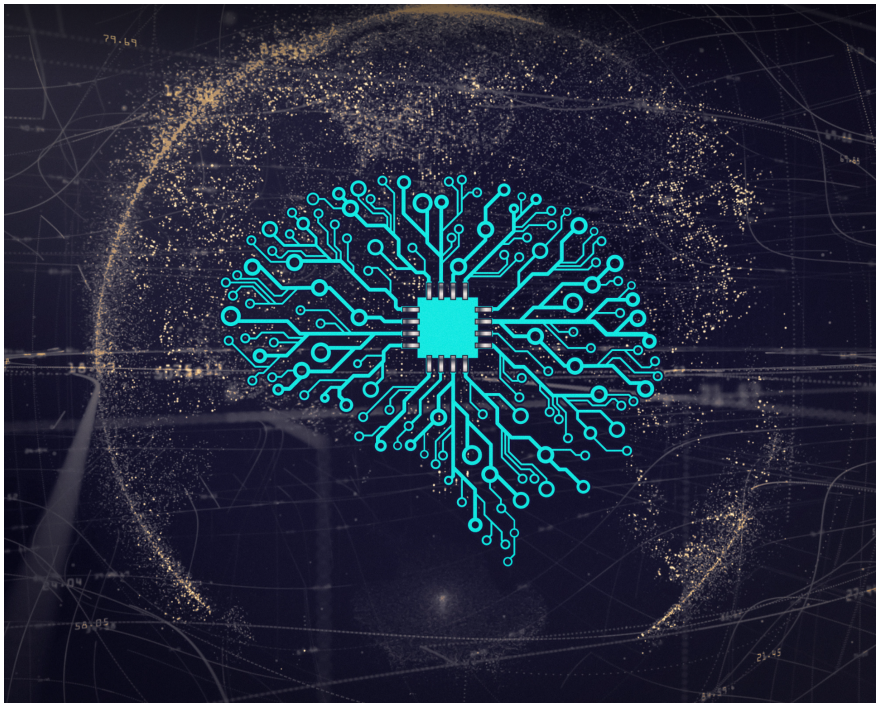
Machine Learning for Supply Chain Forecasting

How can you use Machine Learning to forecast demand in supply chains?



Source: https://www.vpnsrus.com/

. . .

The article below is an extract from my book Data Science for Supply Chain Forecast, available here. You can find my other publications here. I am also active on LinkedIn.

. . .

What is machine learning?

Supply chain practitioners usually use old-school statistics to predict demand. But with the recent rise of machine learning algorithms, we have new tools at our disposal that can easily achieve excellent performance in terms of forecast accuracy for a typical industrial demand dataset. These models will be able to learn many relationships that are beyond the ability of traditional statistical models. For

example, how to add external information (such as the weather) to a forecast model.

Traditional statistical models use a predefined model to populate a forecast based on historical demand. The issue is that these models couldn't adapt to historical demand. If you use a double exponential smoothing model to predict a seasonal product, it will fail to interpret the seasonal patterns. On the other hand, If you use a triple exponential smoothing model on a non-seasonal demand, it might overfit the noise of the demand and interpret it as a seasonality.

Machine learning is different: here, the algorithm (i.e., the machine) will learn relationships from a training dataset (i.e., our historical demand) and then be able to apply these relationships on new data. Whereas a traditional statistical model will use a predefined relationship (model) to forecast the demand, a machine learning algorithm will not assume *a priori* a particular relationship (like seasonality or a linear trend); it will **learn** these patterns directly from the historical demand.

For a machine learning algorithm to learn how to make predictions, we will have to show it both the inputs and the desired respective outputs. It will then automatically understand the relationships between these inputs and outputs.

Another critical difference between using machine learning and exponential smoothing models to forecast our demand is the fact that a machine learning algorithm will **learn patterns from all our dataset**. Exponential smoothing models will treat each item individually, independently of the others. A machine learning algorithm will learn patterns from all the dataset and will apply what works best to each product. One could improve the accuracy of an exponential smoothing model by increasing the length of each time series (i.e., providing more historical periods for each product). With machine learning, we will be able to increase the accuracy of our model by providing more products.

Welcome to the world of machine learning.

Machine learning for demand forecast

In order to make a forecast, the question we will ask the machine learning algorithm is the following:
Based on the last *n* periods of demand, what will the demand be during the next period(s)?

We will train the model by providing it the data with a specific layout:
- *n* consecutive periods of demand as input.
- the demand for the very next period(s) as output.
Let's see an example (with a quarterly forecast to simplify the table):

Product	Inputs				Output
	Q1	Q2	Q3	Q4	Q1 Y+1
#1	5	15	10	7	6
#2	7	2	3	1	4
#3	18	25	32	47	56
#4	4	1	5	3	3

For our forecast problem, we will basically show our machine learning algorithm different extracts of our historical demand dataset as inputs and each time show what the very next demand observation was. In our example above, the algorithm will learn the relationship between the last four quarters of demand and the demand for the next quarter. The algorithm will **learn** that if we have 5, 15, 10 & 7 as the last four demand observations, the next demand observation will be 6, so that its prediction should be 6.

Most people will react to this idea with two very different thoughts. Either people will think that *"it is simply impossible for a computer to look at the demand and make a prediction"* or that *"as of now, the humans have nothing left to do."* Both are wrong.

As we will see later, machine learning can generate very accurate predictions. And as the human controlling the machine, we still have to ask ourselves many questions:
- Which data to feed the algorithm for it to understand the proper relationships.
- Which machine learning algorithm to use (there are many different ones!).
- Which parameters to use in our model. As you will see, each machine learning algorithm has some settings that we can tweak to improve its accuracy.

As always, there is no definitive one-size-fits-all answer.
Experimentation will help you find what is best for your dataset.

Data preparation

The first step of any machine learning algorithm project is to clean and format the data correctly. In our case, we need to format the historical demand dataset to obtain one similar to the table shown above.

Naming convention During our data cleaning process, we will use the standard data science notation and call the inputs **X** and the outputs **Y**. Specifically, the datasets **X_train** & **Y_train** will contain all the historical demand we will use to train our algorithm (**X_train** being the inputs and **Y_train** the outputs). And the datasets **X_test** & **Y_test** will be used to **test** our model.

You can see on the table below an example of a typical historical demand dataset you should have at the beginning of a forecast project.

Product	Y1				Y2				Y3			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
#1	5	15	10	7	6	13	11	5	4	11	9	4
#2	7	2	3	1	1	0	0	1	3	2	4	5
#3	18	25	32	47	56	70	64	68	72	67	65	58
#4	4	1	5	3	2	5	3	1	4	3	2	5

We now have to format this dataset to something similar to the first table. Let's say for now that we want to predict the demand for a product during one quarter based on the demand observations of this product during the previous four quarters. We will populate the datasets **X_train** & **Y_train** by going through the different products we have and each time create a data sample with four consecutive quarters as **X_train** and the next quarter as **Y_train**. This way, the machine learning algorithm will learn the relationship(s) between one quarter of demand and the previous four.

You can see on the table below an illustration for the first iterations. To validate our model, we will keep Y3Q4 aside as a test set.

Loop	Product	Y1				Y2				Y3			
		Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
#1	#1	5	15	10	7	6							
#1	#2	7	2	3	1	1							
#1	#3	18	25	32	47	56							
#1	#4	4	1	5	3	2							
#2	#1		15	10	7	6	13						
#2	#2		2	3	1	1	0						
#2	#3		25	32	47	56	70						
#2	#4		1	5	3	2	5						
#3	#1			10	7	6	13	11					
...	...			...	...	...	...	...					

This means that our training set won't go until Y3Q4 as **it is kept for the test set**: the last loop will be used as a final test.

Our **X_train** and **Y_train** datasets will look like the table below:

Loop	Product	X_train					Y_train
#1	#1	5	15	10	7	→	6
#1	#2	7	2	3	1	→	1
#1	#3	18	25	32	47	→	56
#1	#4	4	1	5	3	→	2
#2	#1	15	10	7	6	→	13
#2	#2	2	3	1	1	→	0
#2	#3	25	32	47	56	→	70
...	...	...	...	...	...	→	...

Remember that our algorithm will learn relationships in X_train to predict Y_train. So we could write that as X_train -> Y_train.

The final test will be given to our tool via these X_test & Y_test datasets:

X_test				Y_test
5	4	11	9	4
1	3	2	4	5
68	72	67	65	58
1	4	3	2	5

These are each time the four latest demand quarters we know for each item just before Y3Q4 (i.e., Y2Q4 to Y3Q3). That means that our algorithm won't see these relationships during its training phase as it will be tested on the accuracy it achieved on these specific prediction exercises. We will measure its accuracy on this test set and assume its accuracy when predicting future demand will be similar.

Dataset length

It is important for any machine learning exercise to pay attention to how much data is fed to the algorithm. The more, the better. On the other hand, the more periods we use to make a prediction (we will call this **x_len**), the less we will be able to loop through the dataset. Also, if we want to predict more periods at once (**y_len**), it will cost us a part of

the dataset, as we need more data (Y_train is longer) to perform one loop in our dataset.

Typically, if we have a dataset with *n* periods, we will be able to make *1+n-x_len-y_len* runs through it.

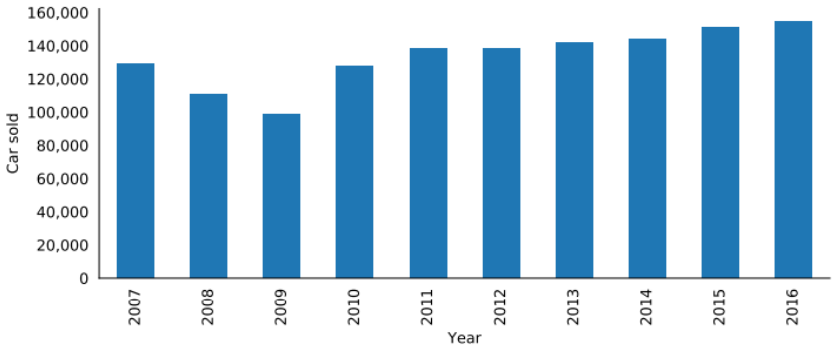
$$loops = 1 + n - x\_len - y\_len$$

It is a best practice to keep at the very least enough runs to loop through two full years so that $23 + x\_len + y\_len \leq n$. This means that the algorithm will have two full seasonal cycles to learn any possible relationships. If it had just one, you would be facing high risks of overfitting.

Do It Yourself

Data collection

The dataset creation and cleaning is an essential part of any data science project. In order to illustrate all the models we will create in the next chapters, we will use the historical sales of cars in Norway from January 2007 to January 2017 as an example dataset. You can download this dataset here: www.supchains.com/download
You will get a *csv* file called *norway_new_car_sales_by_make.csv*. This dataset contains the sales of 65 car makers across 121 months. On average, a bit more than 140,000 new cars are sold in Norway per year so that the market can then be roughly estimated to be worth 4B$ if we assume that the price of a new car is, on average, around 30,000$ in Norway. This dataset is modest in terms of size, but it is big enough to be relevant to experiment with new models and ideas. Nevertheless, machine learning models might show better results on other bigger datasets.



Bring Your Own Data Set In the second part of the article, we will discuss how to apply a machine learning model to this example dataset. But what we are actually interested in is **your own dataset**. Do not

waste any time and already start to gather some historical demand data so that you can test the following models on your own historical demand data as we progress through the different topics. It is recommended that you start with a dataset with at least three years of data (5 would be better) and more than a hundred different products. The bigger, the better.

Training and test sets creation

We will make a first code to extract the data from this *csv* and format it with the dates as columns and the products as lines.

```
# Load the CSV file (should be in the same directory)
data = pd.read_csv("norway_new_car_sales_by_make.csv")

# Create a column "Period" with both the Year and the Month
data["Period"] = data["Year"].astype(str) + "-" +
data["Month"].astype(str)
# We use the datetime formatting to make sure format is
consistent
data["Period"] =
pd.to_datetime(data["Period"]).dt.strftime("%Y-%m")

# Create a pivot of the data to show the periods on columns
and the car makers on rows
df = pd.pivot_table(data=data, values="Quantity",
index="Make", columns="Period", aggfunc='sum', fill_value=0)

# Print data to Excel for reference
df.to_excel("Clean Demand.xlsx")
```

Note that we print the results in an excel file for later reference. It is always good practice to visually check what the dataset looks like to be sure the code worked as intended.

You can also define a function to store these steps for later use.

```
def import_data():
 data = pd.read_csv("norway_new_car_sales_by_make.csv")
 data["Period"] = data["Year"].astype(str) + "-" +
data["Month"].astype(str)
 data["Period"] =
pd.to_datetime(data["Period"]).dt.strftime("%Y-%m")
 df =
pd.pivot_table(data=data,values="Quantity",index="Make",colu
mns="Period",aggfunc='sum',fill_value=0)
 return df
```

Now that we have our dataset with the proper formatting, we can create our training and test sets. For this purpose, we will create a function **datasets** that takes as inputs:

df our initial historical demand;
x_len the number of months we will use to make a prediction;
y_len the number of months we want to predict;
y_test_len the number of months we leave as a final test;

and returns X_train, Y_train, X_test & Y_test.

```
def datasets(df, x_len=12, y_len=1, y_test_len=12):
```

```
D = df.values
 periods = D.shape[1]

 # Training set creation: run through all the possible time
windows
 loops = periods + 1 — x_len — y_len — y_test_len
 train = []
 for col in range(loops):
 train.append(D[:,col:col+x_len+y_len])
 train = np.vstack(train)
 X_train, Y_train = np.split(train,[x_len],axis=1)

 # Test set creation: unseen “future” data with the demand
just before
 max_col_test = periods — x_len — y_len + 1
 test = []
 for col in range(loops,max_col_test):
 test.append(D[:,col:col+x_len+y_len])
 test = np.vstack(test)
 X_test, Y_test = np.split(test,[x_len],axis=1)

 # this data formatting is needed if we only predict a
single period
 if y_len == 1:
 Y_train = Y_train.ravel()
 Y_test = Y_test.ravel()

 return X_train, Y_train, X_test, Y_test
```

In our function, we have to use **.ravel()** on both Y_train and Y_test if we only want to predict one period at a time.
array.ravel() reduces the dimension of a NumPy array to 1D.
Our function always creates Y_train and Y_test as 2D arrays (i.e., arrays with rows and columns). If we only want to predict one period at a time, these arrays will then only have one column (and multiple rows). Unfortunately, the functions we will use later will need 1D arrays if we want to forecast only one period.

We can now easily call our new function **datasets(df)** as well as **import_data()**.

```
import numpy as np
import pandas as pd
df = import_data()
X_train, Y_train, X_test, Y_test = datasets(df)
```

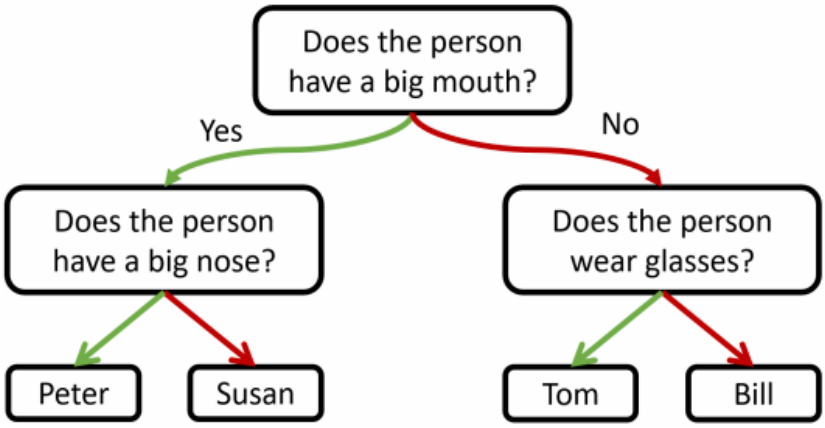
We now obtain the datasets we need to feed our machine learning algorithm (X_train & Y_train) and the datasets we need to test it (X_test & Y_test).

Note that we took y_test_len as 12 periods. That means we will test our algorithm over 12 different predictions (as we only predict one period at a time).
Forecasting multiple periods at once You can change y_len if you want to forecast multiple periods at once. You need to pay attention to keep *y_test_len ≥ y_len; otherwise*, you won't be able to test all the predictions of your algorithm.

Regression Tree

As a first machine learning algorithm, we will use a **decision tree**. Decision trees are a class of machine learning algorithms that will create a map (a tree actually) of questions to make a prediction. We call these trees **regression trees** if we want them to predict a number or **classification trees** if we want them to predict a category or a label.

In order to make a prediction, the tree will start at its foundation with a first yes/no question; and based on the answer, it will continue asking new yes/no questions until it gets to a final prediction. Somehow you could see these trees like a big game of *Guess Who?* (the famous '80s game): the model will ask multiple consecutive questions until it gets to a right answer.



In a decision tree, each question is called a **node**. For example, in the figure above, '*Does the person have a big nose?*' is a node. Each possible final answer is called a **leaf**. In the example above, each leaf contains only one single person. But that is not mandatory. You could imagine that multiple people have a big mouth and a big nose. In such case, the leaf would contain multiple values.

The different pieces of information that a tree has at its disposal to split a node are called the **features**. For example, the tree we had on the figure above could split a node on the three features *Mouth*, *Nose,* and *Glasses*.

How does it work?
To illustrate how our tree will grow, let's take back our quarterly dummy dataset.

X_train				Y_train
5	15	10	7	6
15	10	7	6	13
10	7	6	13	11
7	6	13	11	5
6	13	11	5	4
13	11	5	4	11
11	5	4	11	9
7	2	3	1	1

Based on this training dataset, a **smart** question to ask yourself to make a prediction is: Is the first demand observation > 7?

X_train				Y_train	Is the first demand observation >7?
5	15	10	7	6	No
15	10	7	6	**13**	**Yes**
10	7	6	13	**11**	**Yes**
7	6	13	11	5	No
6	13	11	5	4	No
13	11	5	4	**11**	**Yes**
11	5	4	11	**9**	**Yes**
7	2	3	1	1	No

This is a smart question as you know that the answer (Yes/No) will provide an interesting indication of the behavior of the demand for the next quarter. If the answer is *yes*, the demand we try to predict is likely to be rather high (>8), and if the answer is *no*, then the demand we try to predict is likely to be low (≤7).

Here is an example of a **bad** question.

X_train				Y_train	Is the third demand observation <6?
5	15	10	7	6	No
15	10	7	6	13	No
10	7	6	13	11	No
7	6	13	11	5	No
6	13	11	5	4	No
13	11	**5**	4	**11**	**Yes**
11	5	**4**	11	**9**	**Yes**
7	2	**3**	1	**1**	**Yes**

This does not help as this question does not separate our dataset into two different subsets (i.e., there is still a lot of variation inside each subset). If the answer to the question *Is the third demand observation <6?* is *yes*, we still have a range of demand going from 1 to 11, and if the answer is *no,* the range goes from 4 to 13. This question is simply not helpful in forecasting future demand.

Without going too much into details of the tree's mathematical inner workings, the algorithm to grow our tree will, at each node, choose a question (i.e., a **split**) about one of the available **features** (i.e., the previous quarters) that will minimize the prediction error across the two new data subsets.
The first algorithm proposed to create a decision tree was published in 1963 by Morgan and Sonquist in their paper "*Problems in the Analysis of*

Survey Data and a Proposal." There are many different algorithms on how to grow a decision tree (many were developed since the '60s). They all follow the objective of asking the most meaningful questions about the different features of a dataset in order to split it into different subsets until some criterion is reached.

Parameters
It's important to realize that without a criterion to stop the growth of our tree, it will grow until each data observation (otherwise called **sample**) has its own leaf. This is a terrible idea as even though you will have perfect accuracy on your training set, you will not be able to replicate these results on new data. We will limit the growth of our tree based on some criterion. Let's take a look at the most important ones (we are already using the **scikit-learn** naming convention).

Max depth Maximum amount of consecutive questions (nodes) the tree can ask.
Min samples split Minimum amount of samples that are required in a node to trigger a new split. If you set this to 6, a node with only 5 observations left won't be split further.
Min samples leaf Minimum amount of observations that need to be in a leaf. This is a very important parameter. The closer this is to 0, the higher the risk of overfitting, as your tree will actually grow until it asks enough questions to treat each observation separately.
Criterion This is the KPI that the algorithm will minimize (either MSE or MAE).

Of course, depending on your dataset, you might want to give different values to these parameters. We will discuss how to choose the best parameters in the following chapter.

Do It Yourself
We will use the **scikit-learn** Python library (www.scikit-learn.org) to grow our first tree. This is a well-known open-source library that is used all over the world by data scientists. It is built on top of NumPy so that it interacts easily with the rest of our code.

The first step is to call **scikit-learn** and create an instance of a regression tree. Once this is done, we have to train it based on our X_train and Y_train arrays.

```
from sklearn.tree import DecisionTreeRegressor

# — Instantiate a Decision Tree Regressor
tree = DecisionTreeRegressor(max_depth=5,min_samples_leaf=5)
```

```
tree = DecisionTreeRegressor(max_depth=5,min_samples_leaf=5)

# — Fit the tree to the training data
tree.fit(X_train,Y_train)
```

Note that we created a tree with a maximum depth of 5 (i.e., a maximum of five yes/no consecutive questions are asked to classify one point), where each tree leaf has at minimum 5 samples.

We now have a tree trained to our specific demand history. We can already measure its accuracy on the training dataset.

```
# Create a prediction based on our model
Y_train_pred = tree.predict(X_train)

# Compute the Mean Absolute Error of the model
import numpy as np
MAE_tree = np.mean(abs(Y_train —
Y_train_pred))/np.mean(Y_train)

# Print the results
print("Tree on train set MAE%:",round(MAE_tree*100,1))
```

You should obtain an MAE of 15.1%. Now let's measure the accuracy against the test set:

```
Y_test_pred = tree.predict(X_test)
MAE_test = np.mean(abs(Y_test —
Y_test_pred))/np.mean(Y_test)
print("Tree on test set MAE%:",round(MAE_test*100,1))
```

We now obtain around 21.1%. This means that our regression tree is overfitted to the historical demand: we lost 6 points of MAE in the test set compared to the historical dataset.

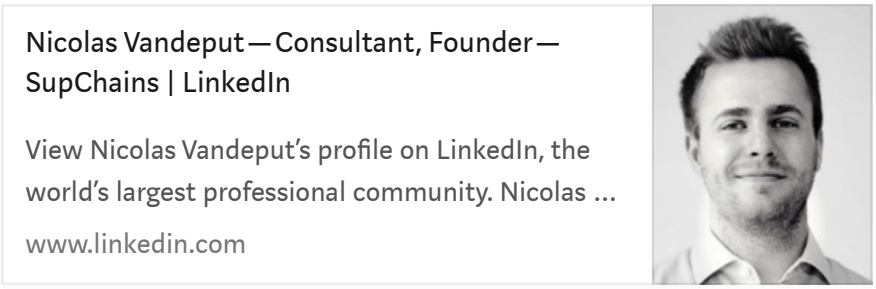
Going further

There are many ways to improve this result further:
- Optimize the tree parameters.
- Use more advanced models (like a Forest, ETR, Extreme Gradient Boosting).

- Optimize the input data.
- Use external data.

All of these are explained in the book **Data Science for Supply Chain Forecast** (available on Amazon)

About the author



Nicolas Vandeput is a supply chain data scientist specialized in demand forecasting and inventory optimization. He founded his consultancy company SupChains in 2016 and co-founded SKU Science —a fast, simple, and affordable demand forecasting platform—in 2018. Passionate about education, Nicolas is both an avid learner and enjoys teaching at universities: he has taught forecasting and inventory optimization to master students since 2014 in Brussels, Belgium. Since 2020, he is also teaching both subjects at CentraleSupelec, Paris, France. He published *Data Science for Supply Chain Forecasting* in 2018 (2nd edition in 2021) and *Inventory Optimization: Models and Simulations* in 2020.