False Prophet: Feature Engineering for a Homemade Time Series Regression

Building on ideas from Meta's Prophet package to create powerful features for time series machine learning models

Bradley Stephen Shaw

.

Published in

Towards Data Science

.

15 min read

.

Oct 14

Scott Rodgerson on Unsplash

Meta's Prophet package[1] is one of the most widely-used packages for time series. At least anecdotally, according to me, after looking through a list of time series articles that I've bookmarked for later reading.

Sarcasm aside, I have used the package before and I love it.

Another great resource for time series modelling is Vincent Warmerdam's talk titled "Winning with Simple, even Linear, Models"[2] where he touches on modelling time series with linear models (with a bit of preparation).

Now, there are some elements of data science which blur the boundaries of art and science — think hyperparameter tuning, or defining the structure of a neural network.

We're going to lean into the art and do what a lot of the great artists have done: borrow ideas from others. So, in this series of articles we'll be borrowing feature engineering ideas from Prophet, and linear modelling ideas from Vincent to perform our very own time series regression with a real-world time series.

The big picture

Let's touch first on what the overall goal is, before we hone in on feature engineering.

The overarching goal is simple — to generate the most accurate forecast of future events across a specified time horizon.

We'll start from scratch with a time series containing only a date variable and the quantity of interest. From this, we're going to derive additional bits of information which will allow us to model future outcomes accurately. These extra features will be heavily "inspired" by Prophet.

We'll then feed our engineered data into a lightweight model, and let it learn how to best forecast into the future. Later on, we'll dive into the model's internal workings — after all, we'll need to understand what's driving our forecasts.

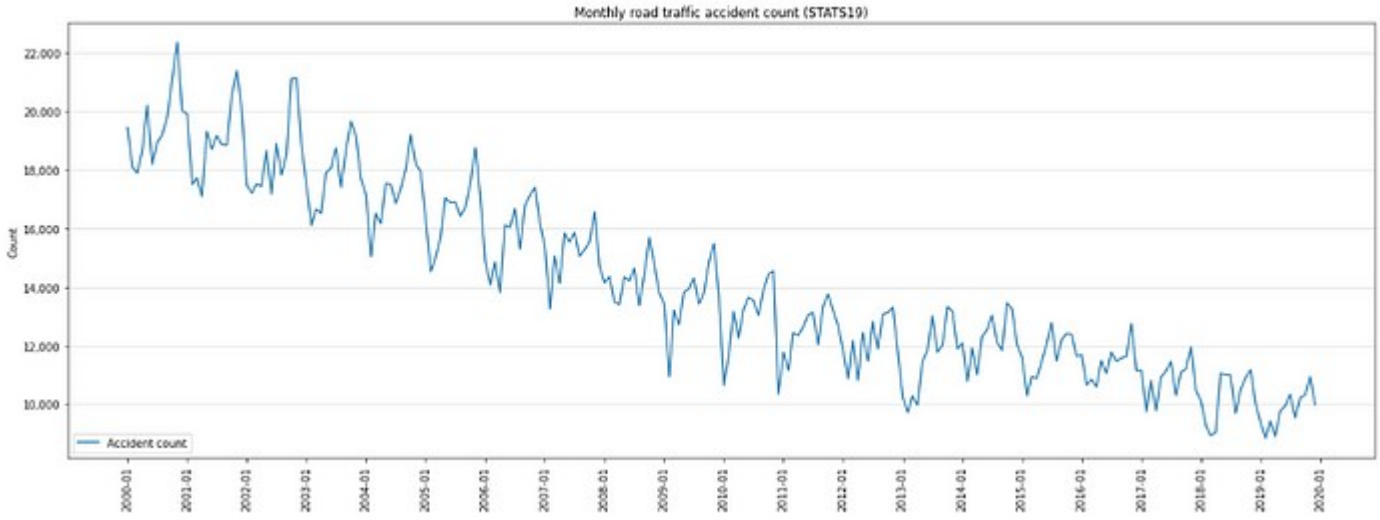
Now that we've seen the forest let's get a close up of the trees, starting with a look at our data.

Data

We're going to be using real-world data from the UK — in this case, road traffic accidents.

This is the STATS19[3] data set produced by the UK government. This data set is quite large so to make things a bit more manageable, we're going to aggregate the daily accident count up to a monthly figure.

Visualising our time series, we see a downward trend and a strong yearly pattern. It could also be argued that these patterns change at some point between 2012 and 2014.



That's already two kinds of features that we'll need to create — something to capture the overall direction of travel, and something that captures the repeated annual pattern (or seasonality).

Feature engineering

We'll touch on the general ideas that will drive our engineering before moving on to its implementation.

The beauty of Prophet

Prophet uses a decomposable time series model with three main components combined additively (with a little randomness on the side). Mathematically, this is:

$$y(t) = g(t) + s(t) + h(t) + \epsilon_t$$

> Here *g(t)* is the trend function which models non-periodic changes in the value of the time series, *s(t)* represents periodic changes (e.g., weekly and yearly seasonality), and *h(t)* represents the effects of holidays which occur on potentially irregular schedules over one or more days.[4]

It's this decomposable model form that makes Prophet so flexible, and it's this idea that a time series is separable that will guide our feature engineering: that is, we'll generate features that will help us model each one of these components.

Our imitation won't be a Prophet doppelganger — we're only taking inspiration from it. So we'll make a couple of changes:

- *g(t)* will also represent step changes or change points in the time series.
- We won't focus too much on the error term (epsilon), other than to remember that Prophet uses it to represent "idiosyncratic changes which are not accommodated by the model"[4].

Aside: if you're unfamiliar with the components of a time series, this article is a good summary:

Let’s Do: Time Series Decomposition

A guide to effectively breaking a time series into its constituent parts

pub.towardsai.net

We’ll start with basic features that we can get from the date field, before deriving some more imaginative features.

Base features: step zero

As a warm up, let’s get out some basic date-related features:

```
# set date as the Frame index
df.set_index('date', inplace=True)
# simple date features
df['yr'] = df.index.year
df['qtr'] = df.index.quarter
df['mth'] = df.index.month
df['dim'] = df.index.days_in_month
```

All of these features are available directly. It’s probably clear — even to the uninitiated — that these features are likely going to be predictive of the monthly accident count.

Time for some themed engineering.

Trend

Trend, or long-term changes over time, can take various forms.

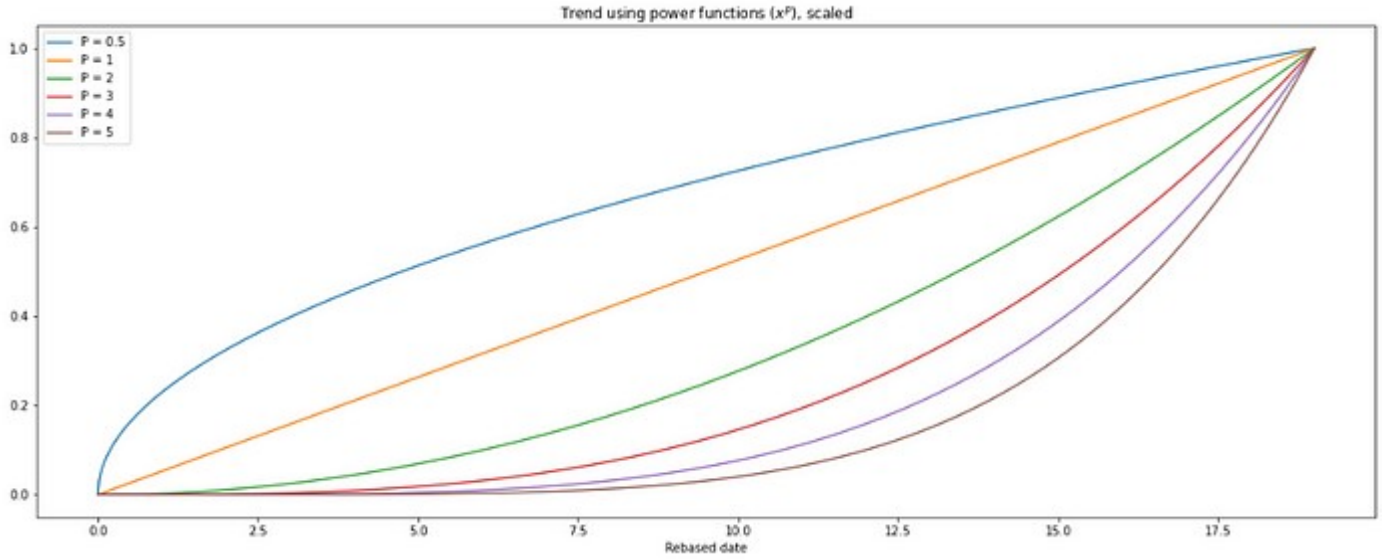
If present at all, trends can often be quite simple — consistent upward or downward changes are not uncommon. The airline passenger count data that’s used in many time series demonstrations[5] has a very clear and simple trend at play.

However, trends can get more complicated than that. They could for instance, be non-linear, where there is an acceleration or deceleration in the rate of change. There could be multiple instances of acceleration or deceleration. Or there could be step changes where there is a sudden change in location of the trend.

We’ve seen above that there looks to be a downward linear trend in our data, with a change point somewhere between 2012 and 2014. I’m not entirely sure of the exact form of the trend, so I’ll create a variety of them and let the model figure out which is best:

```
# fraction of year
df['yr_fraction'] = df.index.year + (df.index.month - 1) / 12
# add non-linearity
yr_fraction_rebased = df['yr_fraction'] - df['yr'].min()
df['yr_fraction_sq'] = yr_fraction_rebased ** 2
df['yr_fraction_cube'] = yr_fraction_rebased ** 3
df['yr_fraction_quad'] = yr_fraction_rebased ** 4
df['yr_fraction_quint'] = yr_fraction_rebased ** 5
df['yr_fraction_sqrt'] = yr_fraction_rebased ** 0.5
```

Visually, this gives us a number of possible trends (with some scaling to get everything to fit on the same chart):



Aside: it's important to note that while all of these trends look to be increasing, the model will be able to use these to capture the decreasing trend in the data, by for example using negative weights or coefficients. This applies not only to the trend components, but all features used in the model.

Now for some change points.

Prophet detects change points by first specifying a large number of potential change points, and then using as few of them as possible[6]. Prophet's default approach is to create 25 evenly spaced change points over the first 80% of the data.

We'll do something similar by first creating many potential change points and then letting the model choose which points to use. This is not too dissimilar from Prophet, but there's no constraints on spacing.

```
# changepoints
changepoints = pd.DataFrame()
for date in df.index.unique():
    date = pd.to_datetime(date)
    date_str = f'change_{date.strftime("%Y_%m")}'

    # allow only X-erly changes
    if date.month % 3 == 0:
        temp = pd.DataFrame(
            {date_str:np.where(df.index <= date,0,1)}
        )
        changepoints = pd.concat([changepoints,temp], axis=1)
```

If we take a look at the first 12 rows, we see how the change point creation has worked:

	change_2000_03	change_2000_06	change_2000_09	change_2000_12	change_2001_03	change_2001_06	change_2001_09	change_2001_12	change_2002_03
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0	0
5	1	0	0	0	0	0	0	0	0
6	1	1	0	0	0	0	0	0	0
7	1	1	0	0	0	0	0	0	0
8	1	1	0	0	0	0	0	0	0
9	1	1	1	0	0	0	0	0	0
10	1	1	1	0	0	0	0	0	0
11	1	1	1	0	0	0	0	0	0

Not very complex at all as we have a column for each change point feature indicating whether or not the observation happened before or after a given date.

It's worth mentioning that I've only allowed change points to happen at the end of each quarter. Unless we're absolutely certain of changes in the series, the setting of these points can be a bit of an art where we balance flexibility against overreaction; change points need to be frequent enough to capture real changes in trend but not so frequent that they begin to capture noise.

In this case quarterly change points have a bit going for them. Firstly, they effectively put a minimum time threshold on how long a change has to last before it is considered "real" — potentially useful in reducing the model's propensity to confuse signal for noise.

In the UK, quarterly changes roughly align with seasonal changes and significant calendar changes (e.g. 1 January).

There are external environment effects to consider too: new registration plates are released around March and September of each year, which usually drive a spike in new car sales. As new cars are generally safer than old cars, it's not unreasonable to imagine that the change in car parc mix would have an impact on the number of road traffic accidents.

While it's probably not too bad a place to start from, we might have to circle back later for some fine-tuning.

Seasonality

We refer to regular or periodic effects present in a time series as seasonality.

Prophet uses Fourier series to represent seasonal effects in the additive model. This generalises as the following:

The seasonality $s(t)$ can be expressed as:

$$s(t) = \sum_{n=1}^{N} \left(a_n \cdot \cos\left(\frac{2\pi nt}{P}\right) + b_n \cdot \sin\left(\frac{2\pi nt}{P}\right) \right)$$

where P is the regular period that the seasonality is expected to have.

The Fourier representation essentially implies that all the repetitive effects that we see in a time series can be represented by a series of sine and cosine waves of varying period.

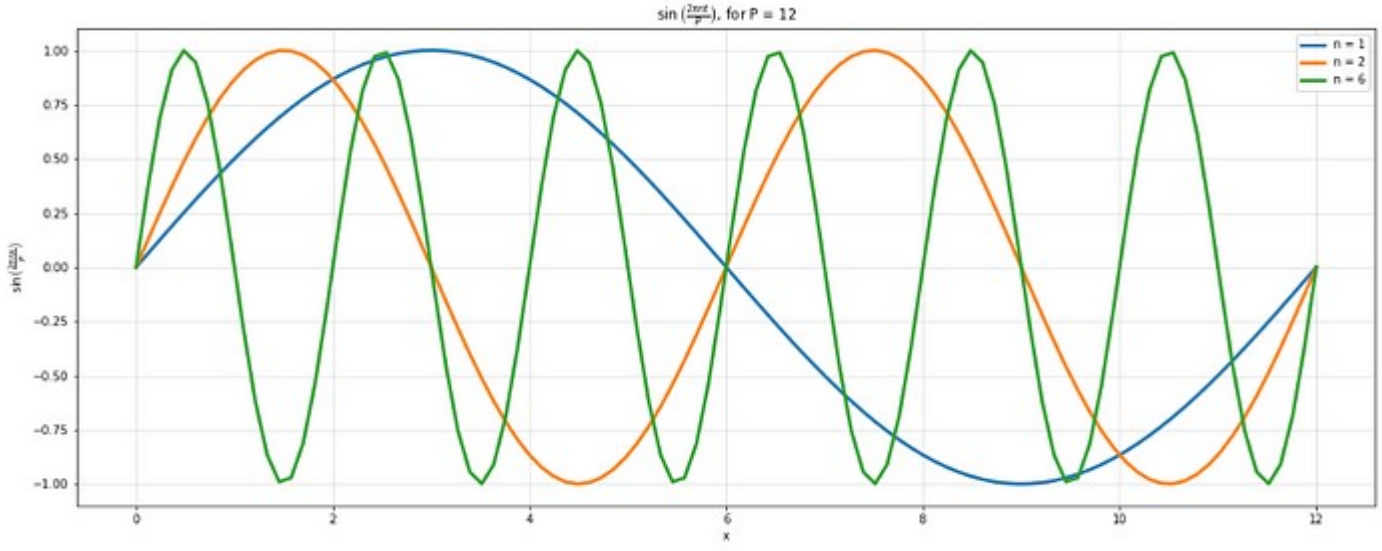
As we're working with monthly data, we'd expect to see seasonal effects around the same time every year; in other words, the *period* of our time series is one year, or 12 months. So we'd need to set $P = 12$.

$N = 10$ and $N = 3$ have been noted to work well for series with yearly and weekly seasonality respectively[4], but we'll extend N to 12 for good measure.

Remember that we're not creating a separate seasonal model, but rather seasonal features which our single model can assemble to represent periodic variations. With that in mind we create our Prophet-inspired features as follows:

```
# Prophet features
for j in range(1,13):
    df[f'prophet_sin_{j}'] = np.sin(2 * np.pi * df['mth'] * j / 12)
    df[f'prophet_cos_{j}'] = np.cos(2 * np.pi * df['mth'] * j / 12)
```

This creates multiple sine and cosine waves of various periods, ready for the model to assemble together in order to capture seasonality. So as input features, these may look a little like so:



Aside: note how higher values of n reduce the period — or time needed to complete a full cycle — of the sine function.

Our model will select and scale these functions in various ways so that the periodic (or *seasonal*) element of the time series is appropriately accounted for… that is, the model will determine the *a* and *b* coefficients in the formula above.

Holidays

> Holidays and events provide large, somewhat predictable shocks to many business time series and often do not follow a periodic pattern, so their effects are not well modeled by a smooth cycle.[4]

An excellent example of this is the Easter weekend which in the UK has a very definite impact on vehicular incidents. However, this holiday weekend doesn't occur during the same month each year — in some instances it falls in March, while in other years we have to wait until April before we can start our search for chocolate rabbits.

While we will know fairly well in advance when Easter occurs (and will of course know when it occurred in the past), it's quite difficult to model with the seasonal approach above. So we'll take a different tack and instead count the number of bank holidays and business days in a month, which should allow us to also capture the effect of more regular holidays like Christmas.

We can use `numpy` to get to business days:

```
# business days
begin = df.index.values.astype('datetime64[D]')
end = (df.index + pd.DateOffset(months = 1)).values.astype('datetime64[D]')
df['bus_days'] = np.busday_count(
    begindates = begin,
    enddates = end
)
# holidays
df['hols'] = pd.Series(df.index).apply(count_holidays).values
```

… but need the `holidays` package and a bit of help from StackOverflow[9] to get to monthly holiday count:

```
import holidays
def count_holidays(u):
    hols = holidays.country_holidays('GB')
    days = pd.date_range(u, u + pd.DateOffset(months = 1))
    return sum(y in hols for y in days)
```

We'll leave out the number of weekend days: as it can be derived from existing features, using the count of weekend days could introduce unwanted feature correlation.

And that it's — feature engineering complete, and we're just about ready to move on to the modelling.

Wrap up and ramble

We've covered a lot of ground in this article. As is becoming tradition, we'll quickly recap and then have a bit of a ramble.

Summary

After touching on our aspirations to build an awesome forecasting model we looked at UK road traffic accident data. We saw strong trend and seasonality in our aggregated monthly counts and knew that we'd need to create a variety of features to capture these effects.

We started our feature engineering process with a light warm up — extracting simple and directly available date features.

We moved on to building features to capture the trend along with some allowance for change points, which were treated fairly simply. There was some potential merit to our logic but we acknowledged that it might need some fine tuning.

We used adapted Fourier transforms to model seasonality, creating 12 sets of seasonal features.

Finally we moved on to creating holiday features, choosing to focus on the number of working and holiday days in a month.

Cyclical feature encoding

When building new features we need to keep with two things in mind — what *could* be predictive of our target and how it would be interpreted by the machine.

A good example of this is month of the year which we usually represent using an integer mapping (i.e. January = 1, …, December = 12). We can be fairly sure that the month of the year would be a strong driver of accident count. But if we were to pass to the model the integer encoding, the model would treat the December of one year as something very different to the January of the following year, even they are temporally adjacent!

We solve this issue with cyclical feature encoding, or more specifically by conversion to polar co-ordinates. Since neither the sine nor cosine transformation deliver a unique encoding on their own, we use the combination of both.

The code above doesn't show any examples of cyclical encoding but it is used in my workflow and turns out to be an important feature in the model (see part 2).

The Prophet features

Following in a similar vein, our "Prophet features" relied heavily on sine and cosine transformations. In reality, these are really Fourier transforms.

Eagle-eyed readers may have picked up on how the Prophet features have been created. In the original paper, the time dimension has been re-based to a certain point and every observation is reflected as being t time units after that. We've not done that, instead opting to go another path. If I ever revisit this, it may be something to consider.

Lagged features: the elephant in the room

So far, I've really just glossed over the use of lagged features. Or lack of use to be precise.

Using previous values of the target quantity to predict the current or future value of the target quantity — i.e. using values that are "lagged" in time — is a staple in a lot of really great time series models. And for good reason, as they are generally strong predictors.

My reluctance to do so centres on the whole purpose of the model — to be good at forecasting. When we forecast with lagged features, we usually have to "walk" the lagged features forward and transition from using actual values of the target to using predicted values of the target.

To make that more concrete, consider a model which uses one lagged feature — that is, we use the value of the target at time $t - 1$ to predict the value of the target at time t. We're interested in using the model to forecast 3 steps into the future.

The first forecast (at time $t + 1$) will use the value of the target today. Since the target value is known there are no issues here and it's business as usual.

Now consider the forecast for time $t + 2$. We need the target value as at time $t + 1$ in order to use our model. Of course, the *true* target value is unknown at this point and so we resort to using the *predicted* target for time $t + 1$. When it comes to forecasting time $t + 3$, we walk forward the prediction from time $t + 2$, so on and so forth. From this it's clear how prediction error can get

baked into the forecast; early errors get compounded as poor predictions get walked forward and reused. I'm not a fan of this.

There is a secondary benefit from not using lagged features, and that is model explainability: we are forced into modelling the target in a different way and really have to think about (and model!) the drivers of outcome.

Oftentimes this leads to a better conversation with stakeholders, as explaining a forecast starts to sound like "long-term trend represents X% and seasonality represents Y% of a forecast of Z" rather than "the forecast is B because the value of the previous forecast was A".

A last note on lagged features before moving on. We aren't constrained to using lagged *target* features, so while we've discussed including previous values of the target feature we could equally include lagged predictors with similar caveats and requirements.

This isn't intended to be a blanket put down of the use of lagged features — I'm sure there are use cases where it makes perfect sense to do so. The number of lags used and length of forecasting window may even mean that this is a non-issue.

Change points

Let's talk about change points, and the creation thereof.

I've created change points in a really simplistic way, and I'm sure there are many ways to improve my implementation. Prophet arguably does it better by creating evenly spaced change points in the first 80% of the data, but then there are a few things to consider.

This reduces the impact of more recent spurious change points on future forecasts — that's a good thing.

But how many true changes happen on evenly-spaced time intervals? And if changes really did happen at that cadence, wouldn't it be better thought of as some seasonal impact? Yes, it's splitting hairs. Yes, it is important. Fine, I'll move on.

While we can model historic change points, it's a bit more difficult to model future change points; there are instances where an upcoming change is known.

For instance, the UK introduced the Civil Liability Act which made changes to the personal injury compensation system in England and Wales. If you were to regularly model the number and cost of compensation claims for whiplash injuries like I do (for work, not pleasure), the implementation of this act in June 2021 resulted in quite a serious step change. But since it was known about in advance, it was possible to take steps to account for it.

These kinds of changes need a case-by-case approach, with pragmatism and common sense being front and centre.

Interactions

Anyone keeping count of elephants in the room? Here's another — we haven't built any features which capture the interaction between predictors.

Interactions are incredibly useful features which can capture the relationship between various predictors. An interaction occurs when an independent variable has a different effect on the outcome depending on the values of another independent variable[9].

In our case one of the more interesting motivations for using interactions would be to allow the seasonality to change over time, as we currently assume — and model — that the same seasonal effect holds true for more than twenty years. There's no obvious evidence to the contrary but we could potentially eke out more forecasting power from the model by interacting time with some features.

We'll have to add this to our list of things to do next time.

Holidays

Lastly, a quick word on holidays.

We touched on some of the headaches that the Easter weekend can give us and came up with a simple solution.

A real enhancement to the feature engineering would be the incorporation of school holidays. These will likely have an impact on the number of road traffic accidents, and so would be strong predictors.

Unfortunately it's not so easy to do as schools in the UK go on holiday at slightly different times for slightly different lengths of time. Perhaps we could get really imaginative and create a distribution of school holidays, and allocate that to each month — another one for next time.

That's it from me. I hope you enjoyed reading this as much as I enjoyed writing it.

As always, please let me know what you think — I'm really interested to hear about your experiences with Prophet or with modelling time series in different ways.

As I mentioned, I'll be tackling the modelling in a forthcoming article — keep your eyes peeled for that.

Until next time.

References and resources

1. GitHub — facebook/prophet: Tool for producing high quality forecasts for time series data that has multiple seasonality with linear or non-linear growth.
2. Vincent Warmerdam: Winning with Simple, even Linear, Models | PyData London 2018 — YouTube
3. https://roadtraffic.dft.gov.uk/downloads used under the Open Government Licence (nationalarchives.gov.uk)
4. Forecasting at scale (peerj.com)
5. A comprehensive guide to time series decomposition | Towards AI
6. Trend Changepoints | Prophet (facebook.github.io)
7. Civil-liability-act-2018-Q-and-A.docx (live.com)
8. interaction.pdf (mcgill.ca)
9. https://stackoverflow.com/a/59681727/11637704

Machine Learning
Feature Engineering
Time Series Analysis
Forecasting
Deep Dives

tds

Written by Bradley Stephen Shaw

356 Followers
·Writer for

Towards Data Science

A pricing actuary, writing about interesting intersections of actuarial science and data science | LinkedIn: https://www.linkedin.com/in/bradley-shaw-f