Bayesian Modeling for Ford GoBike Ridership with PyMC3 — Part I

Jay Franck Follow
Jan 14 · 7 min read

Photo by Andrew Gook on Unsplash

Bike shares are a large part of the transport equation for cities around the world. In San Francisco, one of the major players in the bike share game is Ford with its GoBike program. Conveniently, they kindly release their data for people like me to study. I wonder if it is possible to easily forecast the ridership of the next day in order to ensure enough bikes are available to riders, based on past information?

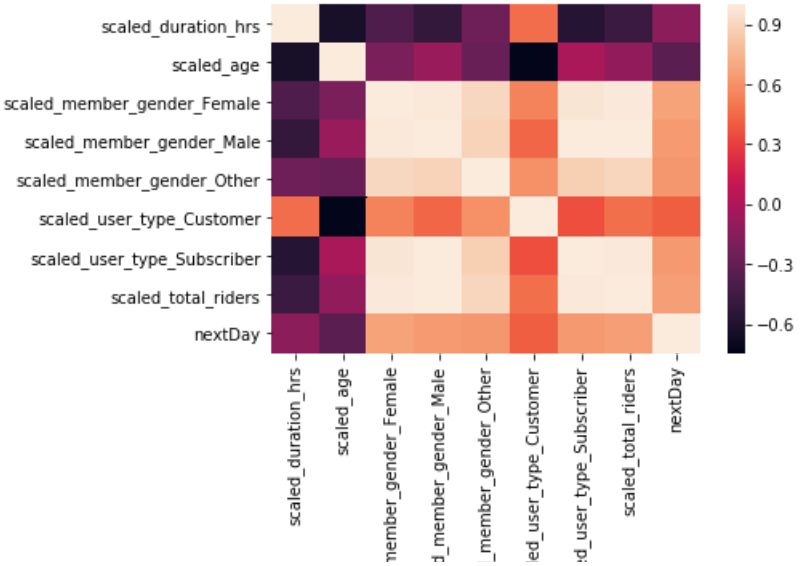
This would be a fairly trivial task to complete if I were to use sklearn to build a Linear Regression model. Often I find myself looking for data sets to learn a new tool or skill in Machine Learning. I have been trying to find an excuse to try one of the probabilistic programming packages (like PyStan or PyMC3) for years now, and this bike share data seemed like a great fit.

A number of companies are looking towards Bayesian inference for their internal predictive models. As computational cost goes down, the notoriously long training time for these systems decreases. Notably Uber released Pyro, an open source framework that can appears to be fairly flexible and easy to use. Quantopian is a frequent user of PyMC3. Booz Allen Hamilton and GoDaddy are two other firms that I am aware of that are pursuing these types of ML models.

```
aggregations = {
    'duration_hrs':'mean',
    "age" :"mean",
    "member_gender_Female":"sum",
    "member_gender_Male":"sum",
    "member_gender_Other":"sum",
    "user_type_Customer":"sum",
    "user_type_Subscriber":"sum",
}
day=df.groupby("start_day").agg(aggregations)
day.head()
```

In this post I have constructed a simple example that I hope will be instructive to a PyMC3 beginner. I downloaded GoBike data from 2017 — September 2018. I then aggregated the data on a day-by-day basis to assemble information about the ridership details (e.g., mean age, subscription membership), the length of their rides, and the number of total riders.

The issue with these Bayesian ML tools is that they can take a long time to train, especially on massive data. For this aggregated dataset, I have only 100s of rows, which is easily trainable on a laptop. After a little research, I settled on learning PyMC3 as my package of choice. It seems like it has a number of great tutorials, a vibrant community, and a fairly easy to use framework.



I begin this illustrative example by scaling the data with a RobustScaler and plotting the seaborn correlation heatmap to see if there are any patterns our model could learn. We are attempting to predict the value of *nextDay*. From the heatmap, we can see some simple relationships between variables, both positive (*scaled_total_riders*) and negative (*scaled_duration_hrs*).

```
Let us try some baseline predictions: Naive Average of all ridership

1896.0556824966166
```

Let us make a naive prediction and use that as a baseline. What would the RMSE be if we just took the average number of daily riders and used that as our prediction? We would expect to be off by approximately 1900 riders per day by taking this easy approach.

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
lr.fit(X_train,Y_train)
preds = lr.predict(X_test)
np.sqrt(mean_squared_error(Y_test, preds))
```

```
1393.020449007788
```

A linear model with sklearn performs slightly better in RMSE, and is quite easy to implement. The model is a series of weights for each variable in our data, in addition to an intercept. How do we interpret the confidence our model has in each of those individual parameters?

This can be where the Bayesian magic really shines. It undoubtedly takes more lines of code, more thought, and longer training time than the sklearn example above. I promise you, dear reader, that it can all be worth it.

First, PyMC3 runs on Theano under the hood. We have to make some slight changes to our pandas/numpy data, and the most major change is by setting a shared tensor, as follows.

```
#We initiate a shared theano array for training and
#    testing splits for ease of use.
model_input = theano.shared(np.array(X_train))
model_output = theano.shared(np.array(Y_train))
```

When we look to make predictions for our model, we can swap out X_train for X_test and use the same variable name.

Now that we have our data set up, we need to build our model, which we initialize by calling pm.Model() . Inside of this model context, we need to build our complete set of assumption about our priors (parameters) and output. Normal (Gaussian) distributions are fairly safe bets for your first model.

```
print('Running on PyMC3 v{}'.format(pm.__version__))
```

```
big_model = pm.Model()

with big_model:
    """
    Our simple, linear model requires an intercept (alpha) and a weight
    for each of our parameters (beta). I have no prior knowledge about what
    the weights could be, but I know the features are scaled by the
    RobustScaler. Therefore, we can assume they might be of centered around
    0 (mu). In this example, we did not choose to scale our target value
    (riders the next day), so the slope of our scaled parameters might be
    quite large. Therefore, we set sd=100. If we set it too low, it may
    never find the correct beta value for that parameter.
    """
    alpha = pm.Normal('alpha', mu=0, sd=100)
    beta = pm.Normal('beta', mu=0, sd=100, shape=8)

    """
    The values we are trying to predict will be a simple dot product of our
    features with the weights of our model (transpose of beta) plus our
    intercept. We take the exponential of this value because we will
    be modelling our output as a Poisson distribution. If we modelled it
    as a Gaussian, we would remove the np.exp and introduce a value
    for the noise (sigma).
    """
    values = np.exp(alpha + T.dot(model_input, beta.T) )

    """
    This is the final output of our model. Our target variable (model_output)
    is being modelled as a Poisson, as we are dealing with a simple counting
    statistic (the number of riders the next day). We could reasonably choose
    a different distribution in PyMC3 (Normal, DiscreteUniform, et al.) but
    this seems intuitive.
    """
    Y_obs = pm.Poisson('Y_obs', mu=values, observed=model_output)
```

This constitutes our model specification. Now we have to learn what the posterior distribution of our model weights could be. Unlike sklearn, the coefficients are now a distribution of values, not a single point. We sample a range of possible weights, and the coefficients that appear to fit our data well are retained in something called a *trace*. The sampling functions (NUTS, Metropolis, *et al.*) are well beyond the scope of this post, but there

are vast repositories of knowledge describing them. Here we build our trace from our model:

```
"""
Now that we have specified our model, we need to
begin estimating our model parameters. We could specify
a sampler (Metropilis, NUTS, etc.) or we could choose
.sample(), which chooses a sampler suited to our data.
Generally it chooses NUTS.
"""
with big_model:
    start = pm.find_MAP() # Use the MAP estimate as a starting value
    nuts_trace = pm.sample(8000, scaling=start) # Begin to build our trace
```

```
/opt/conda/lib/python3.6/site-packages/pymc3/tuning/starting.py:61: UserWarning: find_MAP shoul
d not be used to initialize the NUTS sampler, simply call pymc3.sample() and it will automatica
lly initialize NUTS in a better way.
  warnings.warn('find_MAP should not be used to initialize the NUTS sampler, simply call pymc3.
sample() and it will automatically initialize NUTS in a better way.')
logp = -49,086, ||grad|| = 1.2552: 100%|████████| 43/43 [00:00<00:00, 1432.19it/s]
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [beta, alpha]
Sampling 2 chains:  84%|██████▍  | 14195/17000 [20:31<03:49, 12.24draws/s]
```

The NUTS sampler complains that using find_MAP() is not a good idea, but frequently this is used in tutorials, and did not seem to hurt my performance.

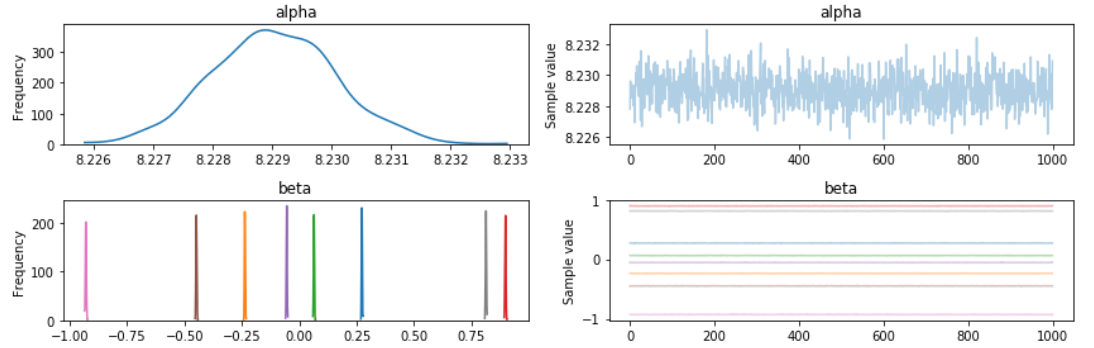
We can also try a different sampler that tries to approximate the posterior distributions:

```
"""
What if we have a large, complicated model that we can only
```

```
approximate the posterior distribution?  Use Variational Inference:
https://docs.pymc.io/api/inference.html?highlight=advi#pymc3.variational.inference.ADVI

It works extremely quickly compared to NUTS, and can offer increased performance in
estimating the model parameters for very complex distributions. Simple distributions
will be better fit by NUTS.
"""
with big_model:
    inference = pm.ADVI()
    approx = pm.fit(n=100000, method=inference)
```

```
Average Loss = 49,137:  93%|[redacted]| 93439/100000 [00:53<00:03, 1754.00it/s]
```



The trace can be plotted, and generally looks like this. The *beta* parameters look fairly constrained in distribution (left plots) and seem to be reasonably consistent across the last 1000 sampled items in our trace (right plot). The *alpha* parameter looks less certain.

```
]: pm.summary(advi_trace[-1000:])
```

]:

	mean	sd	mc_error	hpd_2.5	hpd_97.5
alpha	8.229016	0.001038	0.000034	8.227141	8.231264
beta__0	0.272588	0.001570	0.000051	0.269436	0.275513
beta__1	-0.226587	0.001654	0.000051	-0.229865	-0.223425

beta_1	-0.236587	0.001654	0.000051	-0.239865	-0.233435
beta_2	0.064149	0.001733	0.000051	0.060740	0.067520
beta_3	0.897056	0.001884	0.000060	0.893289	0.900494
beta_4	-0.054657	0.001656	0.000055	-0.058255	-0.051722
beta_5	-0.447547	0.001863	0.000054	-0.450896	-0.443626
beta_6	-0.928436	0.002046	0.000063	-0.932394	-0.924150
beta_7	0.812484	0.001812	0.000065	0.808800	0.816012

Now that we have our posterior samples, we can make some predictions. We generally observe a so-called 'burn in' period in PyMC3 where we discard the first thousand samples of our trace (*trace[1000:]*), as these values may not have converged. We than draw 1000 sample weights from this trace, calculate what the predictions might be, and take the mean of that value as our most probable prediction for that data point. From here, we simply calculate the RMSE.

```
"""
Let's get the RMSE of our model as understood by the NUTS and ADVI
sampler. We will test our training data error first.

The NUTS sampler does a pretty decent job on predictions, and has a
better RMSE than our sklearn LR model.
"""

def scoreModel(trace,y,model_name):
    ppc = pm.sample_ppc(trace[1000:], model=model_name, samples=1000)
    pred = ppc['Y_obs'].mean(axis=0)
    return np.sqrt(mean_squared_error(y, pred))

scoreModel(nuts_trace,Y_train,big_model)
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:10: DeprecationWarning: sample_ppc
() is deprecated.  Please use sample_posterior_predictive()
  # Remove the CWD from sys.path while we load stuff.
100%|██████████| 1000/1000 [00:10<00:00, 92.63it/s]
```

```
1260.7169755936254
```

If we want to test on our holdout dataset :

```
"""
Now we simply switch out our Theano tensor with our
testing data. Using the shared value, we do not have
to respecify our model first.
"""
model_input.set_value(np.array(X_test))
model_output.set_value(np.array(Y_test))

scoreModel(nuts_trace,Y_test,big_model)
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:10: DeprecationWarning: sample_ppc
() is deprecated.  Please use sample_posterior_predictive()
  # Remove the CWD from sys.path while we load stuff.
100%|██████████| 1000/1000 [00:10<00:00, 100.96it/s]

1449.8860838107016
```

So this model that we built performs better than our naive approach (average ridership) but slightly worse than our sklearn model. In an included example in the Github repo, I was able to build a similar model that beat the sklearn model by scaling the Y value, and modeling it as a Normally distributed variable.

Further tuning of the model parameters, using different scalings, assuming a wider range of possible beta parameters can all be employed to lower the RMSE of this example. The goal of this post is to introduce the basics of model building and provide an editable example that you can play around

with and learn from! I encourage you to provide feedback in the comments section below.

To recap, there is a price to pay for Bayesian models. It certainly takes longer to implement and write a model. It requires some background knowledge on Bayesian statistics. The training time is orders of magnitude longer than using sklearn. However, tools like PyMC3 can offer greater control, understanding, and appreciation for your data and the model artifacts.

Although there are a number of good tutorials in PyMC3 (including its documentation page) the best resource I found was a video by Nicole Carlson. It explores how a sklearn-familiar data scientist would build a PyMC3 model. Careful readers will find numerous examples that I adopted from that video. I also learned a lot from Probabilistic Programming and Bayesian Methods for Hackers, which is a free notebook based tutorial on practical Bayesian models using PyMC3. These two resources are absolutely amazing. Duke also has an example website that has numerous data situations that I found informative. Towards Data Science has also hosted a number of cool posts throughout the year that focused on Bayesian analysis and have helped inspire this post.

In the next blog post, I will illustrate how to build a Hierarchical Linear Model (HLM) that will greatly improve the performance of our initial approach. Below are a Kaggle kernel that you can fork and a Github repo that you can clone to play around with the data and develop your own PyMC3 models with. Thank you for reading!

DayByDayPredictions | Kaggle

Edit description

www.kaggle.com

Github Repo

Machine Learning Pymc3 Data Science Bayesian Machine Learning Bayesian Statistics