrong it can be!

A great and easy-to-use implementation of DeepAR is available in the [Gluon](https://ts.gluon.ai/api/gluonts/gluonts.model.deepar.html) package.

Time series model selection

In the previous part of this article, you have seen a large number of time series models, divided into classical time series models, supervised machine learning models, and recent developments including LSTMs, Prophet, and DeepAR.

The final deliverable of a time series forecasting task will be to select one model only. This has to be the model that delivers the best result for your use case. In this part of the article, you will learn how to choose one model among the huge list of potential models.

Time series models evaluation

Time series metrics

The first thing to define when selecting models is the metric that you want to look at. In the previous part, you have seen multiple fits with different qualities (think about the linear regression vs the random forest).

To go further with model selection, you will need to define a metric to evaluate your models. A very often used model in forecasting is the Mean Squared Error. This metric measures the error at each point in time and takes the square of it. The average of those squared errors is called the Mean Squared Error. An often-used alternative is the Root Mean Squared Error: the square root of the Mean Squared Error.

Another frequently used metric is the Mean Absolute Error: rather than taking the square of each error, it takes the absolute value here. The Mean Absolute Percent Error is a variation on this where the Absolute Error at each point in time is expressed as a percentage of the actual value. This yields a metric that is a percentage, which is very easy to interpret.

Time series train test split

The second thing to think about when evaluating Machine Learning is to consider that a model that works well on the training data, does not necessarily work well on new, out-of-sample data. Such a model is called an overfitting model.

There are two common approaches that can help you to estimate whether a model is generalizing correctly: the train-test-split and cross-validation.

The train test split means that you remove a part of your data before fitting the model. As an example, you could remove the last 3 years from the CO2 database and use the remaining 40 years for fitting the model. You’d then forecast the three years of test data and measure the evaluation metric of your choice between your predictions and the actual values of the last three years.

To benchmark and choose models, you could build multiple models on the 40 years of data and do the test set evaluation on all of them. Based on this test-performance, you could select the model that is most performant.

Of course, if you are building a short-term forecasting model, using three years of data would not make sense: you’d choose an evaluation period that is comparable to the period that you’d forecast in reality.

Time series cross-validation

A risk with the train test split is that you measure only at one point in time. In non-time series data, the test set is generally generated by a random selection of data points. In time series, however, this does not work in many cases: when sequences are used, you can not remove one point in the sequence and still expect the model to work.

Time series train test splits are therefore best applied by selecting the final period as the test set. The risk here is that this can go wrong if your last period is not very reliable. In recent covid periods, you can imagine that many business forecasts have gone completely off: the underlying trends have changed.

Cross-validation is a method that does a repeated train test evaluation. Rather than making one train test split, it makes multiple (the exact number is a user-defined parameter). For example, if you use 3-fold cross-validation, you will split your data set into three equal parts. You will then fit three times the same model on two-thirds of the data set and use the other third for evaluation. In the end, you have three evaluation scores (each on a different test set) and you can use the average as the final metric.

**SEE ALSO**

[Cross-Validation in Machine Learning: How to Do It Right](https://neptune.ai/blog/cross-validation-in-machine-learning-how-to-do-it-right)

By doing this, you avoid selecting a model that works on the test set by chance: you have now made sure that it works on multiple test sets.

In time series, however, you cannot apply a random selection to obtain multiple test sets. If you’d do this, you’d end up with sequences with many missing data points.

A solution can be found in time series cross-validation. What it does is create multiple train test sets, but each of the test sets is the end of the period. For example, the first train test split could be built on the first 10 years of data (5 train, 5 test). The second model would be done on the first 15 years of data (10 train, 5 test), etc. This can work well but has the disadvantage that each of the models does not use the same number of years in the training data.

An alternative is to do a rolling split (always 5 years train, 5 years test), but here the disadvantage is that you can never use more than 5 years for training data.

Time series model experiments

In conclusion, when doing time series model selection, the following questions are key to define before starting to experiment:

* Which metric are you using?
* Which period do you want to forecast?
* How to make sure your model works on the future data points that have not been seen by the model?

Once you have the answer to the aforementioned questions, you can start trying out different models, and use the defined evaluation strategy to select and improve models.

An example use case for time series modeling

In this part, you will work on a forecast for the next day of the S&P 500. You could imagine running your model every night and then the next day you would know if the stock market is going up or down. If you have a very accurate model for doing this, you could easily make a lot of money (don’t treat it as financial advice ;)).

Stock market forecasting data and definition of the evaluation method

Obtaining stock market data

You can use the Yahoo Finance package in Python to automatically download stock data.

!pip install yfinance

**import** yfinance **as** yf

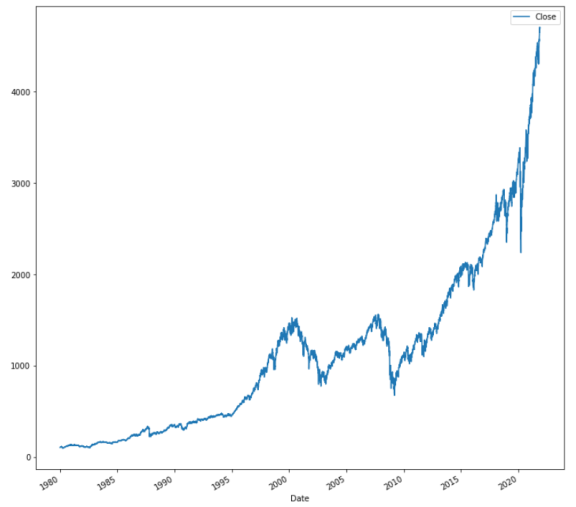
*# taking the close price (end of day)*

sp500\_data = yf.download('^GSPC', start="1980-01-01", end="2021-11-21")

sp500\_data = sp500\_data[['Close']]

sp500\_data.plot(figsize=(12, 12))

You can see the evolution of the S&P500 closing prices since 1980 in the plot:



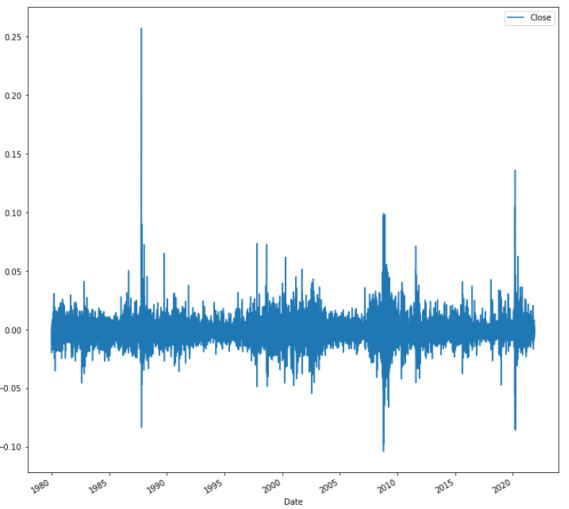
*The evolution of the S&P500 closing prices since 1980 | Source: Author*

With stock data, the absolute price is not actually that relevant. What is more interesting for stock traders is to know whether prices are going up or down, and by what percentage. You can change the data into percentage increase or decrease as follows:

difs = (sp500\_data.shift() - sp500\_data) / sp500\_data

difs = difs.dropna()

difs.plot(figsize=(12, 12))



*Plot of the percentage difference of the S&P | Source: Author*

Defining the experimental approach

The goal of the models will be to have the best possible prediction of the change in the stock price of the next day. It is necessary to decide on an approach so that you can automate the process a little bit here.

As you want to predict for one day only, you can understand that the test set will be very small (one day only). Therefore, it would be best to create a lot of test splits to make sure that there is an acceptable amount of model evaluation.

This can be obtained by the Time Series Split that was explained earlier. For example, you can set up a Time Series Split that will make 100 train test sets, in which each train test set uses three months of training data and one day of test data. This will work fine for this example to understand the principle of model selection in time series.

Building a classical time series model

Let’s start with a classical time series model on this problem: the Arima model. In this code, you will set up the automated creation of Arima models with orders ranging from (0, 0, 0) to (4, 4, 4). Each of the models will be built and evaluated using a Time Series Split with 100 splits, in which the train size is a maximum of three months and the test size is always one day.

Because of the sheer number of runs involved, the results are logged to Neptune.ai for ease of comparison. For following along, you can set up a free account and get more info [here](https://docs.neptune.ai/getting-started/hello-world" \t "_blank).

**import** numpy **as** np

**from** sklearn.metrics **import** mean\_squared\_error

**from** sklearn.model\_selection **import** TimeSeriesSplit

**import** neptune.new **as** neptune

param\_list = [(x, y, z) **for** x **in** range(5) **for** y **in** range(5) **for** z **in** range(5)]

**for** order **in** param\_list:

run = neptune.init(

project="YOU/YOUR\_PROJECT",

api\_token="YOUR\_API\_TOKEN",

)

run['order'] = order

*# for each param combi do a ts split*

*# max 3 months training data*

*# 1 day test size*

mses = []

tscv = TimeSeriesSplit(n\_splits=100,

max\_train\_size = 3\*31,

test\_size=1)

**for** train\_index, test\_index **in** tscv.split(y):

**try**:

train = y[train\_index]

test = y[test\_index]

*# for each ts split do a model*

mod = sm.tsa.ARIMA(train, order=order)

res = mod.fit()

pred = res.forecast(1)[0]

mse = mean\_squared\_error(test, pred)

mses.append(mse)

**except**:

*# ignore models that error*

**pass**

**try**:

average\_mse = np.mean(mses)

std\_mse = np.std(mses)

run['average\_mse'] = average\_mse

run['std\_mse'] = std\_mse

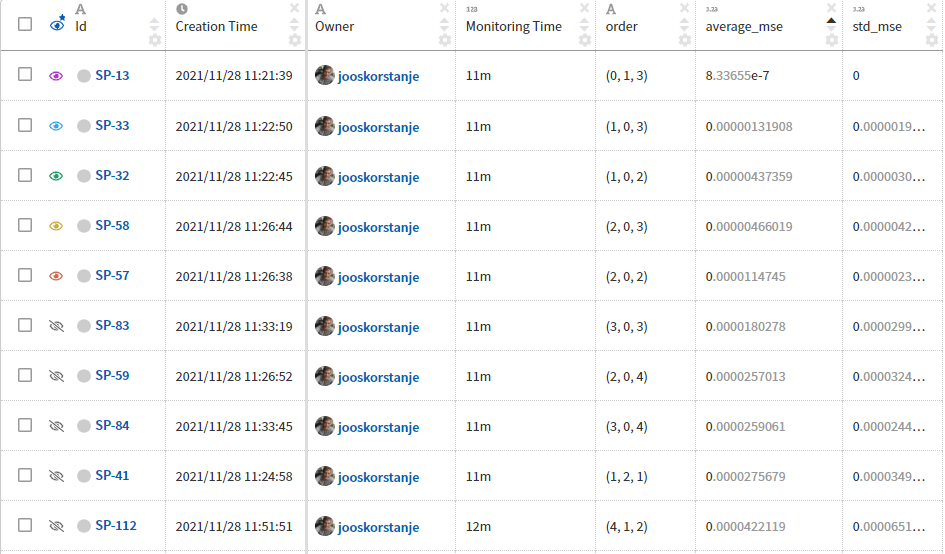
**except**:

run['average\_mse'] = **None**

run['std\_mse'] = **None**

run.stop()

You can see the results in a table format:

[](https://i0.wp.com/neptune.ai/wp-content/uploads/How-to-Select-a-Model-For-Your-Time-Series-Prediction-Task_7.png?ssl=1)

[*Results in the Neptune UI | Source: Author*](https://app.neptune.ai/jooskorstanje/sp500/experiments?split=tbl&dash=leaderboard&viewId=standard-view)

The model with the lowest average MSE is the model with order (0, 1, 3). However, as you can see, the standard deviation of this model is suspiciously 0. The next two models in the line are ARIMA(1, 0, 3) and ARIMA(1, 0, 2). They are very similar and this would indicate that the result is reliable. The best guess here would be to take the ARIMA(1, 0, 3) as the best model which has an average MSE of 0.00000131908 and an average standard deviation of 0.00000197007.

Building a supervised machine learning model

Let’s now move on to a supervised model and find out whether the performances differ from the classical time series model.

In supervised machine learning for forecasting, there is a decision to make on feature engineering. As explained earlier in the article, supervised models use dependent variables (the ones to predict) and independent variables (the predictors).

In some use cases, you may have a lot of data about the future. For example, if you want to predict the number of customers of a restaurant, you could use external data on the number of reservations that have been made for future dates as independent variables.

For the current stock market use case, you do not have those data: you only have the stock price over a period of time. However, supervised models cannot be built using only a target variable. You’ll need to find a way to extract seasonality from the data and use feature engineering to create independent variables. As the stock market is known to have a lot of autocorrelation effects, let’s try a model that uses the values of the past 30 days as predictor variables to predict the 31st day.

You can create a data set that has all of the possible combinations of 30 training days and 1 test day (always consecutive) of the S&P500 and you would be able to create an enormous training database this way:

**import** yfinance **as** yf

sp500\_data = yf.download('^GSPC', start="1980-01-01", end="2021-11-21")

sp500\_data = sp500\_data[['Close']]

difs = (sp500\_data.shift() - sp500\_data) / sp500\_data

difs = difs.dropna()

y = difs.Close.values

*# window through the data*

X\_data = []

y\_data = []

**for** i **in** range(len(y) - 31):

X\_data.append(y[i:i+30])

y\_data.append(y[i+30])

X\_windows = np.vstack(X\_data)

Now that you have the training database, you can use regular cross-validation: after all, the rows of the data set can be used independently. They are all sets of 30 training days and 1 ‘future’ test day. Thanks to this data preparation, you can use regular KFold cross-validation.

**import** numpy **as** np

**import** xgboost **as** xgb

**from** sklearn.model\_selection **import** KFold

**import** neptune.new **as** neptune

**from** sklearn.metrics **import** mean\_squared\_error

*# specify the grid for the grid search of hyperparameter tuning*

parameters={'max\_depth': list(range(2, 20, 4)),

'gamma': list(range(0, 10, 2)),

'min\_child\_weight' : list(range(0, 10, 2)),

'eta': [0.01,0.05, 0.1, 0.15,0.2,0.3,0.5]

}

param\_list = [(x, y, z, a) **for** x **in** parameters['max\_depth'] **for** y **in** parameters['gamma'] **for** z **in** parameters['min\_child\_weight'] **for** a **in** parameters['eta']]

**for** params **in** param\_list:

mses = []

run = neptune.init(

project="YOU/YOUR\_PROJECT",

api\_token="YOUR\_API\_TOKEN",

)

run['params'] = params

my\_kfold = KFold(n\_splits=10, shuffle=**True**, random\_state=0)

**for** train\_index, test\_index **in** my\_kfold.split(X\_windows):

X\_train, X\_test = X\_windows[train\_index], X\_windows[test\_index]

y\_train, y\_test = np.array(y\_data)[train\_index], np.array(y\_data)[test\_index]

xgb\_model = xgb.XGBRegressor(max\_depth=params[0],gamma=params[1], min\_child\_weight=params[2], eta=params[3])

xgb\_model.fit(X\_train, y\_train)

preds = xgb\_model.predict(X\_test)

mses.append(mean\_squared\_error(y\_test, preds))

average\_mse = np.mean(mses)

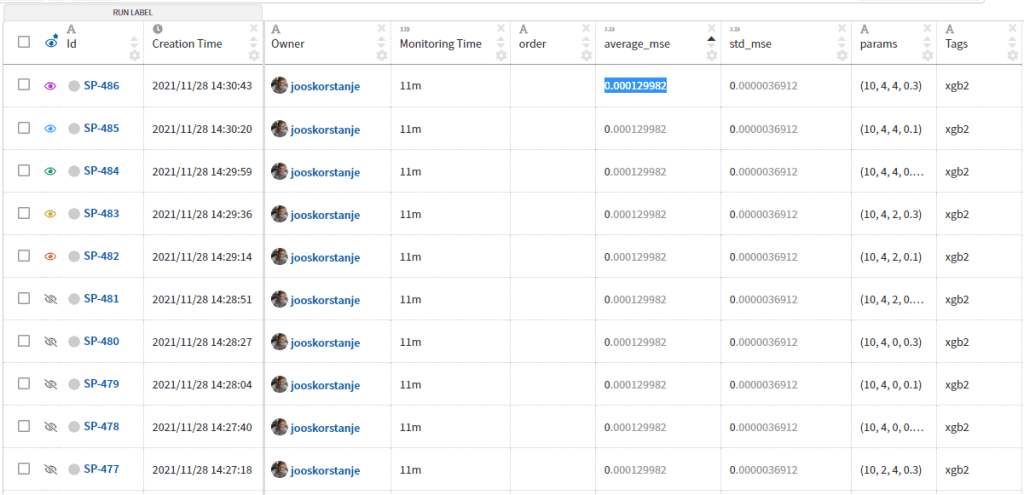
std\_mse = np.std(mses)

run['average\_mse'] = average\_mse

run['std\_mse'] = std\_mse

run.stop()

Some of the scores that were obtained using this loop are shown in the below table:

[](https://i0.wp.com/neptune.ai/wp-content/uploads/How-to-Select-a-Model-For-Your-Time-Series-Prediction-Task_21.png?ssl=1)

[*Results in the Neptune UI | Source: Author*](https://app.neptune.ai/jooskorstanje/sp500/experiments?split=tbl&dash=leaderboard&viewId=standard-view&query=(Tags%3Astring%20CONTAINS%20%22xgb%22)&sortBy=%5B%22sys%2Fcreation_time%22%5D&sortFieldType=%5B%22datetime%22%5D&sortFieldAggregationMode=%5B%22auto%22%5D&sortDirection=%5B%22descending%22%5D&suggestionsEnabled=true&lbViewUnpacked=true)

The parameters that were tested in this GridSearch are presented in below table:

**Parameter name**

**Values tested**

**Description**

Max Depth

2, 4, 6 8, 10

The deeper the trees, the more complex they are. Setting this parameter can help you to avoid having a too complex (overfitting) model

Min Child Weight

0, 2, 4

If the tree splitting creates a node with a sum that is lower than this value, the model will stop splitting. This is another way to avoid overly complex models

Eta

0.01, 0.1, 0.3

Stepsize of the optimization that is used to prevent overfitting

Gamma

0, 2, 4

Minimum loss reduction that allows further splitting of a node: the higher this value, the less splits will be made in the trees

For more reading on XGBoost tuning check the official XGBoost documentation on the topic over [here](https://xgboost.readthedocs.io/en/latest/tutorials/param_tuning.html" \t "_blank).

The best (lowest) MSE obtained by this XGBoost is 0.000129982. There are multiple hyperparameter combinations that obtain this score. As you can see, the XGBoost model is much less performant than the classical time series model, at least in the current configuration. Another method of organizing the data may be necessary to get better results from XGBoost.

Building a deep learning-based time series model

As a third model for the model comparison, let’s take an LSTM and see whether this can beat the ARIMA model. You can do a model comparison using cross-validation as well. However, this can be fairly long to run. In this case, you see how a train/test split has been used instead.

You can build the LSTM using the following code:

**import** yfinance **as** yf

sp500\_data = yf.download('^GSPC', start="1980-01-01", end="2021-11-21")

sp500\_data = sp500\_data[['Close']]

difs = (sp500\_data.shift() - sp500\_data) / sp500\_data

difs = difs.dropna()

y = difs.Close.values

*# create windows*

X\_data = []

y\_data = []

**for** i **in** range(len(y) - 3\*31):

X\_data.append(y[i:i+3\*31])

y\_data.append(y[i+3\*31])

X\_windows = np.vstack(X\_data)

*# create train test split*

**from** sklearn.model\_selection **import** train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_windows, np.array(y\_data), test\_size=0.2, random\_state=1)

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X\_train, y\_train, test\_size=0.25, random\_state=1)

*# build LSTM using tensorflow keras*

**from** sklearn.model\_selection **import** GridSearchCV

**import** numpy **as** np

**import** xgboost **as** xgb

**from** sklearn.model\_selection **import** KFold

**import** neptune.new **as** neptune

**from** sklearn.metrics **import** mean\_squared\_error

archi\_list = [

[tf.keras.layers.LSTM(32, return\_sequences=**True**, input\_shape=(3\*31,1)),

tf.keras.layers.LSTM(32, return\_sequences=**True**),

tf.keras.layers.Dense(units=1)

],

[tf.keras.layers.LSTM(64, return\_sequences=**True**, input\_shape=(3\*31,1)),

tf.keras.layers.LSTM(64, return\_sequences=**True**),

tf.keras.layers.Dense(units=1)

],

[tf.keras.layers.LSTM(128, return\_sequences=**True**, input\_shape=(3\*31,1)),

tf.keras.layers.LSTM(128, return\_sequences=**True**),

tf.keras.layers.Dense(units=1)

],

[tf.keras.layers.LSTM(32, return\_sequences=**True**, input\_shape=(3\*31,1)),

tf.keras.layers.LSTM(32, return\_sequences=**True**),

tf.keras.layers.LSTM(32, return\_sequences=**True**),

tf.keras.layers.Dense(units=1)

],

[tf.keras.layers.LSTM(64, return\_sequences=**True**, input\_shape=(3\*31,1)),

tf.keras.layers.LSTM(64, return\_sequences=**True**),

tf.keras.layers.LSTM(64, return\_sequences=**True**),

tf.keras.layers.Dense(units=1)

],

]

**for** archi **in** archi\_list:

run = neptune.init(

project="YOU/YOUR\_PROJECT",

api\_token="YOUR\_API\_TOKEN",

)

run['params'] = str(len(archi) - 1) + ' times ' + str(archi[0].units)

run['Tags'] = 'lstm'

lstm\_model = tf.keras.models.Sequential(archi)

lstm\_model.compile(loss=tf.losses.MeanSquaredError(),

optimizer=tf.optimizers.Adam(),

metrics=[tf.metrics.MeanSquaredError()]

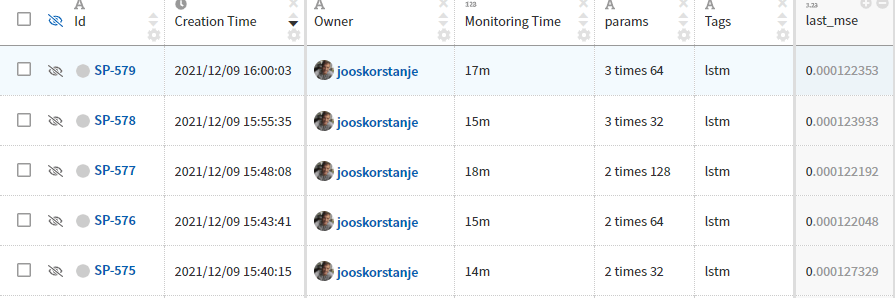
)

history = lstm\_model.fit(X\_train, y\_train, epochs=10, validation\_data=(X\_val, y\_val))

run['last\_mse'] = history.history['val\_mean\_squared\_error'][-1]

run.stop()

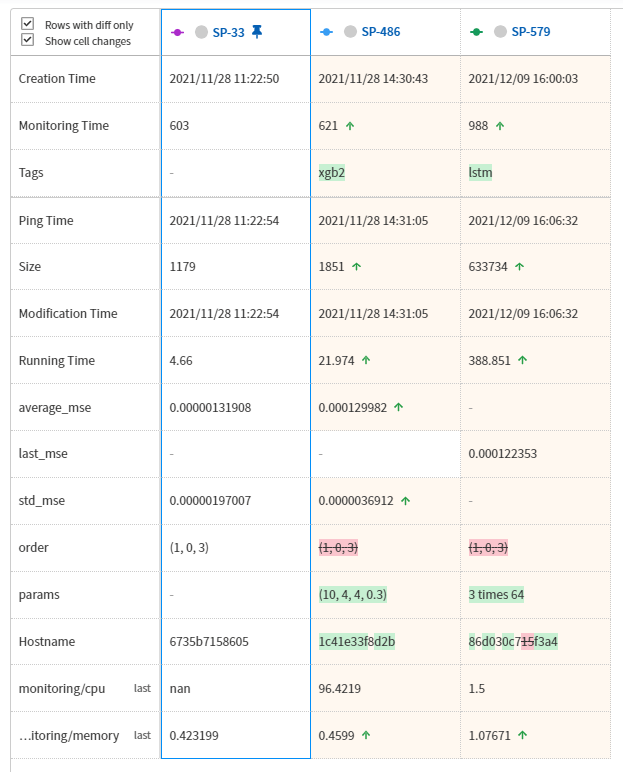
You will see the following output for the 10 epochs:

[](https://i0.wp.com/neptune.ai/wp-content/uploads/How-to-Select-a-Model-For-Your-Time-Series-Prediction-Task_23.png?ssl=1)[*Results in the Neptune UI | Source: Author*](https://app.neptune.ai/jooskorstanje/sp500/experiments?split=tbl&dash=leaderboard&viewId=standard-view&query=(Tags%3Astring%20CONTAINS%20%22lstm%22)&sortBy=%5B%22sys%2Fcreation_time%22%5D&sortFieldType=%5B%22datetime%22%5D&sortFieldAggregationMode=%5B%22auto%22%5D&sortDirection=%5B%22descending%22%5D&suggestionsEnabled=true&lbViewUnpacked=true)

The LSTM performed the same as the XGBoost model. Again, there can be multiple things to tune further if you’d want to work more on this. You could think about using longer or shorter training periods. You may also want to work on standardizing the data differently: this often plays a role in neural network performances.

Selecting the best model

As a conclusion of this case study, you could say that the best performance was obtained by the ARIMA model. This has been based on using comparative data for each: three months training period and a one-day forecast.

[](https://i0.wp.com/neptune.ai/wp-content/uploads/Time-series-models.png?ssl=1)[*Comparison of 3 models in the Neptune UI | Source: Author*](https://app.neptune.ai/jooskorstanje/sp500/experiments?compare=MzA0BYFYwTmI&split=cmp&dash=leaderboard&viewId=standard-view&query=&sortBy=%5B%22sys%2Fcreation_time%22%5D&sortFieldType=%5B%22datetime%22%5D&sortFieldAggregationMode=%5B%22auto%22%5D&sortDirection=%5B%22descending%22%5D&suggestionsEnabled=true&lbViewUnpacked=true)

Next steps

If you want to take this model further, there are a lot of things that you could improve. For example, you could try working with longer or shorter training periods. You could also try adding additional data, like seasonal data (day of the week, month, etc.) or additional predictor variables like market sentiment or others. In this case, you would need to switch to the SARIMAX model.

I hope that this article has shown you how to go about model selection in the case of time series data. You have now got an idea of the different models and model categories that could be interesting to work with. You have also seen the tools like windowing and the time series split that are specific to time series model evaluation.

For more advanced reading, I suggest the following sources:

* Forecasting directional movements of stock prices for intraday trading using LSTM and random forests. Authors: Pushpendu Ghosh, Ariel Neufeld, Jajati Keshari Sahoo. <https://arxiv.org/pdf/2004.10178v2.pdf>
* Forecasting S&P 500 Stock Index Using Statistical Learning Models. Authors: Chongda Liu, Jihua Wang, Di Xiao, Qi Liang. <https://pdfs.semanticscholar.org/8a20/9a264c16b7826bac3a234d1bc839c82396d3.pdf>
* Comparison of Financial Models for Stock Price Prediction. Authors: Mohammad Rafiqul Islam and Nguyet Nguyen. <https://www.mdpi.com/1911-8074/13/8/181/htm>