MODELING MADE SIMPLE

Time Series: How to Beat SageMaker DeepAR with Random Forest

Improve your KPI by 15% with 3X faster, free & interpretable model

Dor Meir

Published in Towards Data Science
10 min read · Aug 17, 2022

Photo by Victoire Joncheray on Unsplash

If you're reading this article, you either consider using the Amazon SageMaker DeepAR algorithm for your **multiple time series** dataset, or you're already "stuck"

with one — and are not sure how to improve it any further.

How your dataset probably looks like. Airtable (data by the author)

Before delving into the neety grity technicals, you should contemplate on these:

DeepAR pros

1. ***No Data Science expertise needed*** *— If you have zero knowledge in Data Science, read no further. Go for the DeepAR, it requires almost no knowledge in handling data and creating predictive models.*
2. ***Stability*** *— The metrics' variance is low, as the results are rather stable over time. What you see (at first), is what you get (in the future).*
3. ***Prestige*** *— you'll be using a a state-of-the-art algorithm, probably used by Amazon themselves on their own sales data. Thus, it might be easier to sell to your non-technical co-workers.*

DeepAR cons

1. ***SageMaker expertise needed*** *— The SageMaker manual is 3447 pages long. Though DeepAR is only 16 pages out of it, you'll have to read them carefully and follow the instructions for: formatting your data (i.e. categoricals labelling), manual and automatic tuning of hyper-parameters, answering AWS instance requirements and feeding data to the model correctly.*

2. ***Rigidity & Black box*** *— It's not an easy task to improve an existing DeepAR model. There are a small, fixed list of hyperparameters, while changing their values is useless in case you already used the Hyper Parameters Optimization (which you should!). Also, there are no good enough explanations for the model suchlike features importances, so it's difficult to reiterate over the CRISP-DM phases or just explain the client what the heck is going on in his model.*

3. ***Cost*** *— The model's training runtime on AWS costs both time and money.*

This is how you'll feed your DeepAR model with data and hyper-parameters. Isn't it strange?

If you're still here with me, you either believe (like me) that the pros overwhelm the cons, or you're just curious to see what's ahead. Either way, my experience has taught me you CAN improve the DeepAR score by 15% using a 3X faster, relatively free of charge and a much more interpretable model, and you can do it even on a rather sparsed sales data. But that's not an easy task: all I can offer you is blood, sweat and many-a-hour of coding, all the way to the win.

Are you ready?

1. Know thy tools

Groupby, Groupby, and don't forget to Groupby

First and foremost, it's crucial to approach this problem with the right toolset in mind. Since the nature of the data is of an ordered series, **it's very tempting to pre-process the data using *for loops*** that iterate date after date and product by product (as demonstrated in machinelearningmastery). Nevertheless, I urge you — DON'T DO THAT! if you go down that rabbit hole, you might never come out as your pipeline will take hours to run.

Instead, get to know your new best friend: pandas Groupby operation, and all of it's wonderful little buddies: rolling, expanding, cumsum, cumcount, diff, sum, min, max, mean, pct_change, rank, head, tail, first, last, bfill, ffill, shift, nunique, etc… And, of course, **the Groupby queen**: ***apply.(lambda x:your_function(x))***, which allows for anything you want to do that is not already implemented in Groupby.

Get things sorted out

Don’t forget to ***sort_values()*** in the correct order before concatenating your Groupby result back to the DataFrame, and bare in mind that *bfill* and *ffill* functions are prone to spillover between the different groups. It’s safter to use groupby + merge, as it will ensure those missing values are filled with values calculated on the correct groups.

Assert yourself

It’s always a good idea to test the correctness of the inputs and outputs of the operations you make, even just by typing a simple ***assert*** command. By using *assert*, you can make sure the assumptions you made on the data, later used for your calculations, are true. Trust me, when the data turns complex it’s much harder to make heads or tails of what’s going on there without those useful little asserts.

Just your regular-normal-everyday 5-days rolling window min, max, mean and sum of product sales in the entire country, for each product. easy, right?

-> And the output. Airtable (data by the author)

2. Mind the gap

If your dataset is a complete one, i.e.:

#rows = #timestamps X #indexes

You can skip this point. Otherwise, you should probably fill in those missing observations, so you'd have a correct understanding of the true distribution of the target over the timestamps.

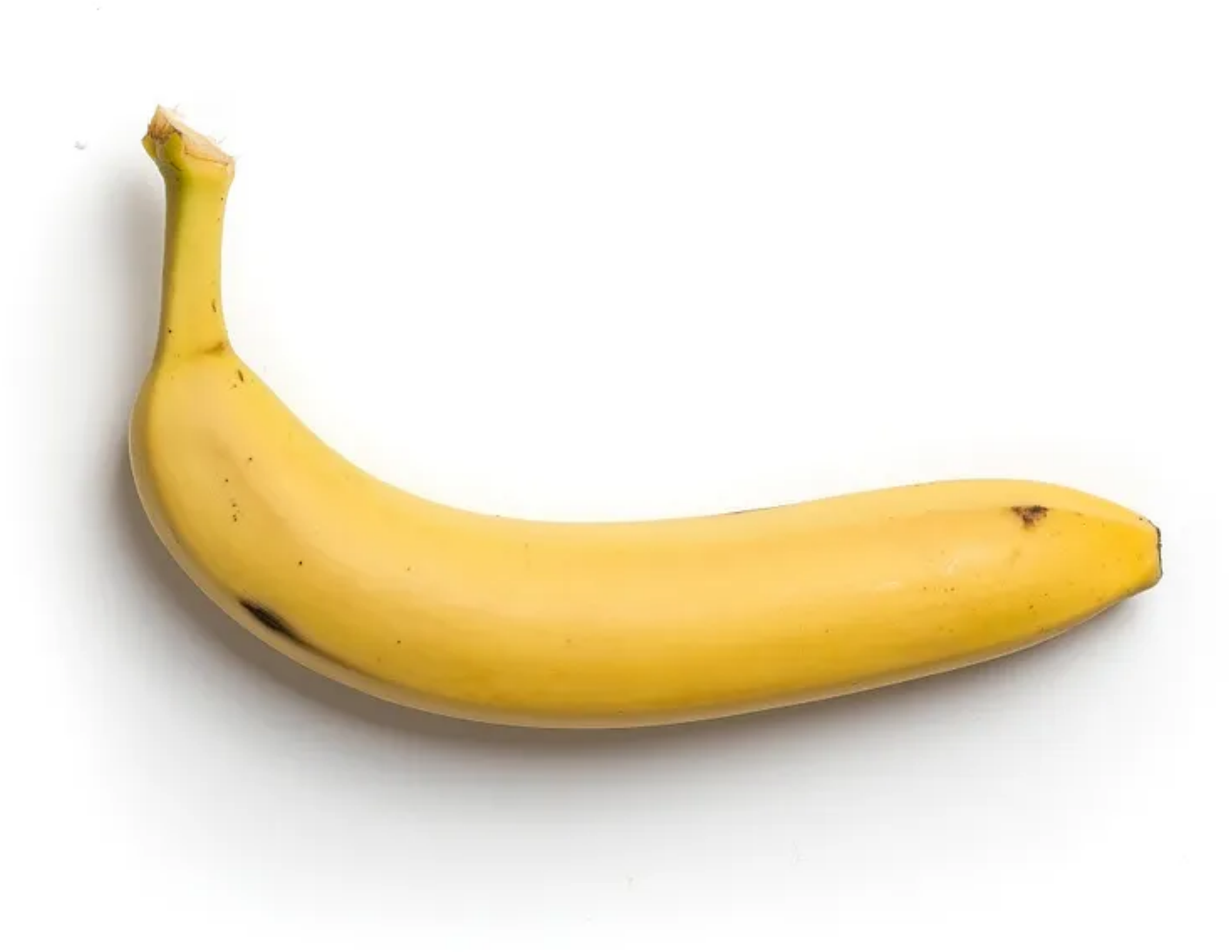
One popular reason for not having your dataset complete lies again in the nature of our multi-index time series dataset: **sales data tends to include information on actual sales, without any zero sales**. If precision is important and you would like your model to know when a sale will not occur, do feed it with zero rows.

Another thing to bear in mind, is that **sometimes timestamps might not come in the form of dates**. Consider a model predicting students' graduation. In this model, the time feature might be the student's academic credits gained so far, and not just some useless calendar dates.

Zeros are important! Airtable (data by the author)

3. What's your goal here anyway?

DeepAR will forecast your quantitative needs. But **what if timing is a little more important than the quantities?** For instance, your final goal is to have reasonable sales predictions for each product in each city in each day, because you must plan the room for the products in your stock. But it's of more importance for you to correctly predict the days where **any quantity** of the product will be sold, since any quantity above zero means you'll be sending a new truck with a load of bananas to the Chicago stock. You don't mind loading some extra bananas in this track but sending a new truck when no banana is going to be sold at all, might be a great waste of your money. If that's the case for you and timing is more important than quantity, you should consider transforming your target into a binary one: 1 for each quantity above 0, and 0 otherwise.



Sometimes it's just one single banana that makes all the difference. Photo by Mockup Graphics on Unsplash

But how will we have both binary predictions and quantitative ones in the same model? Easy — we'll train a two-step model: the first step predicts if a sale will occur, the second will **predict which quantity will be sold only for those times where a sale was predicted.** Don't worry for the extra labour we added here, since even though those are two different models with different targets, they can use the same exact features.

And… voilà! We now have a classification problem. Airtable (data by the author)

Lastly on the goal in matter, an important advice on how to solve any challenge: **spend some time scanning the literature, preferably the academic one.** You might get some excellent ideas on how to define but also solve your problem, from kind researchers who already spent enormous amount of time on this and decided to share their results. So, stand on those giant's shoulders.

4. Change your perspective

Time series algorithms such as ARIMA or RNN (=the under-the-hood of DeepAR), know how to handle your time series dataset as it comes, index by index. Supervised algorithms like Random Forest and XGBoost, on the other hand — don't. They treat each row separately from the others, and you're not supposed to feed them with the timestamp column. Thus, we'll have to convert our data structure into Cross Section (*Groupby.shift* will be handy for this). Though if you do go along this road, know that you must have only one row for each combination of date, city, and product, so Groupby beforehand.

feed it to the model without the date column, the order of rows doesn't matter. Airtable (data by the author)

Warnings

- Adding those lagged features will add cells with missing data in the first timestamps of each group — consider removing those rows altogether or imputing them with some meaningful value. The only thing you **can't** do is to fill them with the original sales data because that's just straight-out target leakage.
- Be careful not to shift rows in the wrong direction, and therefore creating future sales features instead of historical ones, which will again cause a leakage.

5. Enrich your story

Data completeness, project goal and problem perspective are acutely important, still without a good set of features to assist on the storyline, forecast might do poorly. And there are two types of features to our story: ***time dependent***, and ***time fixed*** (independent):

- ***Time fixed features*** tend to originate from categoricals (i.e. locality, name, brand) and can be dealt in the usual ways, ***target encoding*** is a one that reduces sparsity. Just remember to compute on train, save the computed statistics, and reapply them without computing again on tests.

- ***Time dependent features*** can be enriched both with the shifts mentioned in point 4, but also with ***rolling window*** aggregators such as mean, median, sum, min, max, like in point 1's example output. Time dependent features might also contain some useful data as an ***expanding* aggregators**, i.e. for each timestamp, the mean of sales until that timestamp. Lastly, there are the ***full train* statistics**, like having the sum (or weighted sum) of sales over the entire train period, in each timestamp. Those features contain data that's "always true" for some products, just like a *time fixed feature*. For instance, bananas weight in total sales over time is 50%, so each row with any bananas will get 0.5 in the *total_sales* feature.

Invest in the story, it'll return the investment. Photo by Nong V on Unsplash

A crucial issue here is the data pipeline. **You should have an efficient functional pipeline** — remember the pipeline will run twice for both train and test, with some differences between them (as some calculations will only occur in train and copied to test). Make sure each feature is completely independent of the other, so you can add and remove features without having to worry about their effect on other features.

6. Modeling, the right way

Finally, the bread and butter. Here are some time-series-specific modeling advices:

- After filling missing timestamp like in point 2, your dataset might become entirely imbalanced. **Upsample the sales rows**, so that they'll equal the number of rows with zero sales. This will increase the target's signal and can help your model learn better.
- **Don't split your data randomly**. Split it on time (train=past, test=last timestamp), and always remember before deployment to retrain your model on your entire dataset (train + test). That way, the deployed model will be trained on data as close as can be to the new data coming in.
- Remember to **drop any feature with future information**, or any timestamp that doesn't reappear(specific date is no good, specific month is acceptable).

Other general points which are relevant for essentially every ML model:

- **Drop the highly correlated features.** Their contribution will be small, while they create bias to the coefficients of the correlated features.
- **Optimize your decision threshold with respect to your client's KPI!** It might provide a great improvement and it's a rather easy move apply: calculate the KPI's value for a 100 different threshold values between 0 and 1, and choose the threshold that maximizes the KPI for the train data.
- Try different hyper-parameters for different algorithms, plot a train vs. test graph of the metric's improvement over the model's complexity. **Make sure you're close enough to "just fitting"**: on the one hand your model isn't overkilling it and learning the randomness of your train data, and on the other hand your model is training hard enough to achieve satisfactory results.

7. The extra edge

Multiple Models for Multiple Groups

Sometimes your data is so vast and complex, it's a tough job for the model to converge to an optimum solution. For instance, the bananas and apples patterns of sales are genuinely different, but apples sales patterns are also genuinely different over different

cities. It might not stop there, as some stores in the same city sale more apples on weekends and holidays— and, from some unknown reason — they might also sale more apples every third week of the month.

In that case, as well as in any other case, make an effort to **find meaningful groups in the data, and than train an independent model on each different group** (with all rules from point 5 applying for each model). You'd be amazed to see how this step might improve your model even further, even after you already squeezed the feature engineering and hyper-parameters optimization to the bone. Not to mention, DeepAR might not be AS easy to split into different models, due to a minimum of observations required per model or just the extra time and money needed for training more models, so you'll have an inherent advantage over it.

When there are many locked doors and a master is hard to find, use many keys. Photo by Alp Duran on Unsplash

How can you find those groups? You can use the categorical features (For example a model for each city, for each item, for each quarter), or — and that might be more relevant to time series data — you can **cluster together groups with similar sales patterns**. For example, create a feature of the sum of binary sales of a product in a city in the last 6 days before the last timestamp in train. This feature will naturally range from 0 to 6, and so you can train a separate model for each of those 7 groups. So, if Houston dates were sold 3 times in the last 6 days before the last date, and Los Angeles elderberries were sold 5 times — Houston dates and LA elderberries will be fed into different models. Lastly, **you can always find meaningful groups using statistics**: have in mind a list of possible splits of different groups, target-encode those groups, and see if the means of the splits are different "enough". You can also use Chi-squared test for this task.

Summary

*

Amazon SageMaker DeepAR model for multiple time series data is a state-of-the-art algorithm, developed by a tech giant. It doesn't require DS knowledge and it's quite stable and reliable. Nonetheless, it does not always deliver the best results, and even if the results are reasonable — they will hardly be interpretable, and always wear significant time and money costs.

The skilful Data Scientist, I argue, can do better. If she: uses the Groupby command, makes sure the data is complete and full, defines the target correctly, manipulates the time series into cross section, engineers insightful features, traines the model propely and with best achievable accuracy, and perhaps even trains different models for different groups —
She can and will defeat DeepAR's best KPI.

. . .

Feel free to share your feedback and contact me on LinkedIn.

Thank you for reading, and good luck! 🍀

Deepar Sagemaker Random Forest Timeseries Pandas