Modelling a Time Series of Records with PyMC3

Jaime Sevilla[1] and Jonathan Lindbloom[2]

[1]Aberdeen University
[2]Dartmouth College

September 20, 2021

Abstract

We present a Bayesian approach for modeling a time series for a cumulative record that takes the form of the maximum or minimum of a sequence of attempts, in the absence of data for the underlying attempts. We discuss the derivation of the likelihood function, sampling of the posterior via PyMC3, and forecasting the distribution of future records.

How often and by how much are Olympic records beat? What score do we expect future machine learning systems to attain for classification tasks in the absence of new breakthroughs? With what probability will the fasted speed run for our favorite videogame be beaten within the next year?

This article is a tutorial on how to use the probabilistic programming language (PPL) PyMC3 to model how records are set over time. While in many cases one may be able to make inferences about a record indirectly by first fitting a model for attempts at breaking this record and calculating statistics for the record after the fact, *we specifically consider the case that the only data available is the time series for the record itself.*

The article is structured as follows:

1. First, we generate some artificial data for a running maximum.
2. Second, we derive the mathematical formula for the likelihood function of our model.
3. Third, we demonstrate how to fit our model to data using PyMC3.
4. To check our fitted model, we show how to compute the posterior predictive distribution of the cumulative maximum.
5. Finally, we show how to generate probabilistic forecasts of the record in future periods.

A companion Google Colab notebook for this article can be found here. It contains all the code you would need to fit your model to a dataset. Feel free to reuse our code for your own projects!

The Model

Suppose we would like to model a time series for the cumulative record for some task, where the record is the maximum or minimum over some sequence of attempts. In this article, we'll only consider the case where the record is a maximum since the approach for the minimum requires only a slight modification. Let $\{X_t\}_{t \in \mathbb{N}}$ denote a discrete-time stochastic process representing the results from some sequence of attempts at a task. We assume that the X_i are *i.i.d.* according to some random variable X with a common CDF given by F_X and PDF given by f_X. We refer to F_X as the *attempt distribution* for our record.

We assume our observed data takes the form of a time series $\{r_1, r_2, \ldots, r_n\}$ where r_i is the record as of the ith period and n is the number of periods for which the record has been observed. In the case the record

is of a maximum, note that we must have $r_i \leq r_j$ whenever $i \leq j$. To match our observed data, we define a sequence $\{Y_t\}_{t \in \mathbb{N}}$ where

$$Y_i := \max\{X_1, \ldots, X_i\}.$$

We treat $\{r_1, r_2, \ldots, r_n\}$ as noiseless, truncated observations along some sample path $\omega = \{r_1, r_2, \ldots, r_n, \ldots\}$.

1. Synthetic Data for a Running Maximum

To generate some artificial data, let us take our attempt distribution to be a Gaussian with

$$X \sim \mathcal{N}(\mu, \sigma^2)$$

and generate a time series of length $n = 100$. We note that this approach can be applied to scenarios with arbitrary attempt distributions and not just that of our particular case of a Gaussian. To produce a sample path from the record process, we begin by simulating an attempt a_1 from the attempt distribution and set $r_1 = a_1$. The second attempt a_2 is simulated in the same manner, and following **??** we set our record as of the second period as $r_2 = \max\{a_1, a_2\}$. We generate the rest of the sample path in the same manner, where $r_{i+1} = r_i$ unless $a_{i+1} > r_i$, in which case the record is broken and $r_{i+1} = a_{i+1}$.

Our goal is to infer the posterior distribution on μ and σ given only the sequence of the cumulative records $\{r_1, r_2, \ldots, r_n\}$, blind to the sequence of attempts $\{a_1, a_2, \ldots, a_n\}$. Let us fix

$$\mu = 10, \;\; \sigma = 1,$$

and simulate a sample path for the record.

```
import numpy as np
import matplotlib.pyplot as plt

# Parameters
n_periods = 100 # Number of periods to observe
mu_true = 10.0 # Mean of the underlying distribution
sigma_true = 1.0 # Standard deviation  of the underlying distribution
np.random.seed(4) # Fix the seed to reproduce our plots

# Sample generation
index = np.arange(n_periods)
sample_path = []
record = -np.math.inf
for i in index:
  attempt_new = mu_true + sigma_true*np.random.randn() # Draw a new sample
  record_new = np.max([attempt_new, record]) # Update the record
  sample_path.append(record_new)
  record = record_new

# Plot
plt.plot(index, sample_path, color="red")
plt.xlabel("Period")
plt.ylabel("Record")
plt.title("One Sample Path")
```

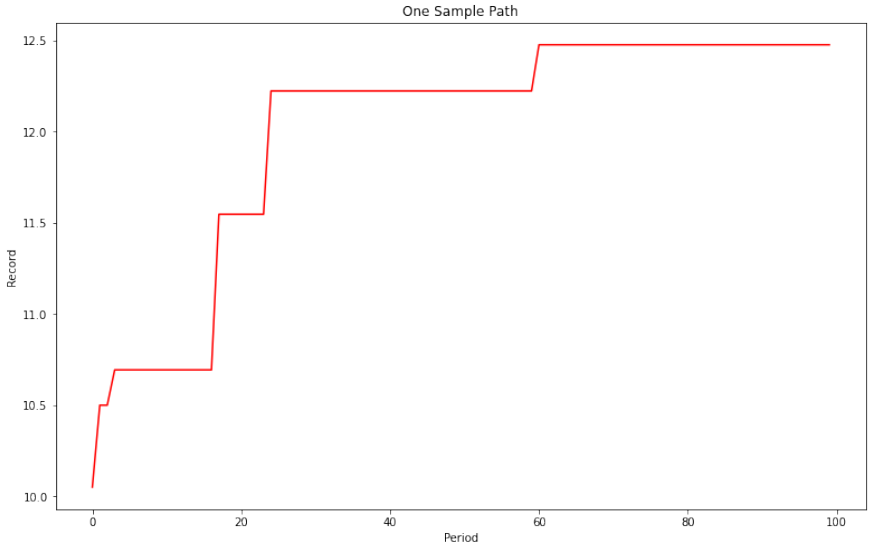

Figure 1: 100 periods of one sample path for the record. Here our underlying attempt distribution is a Gaussian with mean of 10 and a variance of 1. As we expect, the record becomes harder to beat as it approaches the tail of the attempt distribution.

While this is a straightforward way to simulate a sample path, note that we can use the fact that all of the attempts are independent to vectorize this code.

```
# Vectorized version of the above
np.random.seed(4)
attempts = mu_true + sigma_true*np.random.randn(n_periods)
sample_path = np.maximum.accumulate(attempts)
```

As we expect, the record path is monotonically increasing since it is the maximum over an increasing number of attempts. While the record is broken several times in the early periods, as time progresses the record is broken less frequently. While with real-world data we obviously only get to observe one such sample path, according to our stochastic model, it is but one of many possible trajectories we could have observed. Let us draw many alternative sample paths to characterize the distribution over records.

```
# Generate many sample paths
np.random.seed(4)
n_paths = 10000
attempts = mu_true + sigma_true*np.random.randn(n_paths, 100)
sample_paths = np.maximum.accumulate(attempts, axis=1)

# Calculate the 1%, 10%, 50%, 90%, and 99% quantiles
lower_bound_one = np.quantile(sample_paths, q=0.01, axis=0)
lower_bound_ten = np.quantile(sample_paths, q=0.1, axis=0)
medians = np.quantile(sample_paths, q=0.5, axis=0)
upper_bound_ninety = np.quantile(sample_paths, q=0.9, axis=0)
upper_bound_ninety_nine = np.quantile(sample_paths, q=0.99, axis=0)

# Plot
fig, axs = plt.subplots(1, 2, figsize=(13,8))
```

```
# Plot sample paths on the left
axs[0].plot(index, sample_paths[1:,:].T, alpha=0.05)
axs[0].plot(index, sample_paths[0,:], color="red", label="Observed Sample Path")
axs[0].legend()
axs[0].set_ylim(7.5, 15)
axs[0].set_xlabel("Period")
axs[0].set_ylabel("Record")
axs[0].set_title("Many Alternative Sample Paths")

# Plot CI on the right
axs[1].fill_between(index, lower_bound_one, upper_bound_ninety_nine, alpha=0.4, label="99% CI", color="C
axs[1].fill_between(index, lower_bound_ten, upper_bound_ninety, alpha=0.7, label="80% CI", color="C0")
axs[1].plot(index, medians, label="Median")
axs[1].plot(index, sample_paths[0,:], color="red", label="Observed Sample Path")
axs[1].legend()
axs[1].set_ylim(7.5, 15)
axs[1].set_xlabel("Period")
axs[1].set_ylabel("Record")
axs[1].set_title("Credible Interval Over Sample Paths")

fig.tight_layout()
```

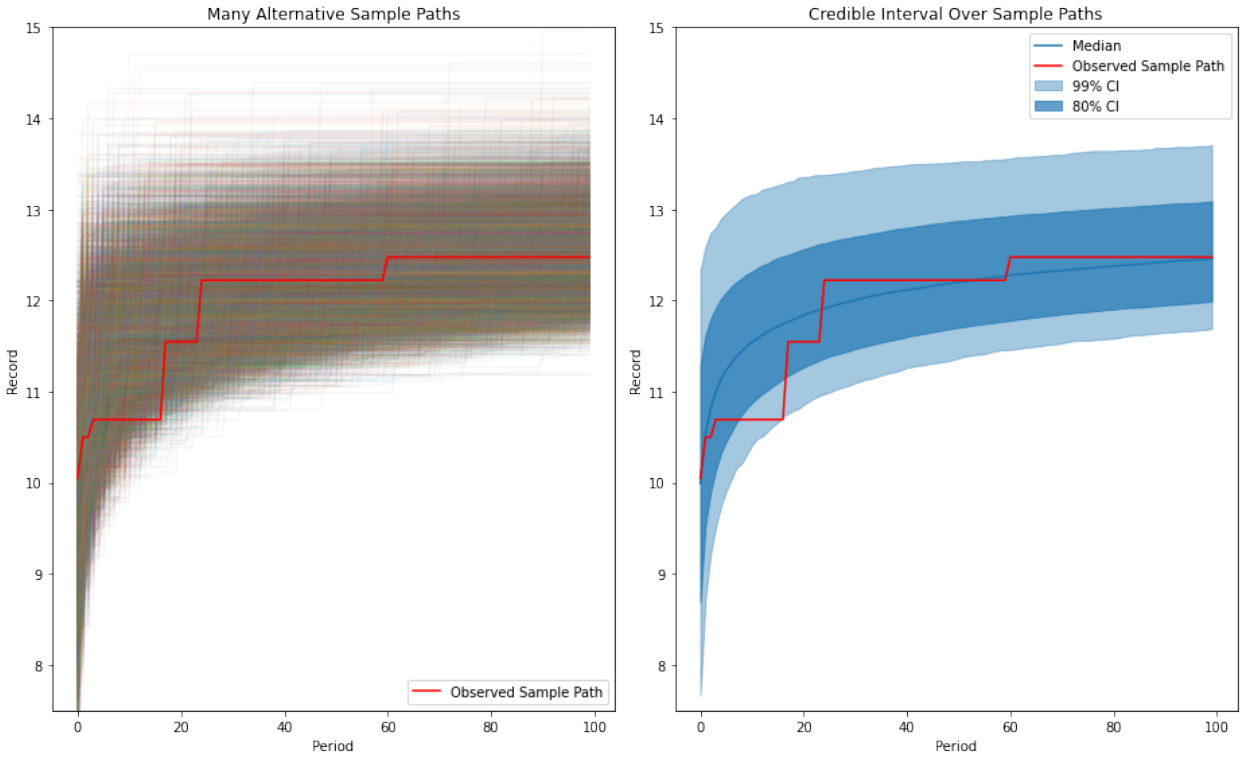

Figure 2: Left: many alternative sample paths drawn according to our stochastic model for the record process. Right: credible intervals over these sample paths at each period.

The point of the above is that even if we had perfect knowledge about the parameters μ and σ of our model, there is still a wide range of behavior we might expect when drawing samples from the record process and we can calculate probabilities to express this variability. This is known as aleatoric uncertainty, as opposed to parametric uncertainty. But perhaps this is not very interesting, since we are likely concerned with the record at some future period instead of in the past. In a similar manner, we can extrapolate into the future by drawing sample paths of the record process conditioned on the observations.

```
# Forecast parameters
n_fcast_periods = 100
n_fcast_paths = 10000

# Calculate new index
last_idx = index[-1]
fcast_index = np.arange(last_idx, last_idx+1+n_fcast_periods, 1)
new_index = np.concatenate((index, fcast_index))

# Draw future attempts
attempts = mu_true + sigma_true*np.random.randn(n_fcast_paths, n_fcast_periods)

# Add a column filled with the last observation of the record
last_record = sample_paths[0,-1]
last_record_col = last_record*np.ones(n_fcast_paths)[:,None]
attempts = np.concatenate((last_record_col, attempts), axis=1)
fcast_paths = np.maximum.accumulate(attempts, axis=1)

# Calculate bands for the CIs
lower_bound_one = np.quantile(fcast_paths, q=0.01, axis=0)
lower_bound_ten = np.quantile(fcast_paths, q=0.1, axis=0)
medians = np.quantile(fcast_paths, q=0.5, axis=0)
upper_bound_ninety = np.quantile(fcast_paths, q=0.9, axis=0)
upper_bound_ninety_nine = np.quantile(fcast_paths, q=0.99, axis=0)

# Plot
fig, axs = plt.subplots(1, 2, figsize=(13,8))
axs[0].plot(index, sample_paths[0,:], color="red", label="Observed Sample Path")
axs[1].plot(index, sample_paths[0,:], color="red", label="Observed Sample Path")

# Plot sample paths on the left
axs[0].plot(fcast_index, fcast_paths.T, alpha=0.05)
axs[0].legend()
axs[0].set_ylim(9.5, 15)
axs[0].set_xlabel("Period")
axs[0].set_ylabel("Record")
axs[0].set_title("Many Forecasted Sample Paths")

# Plot CI on the right
axs[1].fill_between(fcast_index, lower_bound_one, upper_bound_ninety_nine, alpha=0.4, label="99% CI", co
axs[1].fill_between(fcast_index, lower_bound_ten, upper_bound_ninety, alpha=0.7, label="80% CI", color="
axs[1].plot(fcast_index, medians, label="Median", color="Green")
axs[1].legend()
axs[1].set_ylim(9.5, 15)
axs[1].set_xlabel("Period")
```

```
axs[1].set_ylabel("Record")
axs[1].set_title("Credible Interval Over Forecasted Sample Paths")
```

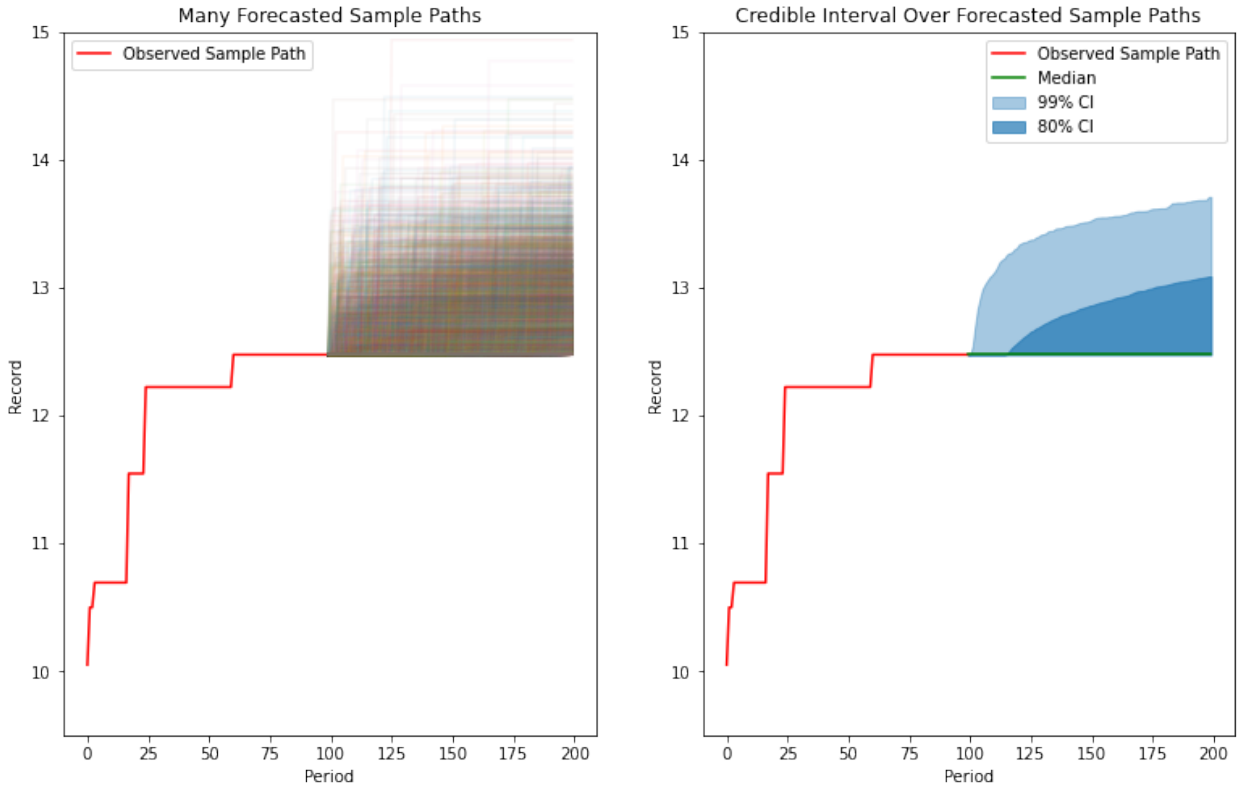

Figure 3: Left: many alternative forward-looking sample paths drawn according to our model. Right: credible intervals over these sample paths at each period. The fact that the median record (shown in green) is still unchanged by the 200th period can be interpreted as there being at least a 50% chance that the last observed record (as of the 100th period) will not be broken by the 200th period.

Note that this analysis is predicated on knowing the true values of μ and σ that generated our time series for the record. In what follows we consider the task of performing Bayesian inference to recover a posterior distribution on these two parameters given our observed time series.

2. Deriving the Likelihood Function

In order to perform Bayesian inference, we need to know the likelihood function associated with our model that expresses the relative likelihood of an observed record time series for any choice of model parameters μ and σ. If you are not interested in the derivation, simply look at equation **??** and skip to the next section.

Let $Y_{1:m}$ denote the collection of random variables $Y_1, \ldots, Y_m$ and θ denote the collection of model parameters. Since

$$Y_{j+1} amp; = \max\{X_1, \ldots, X_j, X_{j+1}\}$$
$$amp; = \max\{Y_j, X_{j+1}\}$$

we know that Y_{j+1} must be independent of all of the Y_i for $i < j$. Then we can express the likelihood function of our model as

$$\mathcal{L}_{Y_{1:n}}(y_1, \ldots, y_n|\theta) = \mathcal{L}_{Y_1}(y_1|\theta) \prod_{j=1}^{n-1} \mathcal{L}_{Y_{j+1}|Y_j=y_j}(y_{j+1}|\theta)$$

where $\mathcal{L}_{Y_{j+1}|Y_j=y_j}(y_{j+1}|\theta)$ is the likelihood function for the random variable $Y_{j+1}|Y_j = y_j$. It turns out that this conditioned likelihood islikelihood is

$$\mathcal{L}_{Y_{j+1}|Y_j=y_j}(y_{j+1}|\theta) = \begin{cases} F_X(y_j), & amp; \text{if } y_{j+1} = y_j, \\ f_X(y_{j+1}), & amp; \text{if } y_{j+1} > y_j, \\ 0, & amp; \text{otherwise}, \end{cases}$$

which is a bit complicated, but that the likelihood function for the entire time series is much simpler.

Proposition 1. Let $X_1, \ldots, X_n$ be a collection of *i.i.d.* continuous random variables with common PDF f_X and CDF F_X, and define $Y_j := \max_{i \leq j} X_i$. Then the joint likelihood for the sequence $Y_1, \ldots, Y_N$ is given by

$$\mathcal{L}_{Y_{1:n}}(y_1, \ldots, y_N) = \prod_{i \in R} f_X(y_i) \prod_{i \notin R} F_X(y_i)$$

where $R \subseteq \{1, \ldots, n\}$ is the set of indices where the record was broken and a new maximum was established.

Proof: see the appendix.

Equation **??** is the key ingredient needed for performing inference of the model parameters. We will use it in the next section, where we demonstrate how to implement the model within the PyMC3 framework.

3. Fitting a Bayesian Model using PyMC3

PyMC3 is a popular probabilistic programming language written for Python that can be used to implement and fit Bayesian probabilistic models to data. In this section, we demonstrate how to use our formula for the likelihood function to fit a model within the PyMC3 framework.

Note that while typically probabilistic programming languages such as PyMC3 are capable of assembling the likelihood function for a model without the user needing to derive the analytic formula themselves, in our case the likelihood function cannot be handled as conveniently. We resort to deriving the likelihood function by hand since we will need to provide it directly to PyMC3. To do so, we will use the `pm.DensityDist` class to implement our custom distribution.

Looking at equation **??**, we see that the likelihood is split into the contributions from the timesteps where the record was broken and those where it wasn't. To make our implementation easier, we will split up the data accordingly in a pre-processing step.

```
### Separate the data where there were jumps in the running max
### and the data where the running max stayed constant
jump_mask = np.insert(np.diff(sample_path) > 0,0,True)
jump_data = obss[jump_mask] # Slice out the data where a new record is set
flat_data = obss[~jump_mask] # Slice the data where the record is maintained
```

Next, we implement the log-likelihood for the joint distribution of the data. To take advantage of the gradient-based No U-Turn Sampler (NUTS) provided by PyMC3, we will need to write out the log-likelihood using the PyMC3 equivalents of vanilla NumPy /SciPy functions.

```
import pymc3 as pm

# Implementation of the log-likelihood for the record data
def logp(jump_data, flat_data, mu, sigma):
  # Attempt distribution
  x_dist = pm.Normal.dist(mu=mu, sigma=sigma)

  # Add likelihood contribution from the jump data
  log_likelihood = pm.math.sum(x_dist.logp(jump_data))

  # Add likelihood contribution from the flat data
  log_likelihood += pm.math.sum(x_dist.logcdf(flat_data))

  return log_likelihood
```

This log-likelihood is specific to the case of an underlying normal distribution for the attempts, but can be easily adapted to other attempt distributions. As an aside, if you want to fit a minimum instead of a maximum, use `pm.math.log1mexp(-x_dist.logcdf(flat_data))` instead of `x_dist.logcdf(flat_data)`. Now we assemble the model.

```
# PyMC model definition
with pm.Model() as model:
  mu = pm.Normal('mu', mu=12.0, sigma=3.0)
  sigma = pm.Exponential('sigma', lam=1.0)
  likelihood = pm.DensityDist('running_max', logp, observed = {'jump_data':jump_data,
                                                               'flat_data':flat_data,
                                                               'mu': mu,
                                                               'sigma': sigma})
```

The choice of priors for the parameters `mu` and `sigma` were arbitrarily chosen such that most of the mass was concentrated in the ballpark of the true values of 10 and 1. In a real application, you obviously don't have this same luxury, so you should carefully choose the parameters priors to reflect your previous beliefs about the problem and the underlying attempt distribution. In practice, we highly recommend simulating data from the prior predictive when choosing priors.

Now we can use the default NUTS sampler method to fit the posterior distribution of the parameters and plot the posterior using ArviZ:

```
with model:
  trace = pm.sample(draws=20000, chains=5, tune=5000, target_accept=0.99,
                    return_inferencedata=True,
                    idata_kwargs={"density_dist_obs": False})
pm.plot_trace(trace, ['mu', 'sigma'])
```

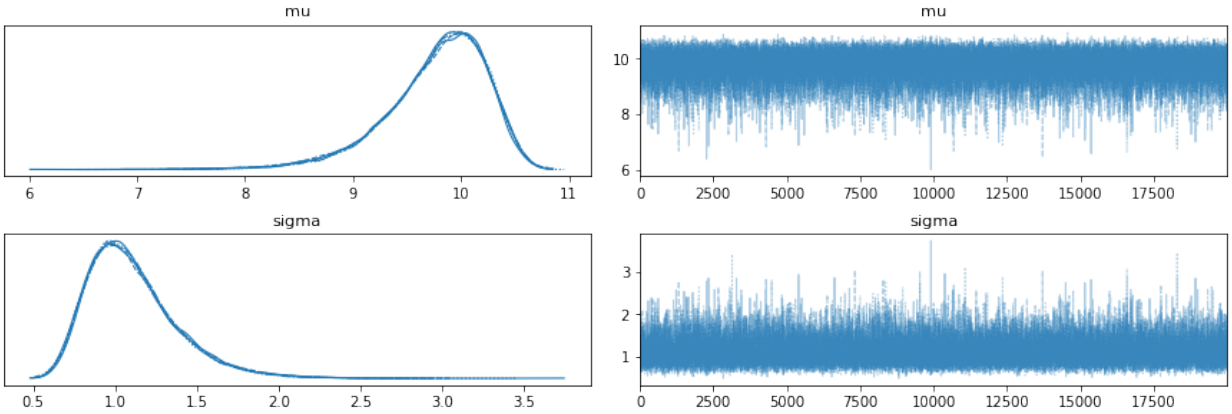

Figure 4: A trace plot for the model posterior.

Note that we have increased the `target_accept`parameter of the NUTS sampler. We have done this to address the divergences we observed without doing this, which we believe is an artifact of fitting to a single sample path rather than many. In our experiments, fitting our model to many sample paths in parallel removes all divergences without the need to increase `target_accept`and correctly recovers the true parameters.

4. Sampling the Posterior Predictive Distribution

As a sanity check for our model fit, it is useful to compare the actual observations to the posterior predictive distribution. In our case, this is the distribution of the cumulative maximum implied by the posterior distribution on our model parameters μ and σ.

In order to take advantage of the convenient `pm.sample_posterior_predictive` function while using a custom likelihood via `pm.DensityDist`, we just need to implement a function `random`that specifies how to sample a time series of records given the model parameters.

```
# Random sampling function
def random(n_periods=100, past_obs=None):

  def _random(point=None, size=None, n_periods=n_periods, past_obs=past_obs):
      mu, sigma = point['mu'], point['sigma']
      attempts = mu + sigma*np.random.randn(n_periods)

      if past_obs is not None:
        last_obs = np.atleast_1d(past_obs[-1])
        attempts = np.concatenate([last_obs, attempts])
        sample_path = np.maximum.accumulate(attempts)[1:]
        full_sample_path = np.concatenate([last_obs, sample_path])
        return full_sample_path
      else:
        sample_path = np.maximum.accumulate(attempts)
        return sample_path
```

```
  return _random
```

While we won't use this feature quite yet, this `random`function can also handle sampling for forecasting. Now we can update our model definition accordingly and sample the posterior predictive distribution.

```
# PyMC3 model definition
with pm.Model() as model:
  mu = pm.Uniform('mu', 0., 20.)
  sigma = pm.Uniform('sigma', 0.5, 1.5)
  likelihood = pm.DensityDist('running_max', logp, random=random, # Here we incorporate the sampling fun
                      observed = {'jump_data':jump_data,
                                  'flat_data':flat_data,
                                  'mu': mu,
                                  'sigma': sigma}
                      )

# Sample posterior distribution
with model:
  trace = pm.sample(draws=20000, chains=5, tune=5000, target_accept=0.99,
                    return_inferencedata=True,
                    idata_kwargs={"density_dist_obs": False})

# Generate predictive posterior
with model:
  post_pred = pm.sample_posterior_predictive(trace, var_names=["running_max"])

## Plot
sample_paths = post_pred["running_max"]

# Calculate the 1%, 10%, 50%, 90%, and 99% quantiles
lower_bound_one = np.quantile(sample_paths, q=0.01, axis=0)
lower_bound_ten = np.quantile(sample_paths, q=0.1, axis=0)
medians = np.quantile(sample_paths, q=0.5, axis=0)
upper_bound_ninety = np.quantile(sample_paths, q=0.9, axis=0)
upper_bound_ninety_nine = np.quantile(sample_paths, q=0.99, axis=0)

# Plot
fig, axs = plt.subplots(1, 2, figsize=(13,8))

# Plot sample paths on the left
axs[0].plot(index, sample_paths[:10000,:].T, alpha=0.05)
axs[0].plot(index, sample_path, color="red", label="Observed Sample Path")
axs[0].legend()
axs[0].set_ylim(7.5, 15)
axs[0].set_xlabel("Period")
axs[0].set_ylabel("Record")
axs[0].set_title("Many Posterior Predictive Sample Paths")


# Plot CI on the right
axs[1].fill_between(index, lower_bound_one, upper_bound_ninety_nine, alpha=0.4, label="99% CI", color="C
axs[1].fill_between(index, lower_bound_ten, upper_bound_ninety, alpha=0.7, label="80% CI", color="C0")
```

```
axs[1].plot(index, medians, label="Median")
axs[1].plot(index, sample_path, color="red", label="Observed Sample Path")
axs[1].legend()
axs[1].set_ylim(7.5, 15)
axs[1].set_xlabel("Period")
axs[1].set_ylabel("Record")
axs[1].set_title("Credible Interval Over Posterior Predictive Sample Paths")

fig.tight_layout()
```

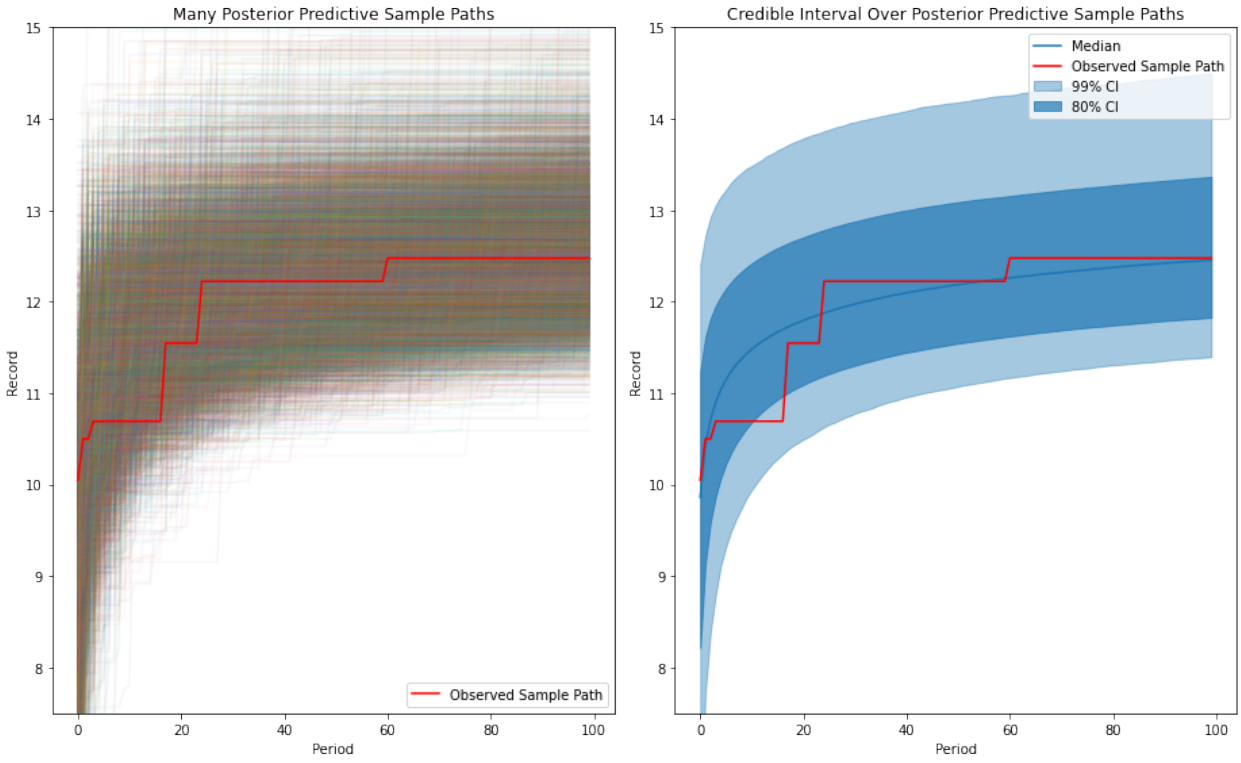

Figure 5: Left: many alternative sample paths for the record drawn from the posterior predictive distribution. Right: credible intervals over these sample paths at each period.

One natural comparison to make is how our posterior predictive distribution on sample paths compares with the same distribution when the true parameters are known exactly.

```
import matplotlib.patheffects as pe

# Plot
fig, axs = plt.subplots(figsize=(13,8))

# First plot the posterior predictive
sample_paths = post_pred["running_max"]
```

```
# Calculate the 1%, 10%, 50%, 90%, and 99% quantiles
lower_bound_one = np.quantile(sample_paths, q=0.01, axis=0)
lower_bound_ten = np.quantile(sample_paths, q=0.1, axis=0)
medians = np.quantile(sample_paths, q=0.5, axis=0)
upper_bound_ninety = np.quantile(sample_paths, q=0.9, axis=0)
upper_bound_ninety_nine = np.quantile(sample_paths, q=0.99, axis=0)

# Plot CI on the right
axs.fill_between(index, lower_bound_one, upper_bound_ninety_nine, alpha=0.4, label="99% CI - Posterior",
axs.fill_between(index, lower_bound_ten, upper_bound_ninety, alpha=0.7, label="80% CI - Posterior", colo
axs.plot(index, medians, label="Median - Posterior", lw=3.0, color="C0", path_effects=[pe.Stroke(linewid

# Next plot the exact distribution
n_paths = 10000
attempts = mu_true + sigma_true*np.random.randn(n_paths, 100)
sample_paths = np.maximum.accumulate(attempts, axis=1)

# Calculate the 1%, 10%, 50%, 90%, and 99% quantiles
lower_bound_one = np.quantile(sample_paths, q=0.01, axis=0)
lower_bound_ten = np.quantile(sample_paths, q=0.1, axis=0)
medians = np.quantile(sample_paths, q=0.5, axis=0)
upper_bound_ninety = np.quantile(sample_paths, q=0.9, axis=0)
upper_bound_ninety_nine = np.quantile(sample_paths, q=0.99, axis=0)

# Plot CI on the right
axs.fill_between(index, lower_bound_one, upper_bound_ninety_nine, alpha=0.4, label="99% CI - Exact", col
axs.fill_between(index, lower_bound_ten, upper_bound_ninety, alpha=0.7, label="80% CI - Exact", color="C
axs.plot(index, medians, label="Median - Exact", lw=3.0, color="C2", path_effects=[pe.Stroke(linewidth=5
axs.plot(index, sample_path, color="red", label="Observed Sample Path")
axs.legend(loc="lower right")
axs.set_ylim(7.5, 15)
axs.set_title("Exact vs. Posterior Credible Interval Comparison")
axs.set_xlabel("Period")
axs.set_ylabel("Record")

fig.tight_layout()
```

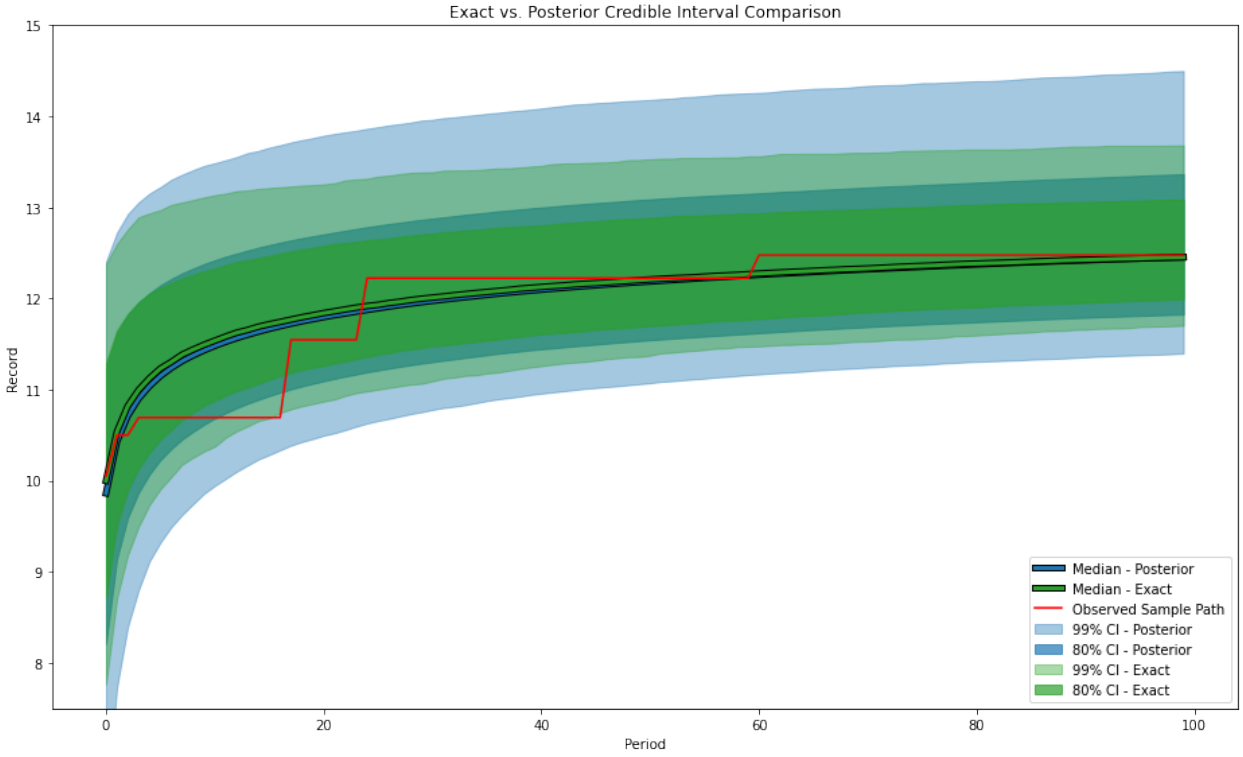

Figure 6: A comparison of credible intervals from the exact model (green) and the posterior predictive (blue). The exact model exhibits aleatoric uncertainty, whereas the posterior model exhibits both aleatoric and parametric uncertainty. In the limit of a large number of observations, the posterior credible intervals will converge to the exact credible intervals.

The posterior predictive distribution helps us characterize the alternative sample paths we would have expected to see, should the "rolls of the dice" had turned out differently. Now we turn our attention to forecasting using the posterior.

5. Predicting Future Records

Given samples of the posterior distribution of the parameters, we can easily generate samples of the forward-looking posterior predictive distribution conditioned on the past observations we used to fit our model. We'll recycle our `random`function from earlier to achieve this.

```
n_fcast_periods = 100
forecast_sampler = random(n_fcast_periods, past_obs = sample_path)

posterior = trace['posterior']
n_samps_subset = 20000
mus = np.random.choice(posterior.mu.values.flatten(), size=n_samps_subset, replace=False)
sigmas = np.random.choice(posterior.sigma.values.flatten(), size=n_samps_subset, replace=False)
```

```
sample_paths = np.zeros((n_samps_subset, len(sample_path)+n_fcast_periods))
for j in range(n_samps_subset):
  sample_paths[j,:] = forecast_sampler( point = {"mu": mus[j], "sigma":sigmas[j]} )
```

Similar to how we just compared the posterior versus exact distributions over the past periods, we can compare the posterior versus exact forecasting distributions.

```
# Calculate new index
last_idx = index[-1]
fcast_index = np.arange(last_idx, last_idx+n_fcast_periods, 1)
new_index = np.concatenate((index, fcast_index))

# Plot
fig, axs = plt.subplots(figsize=(13,8))

# First plot the posterior predictive

# Calculate the 1%, 10%, 50%, 90%, and 99% quantiles
lower_bound_one = np.quantile(sample_paths, q=0.01, axis=0)
lower_bound_ten = np.quantile(sample_paths, q=0.1, axis=0)
medians = np.quantile(sample_paths, q=0.5, axis=0)
upper_bound_ninety = np.quantile(sample_paths, q=0.9, axis=0)
upper_bound_ninety_nine = np.quantile(sample_paths, q=0.99, axis=0)

# Plot posterior
axs.fill_between(new_index, lower_bound_one, upper_bound_ninety_nine, alpha=0.4, label="99% CI - Posteri
axs.fill_between(new_index, lower_bound_ten, upper_bound_ninety, alpha=0.7, label="80% CI - Posterior",
axs.plot(new_index, medians, label="Median - Posterior", lw=3.0, color="C0", path_effects=[pe.Stroke(lin

# # Plot exact
n_paths = 20000
exact_sample_paths = np.zeros((n_paths, 200))
for j in range(n_paths):
  attempts = mu_true + sigma_true*np.random.randn(100)
  last_obs = np.atleast_1d(sample_path[-1])
  attempts = np.concatenate([last_obs, attempts])
  exact_sample_path = np.maximum.accumulate(attempts)[1:]
  exact_sample_path = np.concatenate([sample_path, exact_sample_path])
  exact_sample_paths[j,:] = exact_sample_path


# # Calculate the 1%, 10%, 50%, 90%, and 99% quantiles
lower_bound_one = np.quantile(exact_sample_paths, q=0.01, axis=0)
lower_bound_ten = np.quantile(exact_sample_paths, q=0.1, axis=0)
medians = np.quantile(exact_sample_paths, q=0.5, axis=0)
upper_bound_ninety = np.quantile(exact_sample_paths, q=0.9, axis=0)
upper_bound_ninety_nine = np.quantile(exact_sample_paths, q=0.99, axis=0)

axs.fill_between(new_index, lower_bound_one, upper_bound_ninety_nine, alpha=0.4, label="99% CI - Exact",
axs.fill_between(new_index, lower_bound_ten, upper_bound_ninety, alpha=0.7, label="80% CI - Exact", colo
axs.plot(new_index, medians, label="Median - Exact", lw=3.0, color="C2", path_effects=[pe.Stroke(linewid
axs.plot(index, sample_path, color="red", label="Observed Sample Path")
```

```
axs.legend(loc="lower right")
axs.set_ylim(9.5, 15)
axs.set_title("Exact vs. Posterior Forecasting Credible Interval Comparison")
axs.set_xlabel("Period")
axs.set_ylabel("Record")

fig.tight_layout()
```

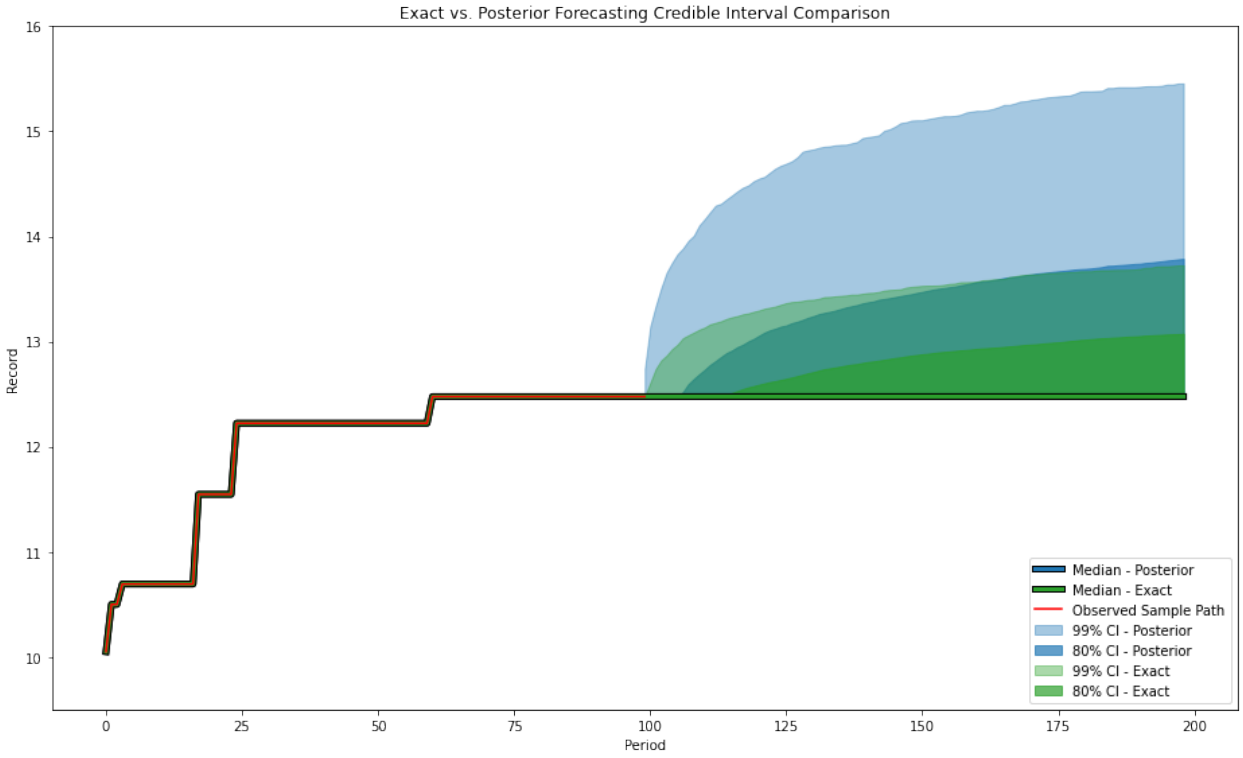

Figure 7: A comparison of credible intervals for the forecasted distribution from the exact model (green) and the posterior predictive (blue). The posterior distribution places higher probabilities on increases in the record than we would expect if we knew the true "data-generating" μ and σ. As we observe more periods in the sample path, in the limit the posterior probabilities will converge to the exact probabilities.

This plot helps show how the record will evolve over time, but what if we wanted to forecast the distribution at a single future period $t = 200$? We can simply slice out the samples corresponding to this time.

```
fig, axs = plt.subplots(figsize=(13,5))
axs.hist(sample_paths[:,-1], color="C0", bins=30, alpha=0.6, density=True, label="Posterior")
axs.hist(exact_sample_paths[:,-1], color="C2", bins=30, alpha=0.6, density=True, label="Exact")
axs.set_xlim(np.amin(sample_paths[:,99]), 15)
plt.xlabel("Record at Time t = 200")
plt.ylabel("Normalized Frequency")
plt.title("Record Forecast Distribution")
plt.legend()
plt.show()
```

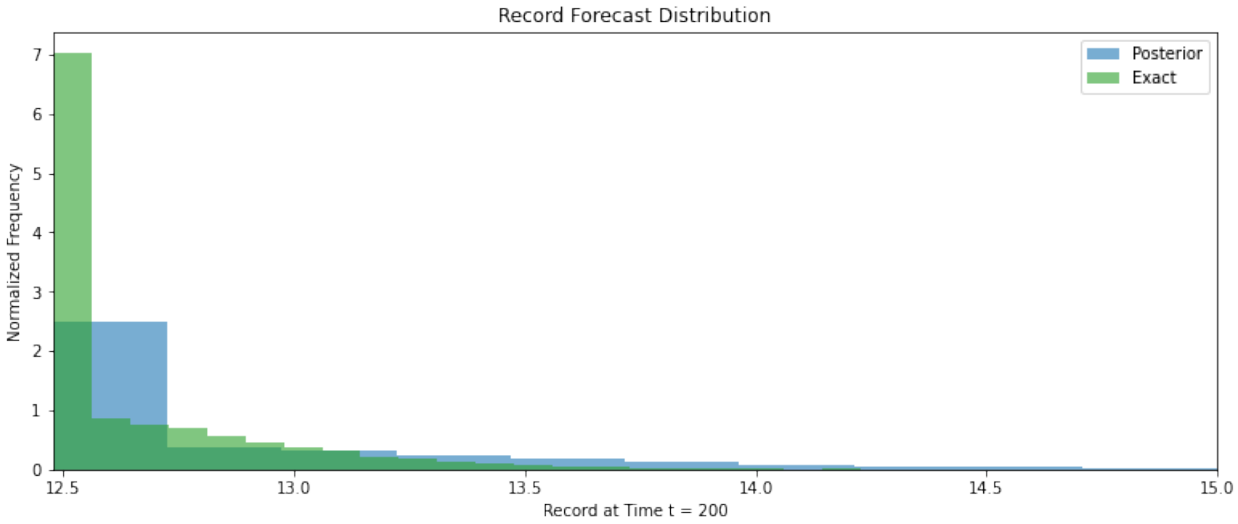

Figure 8: The forecasted distribution of the record in the 200th period. We note that the posterior has a longer tail than the exact distribution.

If we wanted to know the chance that the record breaks 13 by the period $t = 125$, we can compute this probability as a Monte Carlo expectation over our sample paths as

$$P\left(Y_{125} \geq 13\right) \approx \frac{1}{M} \sum_{j}^{M} g\left(\omega_j\right)$$

where $g\left(\cdot\right)$ is the function that returns 1 if this condition is met and 0 otherwise.

```
posterior_probability = (sample_paths[:,149] > 13).sum()/n_samps_subset
exact_probability = (exact_sample_paths[:,149] > 13).sum()/n_paths
```

We find that this probability is about 17% given our posterior, but is about 7% given the true values of μ and σ. This discrepancy arises from our uncertainty about these parameters, which is captured nicely by a Bayesian approach.

Conclusion

In this article we have:

1. Generated a **sample of a cumulative history of records.**
2. Derived the **likelihood function** for said data.
3. Defined and fitted an appropriate **Bayesian model** to the data using PyMC3.
4. Plotted the **predictive posterior distribution** of the data and **forecasted the distribution of the future record.**

We hope this tutorial will help others perform similar analyses with real data, and gain familiarity with implementing more challenging models in PyMC3 using custom distributions.

Acknowledgements

Please cite us as:

Modelling a Time Series of Records with PyMC3 by Jaime Sevilla and Jonathan Lindbloom

Jaime Sevilla is supported by the NL4XAI project and the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 860621.

Jonathan Lindbloom is supported by the Dartmouth College mathematics department.

We thank Diego Chicharro for his help on the proof in the appendix.

Appendix: Derivation of the joint likelihood

Proposition 1: Joint distribution of a historical record over time

Let $X_1, \dots, X_N$ be a collection of *i.i.d.* random continuous variables with pdf f_X and cdf F_X, and define $Y_n := \max_{i \leq n} X_i$. Then the joint likelihood for $Y_1, \dots, Y_N$ is given by:

$$\mathrm{L}_Y(o_1, \dots, o_n) = \prod_{i \in R} f_X(o_i) \prod_{i \notin R} F_X(o_i) (1)$$

where R is the set of indices where a new maximum was established, i.e.

$$R = \big\{ i : o_i > \max\{o_1, \dots, o_{i-1}\} \big\}.$$

First, we note that each Y_n is independent of $Y_1, \dots, Y_{n-2}$ when given Y_{n-1}. This means that we can decompose the joint likelihood as

$$\mathcal{L}_Y(o_1, \dots, o_N) = \mathcal{L}_{Y_1}(o_1) \mathcal{L}_{Y_2|Y_1}(o_2|o_1) \dots \mathcal{L}_{Y_N|Y_{N-1}}(o_N|o_{N-1})$$

Consequently, we only need to compute the conditional likelihood $\mathcal{L}_{Y_n|Y_{n-1}}(o_n|o_{n-1})$. We can write the CDF of $Y_n|Y_{n-1}$ as:

$$F_{Y_n|Y_{n-1}=o_{n-1}}(o_n) = P(Y_n \leq o_n | Y_{n-1} = o_{n-1}) = \; = \begin{cases} 0, & amp; \text{if } o_n < o_{n-1}, \\ F_X(o_{n-1}) = F_X(o_n), & amp; \text{if } o_n = o_{n-1}, \\ F_X(o_{n-1}) + \int_{o_{n-1}}^{o_n} f_X(z) dz = F_X(o_n), & amp; \text{if } o_n > o_{n-1}. \end{cases}$$

We will approximate the likelihood of $Y_n|Y_{n-1}$ as the limit of the difference between the cumulative conditional distribution slightly to the right and to the left of the observation (see this post on Cross Validated for discussion on this approximation).

$\mathcal{L}_{Y_n|Y_{n-1}}(o_n|o_{n-1}) \approx F_{Y_n|Y_{n-1}}(o_n + \epsilon|o_{n-1}) - F_{Y_n|Y_{n-1}}(o_n - \epsilon|o_{n-1})$ for small enough ϵ.

We consider three cases:

1. When $o_n < o_{n-1}$ we have $F_{Y_n|Y_{n-1}}(o_n + \epsilon|o_{n-1}) = 0$ and $F_{Y_n|Y_{n-1}}(o_n - \epsilon|o_{n-1}) = 0$ as long as $o_n + \epsilon < o_{n+1}$, so $F_{Y_n|Y_{n-1}}(o_n + \epsilon|o_{n-1}) - F_{Y_n|Y_{n-1}}(o_n - \epsilon|o_{n-1})$ vanishes.

2. When $o_{n-1} = o_n$ we have that $F_{Y_n|Y_{n-1}}(o_n + \epsilon|o_{n-1}) = F_X(o_{n-1}) + \int_{o_{n-1}}^{o_{n-1}+\epsilon} f_X(z)dz = F_X(o_{n-1}) + O(\epsilon)$,. For small enough ϵ the expression is approximately equal to $F_X(o_n)$.

3. When $o_{n-1} < o_n$ we have that $F_{Y_n|Y_{n-1}}(o_n + \epsilon|o_{n-1}) - F_{Y_n|Y_{n-1}}(o_n - \epsilon|o_{n-1}) = \int_{o_n-\epsilon}^{o_n+\epsilon} f_X(z)dz$.

Assuming f is C1 we can apply Taylor's theorem we get that for every $z \in [o_n - \epsilon, o_n + \epsilon]$ there exists $c \in [o_n - \epsilon, o_n + \epsilon]$ such that $f_X(z) = f_X(o_n) + f'_X(c)(z - o_n)$. And since f is C1 then we have that for every $c \in [o_n - \epsilon, o_n + \epsilon]$ there exists a bound $m > 0$ such that $|f_X(z) - f_X(o_n)| \leq |f'_X(c)(z - o_n)| \leq m|z - o_n|$.

Therefore we have that

$$\left| \int_{o_n-\epsilon}^{o_n+\epsilon} f_X(z)dz - \int_{o_n-\epsilon}^{o_n+\epsilon} f_X(o_n)dz \right| \leq \leq \int_{o_n-\epsilon}^{o_n+\epsilon} |f_X(z) - f_X(o_n)dz| \leq \leq \int_{o_n-\epsilon}^{o_n+\epsilon} m|z - o_n|dz = m\epsilon^2$$

We conclude that $\int_{o_n-\epsilon}^{o_n+\epsilon} f_X(z)dz = \int_{o_n-\epsilon}^{o_n+\epsilon} f_X(o_n)dz + O(\epsilon^2) = 2\epsilon f_X(o_n) + O(\epsilon^2)$, which is approximately equal to $2\epsilon f_X(o_n) \propto f_X(o_n)$ for small enough ϵ.

We dismiss the first case since the observed time series for the record (our data) must be non-decreasing in time. The remaining two cases leave us with

$$\mathcal{L}_{Y_n|Y_{n-1}}(o_n|o_{n-1}) = \begin{cases} F_X(o_n), & amp; \text{if } o_n = o_{n-1}, \\ f_X(o_n), & amp; \text{if } o_n > o_{n-1}. \end{cases}$$

Deriving the likelihood function for an entire time series then follows immediately. □