A Pocket Watch. It looks cool doesn't it. Very hipster-ish.

Multivariate Time Series Forecasting Using Random Forest

Hafidz Zulkifli Follow
Mar 31 · 12 min read ★

Introduction

In my earlier post (Understanding Entity Embeddings and It's Application) [1], I've talked about solving a forecasting problem using entity embeddings — basically using tabular data that have been represented as vectors and using them as input to a neural network based model to solve a forecasting problem.

This time around though, I'll be doing the same via a different technique called Random Forest.

Since I don't intend for this post to be a tutorial on Random Forest, any interested readers keen in diving into the matter can learn more about it in the excellent machine learning book currently being compiled in here [2]. If you somehow fancy yourself as

total beginner in machine learning and want to learn about Decision Trees and how it relates to Random Forest, check out this tutorial on Titanic dataset. [3]

What I will be talking about though is how I built a multivariate forecasting model using Random Forest, and some of the considerations that went with it.

To put things into context, my new task required me to build a model that is able to able to forecast our web traffic months into the future, using multiple variables that may or may not have any effects on the prediction. While I can't divulge on the variables being used (for obvious reasons), it is in essence a web traffic time series data containing the usual features like traffic, user behaviour, demographic and some external features.

Why Random Forest?

I chose to use Random Forest (RF) mainly because the task that I had to deal with required me being able to explain the model and provide predictions. As the typical neural network is pretty much a black box (I'm not sure whether attention mechanisms can give out feature importance .. maybe I haven't read enough papers …), I decided to skip using them this time around.

From the various statistical machine learning technique that is available, I went with RF since it felt more robust and easier to train as compared to others.

However caution needs to be exercised when relying on scikit-learn Random Forest feature importance. In [8], the authors made a claim that due to how it is being calculated, bias can occur in the result. To give us a glimpse of this, consider their example when building a classifier using data from Kaggle's Two Sigma Connect: Rental Listing Inquiries

Image from [8]. Figure 1(a) depicts feature importance from a model that predicts rent prices, while Figure1(b) is a model that predicts interest level.

The above diagram depicts feature importance chart based of 2 different model. The left chart came from a model that tries to predict the price of the rent, and the right is from a model that predicts the interest level. The `random` variable meanwhile is generated using random number generator, to depict randomness and point out any unimportant features (the intuition being any features that is ranked lower than `random` should be considered junk).

As we can see in Figure1(a), `random` is ranked lowest of the bunch — which made sense. While the number of bathroom being the most important feature does seem odd, we'll give it a pass for now (maybe they ARE really concerned about being hygienic and all). For those of us who have actually gone and bought or rented a property though, the common intuition is that usually the location is the most important aspect in determining the price of the place.

In Figure2(a), we can see that `random` is somehow ranked higher than `bedrooms` and `bathrooms` . So somehow our randomly generated numbers is deemed to have predictive power. Weird … (and this concludes my brief foray into demonstrating how bias can occur in RF feature importance).

For more details on the above example, and proposed solution, head to [8]. They also had some good points on why linear models might not be the best method at providing feature importance via regression coefficients.

> *We recommend using permutation importance for all models, including linear models, because we can largely avoid any issues with model parameter interpretation. Interpreting regression coefficients requires great care and expertise; landmines include not normalizing input data, properly interpreting coefficients when using Lasso or Ridge regularization, and avoiding highly-correlated variables (such as country and country_name). To learn more about the difficulties of interpreting regression coefficients, see Statistical Modeling: The Two Cultures (2001) by Leo Breiman (co-creator of Random Forests).*

Hence another +ve for using Random Forest.

Why Not ARIMA?

I did contemplate using ARIMA and had in fact conducted a brief study on the subject to consider whether it suits the task or otherwise. However in the end I came to the understanding that it's mainly suited for building univariate (single variable) based model.

Should the problem be univariate though, I'd probably use Facebook's Prophet [4] package to help me. Trying it out with our traffic data wasn't that hard at all, and with few tweaks on adding yearly, quarterly and monthly seasonality patterns — it was already able to give a really decent result.

The `forecast` package by Hyndman in R was also an option. But I decided against it since I'm trying not to use R in this instance.

Other Methods

In exploring the possible options to solve this problem, I've came across a few methods that warrants a mention:

1. Vector Autoregressive Model (VAR). It's basically a multivariate linear time-series models, designed to capture the dynamics between multiple time-series. For more details, check out [5] and [6]
2. AWS Deep AR. I only came upon this in the latter stage of my work. It's basically a supervised learning model meant to be used for univariate data. However, it also allows for multivariate time series input. More details in [7].

Due to strict deadlines requirement, I was unable to explore the above algorithms — hence I had to pass on them.

Building a Forecasting Model

In this section I'll be going through some of the activities that I felt was important (and particularly unique to) when building a forecasting model using tree based methods.

Random Forest can't predict for more than t+1 right?

This is true — one can't use Random Forest to forecast into more than 1 value or indefinitely into the future as one can with linear model. What this means is that when I'm trying to predict 6 months into the future, I'll need to actually use 6 months of past data as the input.

As an example, if my model is able to predict 6 months into the future; to predict the traffic at 1 July 2019 I will need to use the data points on for 1 Jan 2019 (for simplification, I assume all months have 30 days in a month). To predict for 2 July 2019, I'll use 2 Jan 2019. And so on, until 30th June 2019, which gives me prediction for 30th Dec 2019.

The downside of using this method is inability to use data that is closer to the present (ie using 30th June 2019 data for 1 July 2019 prediction as in the above example). In our scenario, as what we're really interested in is really the amount of traffic that is occurring 6 months down the road, this is still acceptable.

Why 6 months? Because knowing how traffic will pan out (for better or worse) would allow us time to make the necessary changes to try and change it. A shorter time frame makes for rushed decision making and execution.

If however, your use case requires that a really accurate prediction on what happens tomorrow or next week (for example stock trading), it would be better to use data from a more recent time frame.

Adding time based variables

We introduce additional date based variables using the `add_datepart` method from fastai library [9]. Doing this enables us to capture trends, seasonal or cyclical pattern from our dataset. It's quite a useful method to have in one's arsenal, especially if we have to do a lot of work in time series data.

```
def add_datepart(df, fldname, drop=True):
    fld = df[fldname]
    if not np.issubdtype(fld.dtype, np.datetime64):
        df[fldname] = fld = pd.to_datetime(fld,
                                           infer_datetime_format=True)
    targ_pre = re.sub('[Dd]ate$', '', fldname)
    for n in ('Year', 'Month', 'Week', 'Day', 'Dayofweek',
              'Dayofyear', 'Is_month_end', 'Is_month_start',
              'Is_quarter_end', 'Is_quarter_start', 'Is_year_end',
              'Is_year_start'):
        df[targ_pre+n] = getattr(fld.dt,n.lower())

df[targ_pre+'Elapsed'] = fld.astype(np.int64) // 10**9
    if drop: df.drop(fldname, axis=1, inplace=True)
```

Running the method gives us the following variables.

- Month
- Year
- Week

- Day
- Dayofweek
- Dayofyear
- Is_month_end
- Is_month_start
- Is_quarter_end
- Is_quarter_start
- Is_year_end
- Is_year_start
- Elapsed

However we don't necessarily need to use all of them. For example, in my use case my data points are based weekly — hence I don't really have a need for `Day` variable.

Scale/Normalize your data

As trends can sway up and down pretty drastically in a time series chart, to ensure that our algorithm can generalize better on the test set it is preferable to first normalize our data into some smaller scale. Box Cox, Log or the StandardScaler (and other methods of course.. my list is by no means exhaustive) can be used to revise our data to a more appropriate.

Just remember to scale them back once you've done your prediction.

Removing trends in input variables

Ensuring data stationarity is important as this will enable us to provide accurate prediction into the future. To highlight why this is a problem, consider the following diagram.

The y values in the validation set doesn't appear in the training set. [10]

The above demonstrates how one would normally go about splitting the data into training and validation set for a time series problem. Note that due to the split, the training set will fail to capture the y-values of the validation set.

> *Random Forests don't fit very well for increasing or decreasing trends which are usually encountered when dealing with time-series analysis, such as seasonality [10]*

To remedy this, we will need to basically "flatten" the trends so that it becomes "stationary". The simplest way to do this is by using the slope of the data or simply deducting the variable at time *t* it's lagging value at time *t-1*.

Add lagging variables

As Random Forest evaluates data points without bringing forward information from the past to the present (unlike linear models or recurrent neural network), defining lagging variables help bring about patterns from the past to be evaluated at the present. The choice of how far back should we decide to go pretty much depends on how cyclical the data presents itself. Ideally, we want to be able to capture at least one cycle of the data.

Using delta as target

Similar to point made earlier, we do this (add lag variables) even to our target variable, as it is time-based. However since we can't directly get data at t-1, we use current traffic instead. As such, the formula for the predicted traffic is as below.

where traffic(f) is traffic 6 month into the future and traffic(n) is current traffic.

It sometimes help to jot down the formula somewhere as we'll need to apply it before and after our model's prediction.

Establishing benchmark

It goes without saying that having a benchmark to compare against is important so that we can have a point of reference to compare our model with. In my case, I went with 2 types of benchmarks; (i) using current traffic, and (ii) using 6 month average traffic.

In terms of measurement metric for the model, I chose RMSPE (root mean squared percentage error) so that I am able to evaluate the error from a value of 0 to 100 (makes it easier to communicate to stakeholders too).

Curse of dimensionality

This point is not that unique to time series actually. But after creating a few hundred additional variables from the lagging data, our model can easily overfit due to the enormous amount of variables that we introduce while at the same time still relying on only limited amount of data points.

To solve this, I made a conscious decision to limit the amount of variables being used to only the top N, where N is lesser than the total amount of data points.

Backtesting strategy

Initially I was only using the usual train-valid-test data split to check the validity of my model. However as I dive deeper into the subject and nearing the completion, it finally dawned on me that I can never be really sure of the model's performance if I don't subject it to a rigorous time based testing.

Uber [11] as it turns out has a lot of good reference when it comes to the topic of time series.

Two major approaches to test forecasting models. Sliding window (left) and expanding window (right). Image taken from [11]

In my scenario, I've adopted the expanding window method — where my training data grew bigger at every iteration and keeping the testing/validation window interval the same. The testing window are kept the same so that we can compare both it's performance and stability over time.

Ideally, as also mentioned in [11], we should swap the test methodology from expanding window to sliding once our data size is large enough.

A more in depth write-up on the topic can also be found in [12].

Retraining model and predicting

Aside from making the data stationary before training the model, this is one of the biggest take away lesson that I got from doing this exercise.

> *You can't just deploy a static model and score it, the concept of model serving doesn't make sense for time series forecasting. Instead, you need to insure that training and model selection can be done on the fly in production, and you have to insure that your entire training set can be stored and processed in production. [13]*

My usual understanding of how one would deploy ML model has always been that of a model being served for around 3–6 months before needing to be retrained (if any), and perhaps monitored for any signs of performance degradation over those period.

Suffice to say, retraining and predicting on-the-fly as argued in [13] did caught me by surprise at first. However, since the reliability of time series model is pretty much dependent on how recent the data is (ie, imagine we're trying to predict traffic at time $t+1$; the best predictor to use would then be traffic at time t), to retrain the model at every prediction time then isn't that absurd of a thought after all.

Of course, if the structure of the historical data is stable enough; perhaps the retraining can be scheduled at a less frequent interval, or perhaps only when a change-point [15] event has been detected.

Conclusion

The above write-up is mostly based off of my learning notes taken throughout my time working on this use case. When I first started working on it, it never dawned on me that I'd be learning so much — kinda expected it from time series analysis, but not on model building.

What I am quite surprised though is how not-so-straight-forward it is to find information regarding the subject. Arguably though, perhaps not many data scientists out there prefers to build their forecasting model using tree based methods.

My final model, in the end, was not of random forest. I swapped it with a model built using LightGBM for better accuracy and the added benefit of being able to generate prediction intervals to go along with the forecast (by means of quantile regression). Careful consideration has to be taken when using gradient boosting methods though since it's rather easy to build a model that overfits the data. In my case, the documentation at LightGBM [14] gave me some idea on how to overcome this.

Finally, a special thanks to work mates who've been a great bouncing board and helped me gain clarity on the subject over such short interval.

Update (3/4/2019): Added additional section *Random Forest can't predict for more than t+1 right?* and *Scale/Normalize your data* based on reader's feedback. (Thanks Husein.)

. . .

References

1. Understanding Entity Embeddings and It's Application
2. Random Forest Regressors, by Terence Parr and Jeremy Howard
3. Titanic: Getting Started With R, by Trevor Stephens. *(To be honest I just included a plug for this tutorial since it was THE FIRST article that I learned from back in 2015 when I started off in data science. Currently you can find way more articles on decision trees/RF on Medium etc, but way back then—this article was one of the only few out there. Yes, I first started off in R. And yes—I am being sentimental about it. Lol.)*
4. Prophet, Facebook

5. Introduction to Vector AutoRegression (VAR) , IMF
6. Time Series Forecasting Using Recurrent Neural Network and Vector Autoregressive Model: When and How, by Jeffrey Lau
7. DeepAR Forecasting Algorithm, AWS
8. Beware Default Random Forest Importances, by by Terence Parr and Jeremy Howard
9. The method is no longer available as part of the latest library. You can still find in the Rossman notebook though.
10. Why Random Forest can't predict trends and how to overcome this problem?, by Aman Arora
11. Forecasting at Uber: An Introduction, by Uber
12. Time Series Nested Cross-Validation, by Courtney Cochrane
13. 3 facts about time series forecasting that surprise experienced machine learning practitioners, by Skander Hannachi, Ph.D
14. Parameters Tuning, LightGBM
15. Change detection, by Wikipedia

Machine Learning Data Science Artificial Intelligence Time Series Forecasting