The Bootstrap Particle Filter

... and how to implement it in Python

Paul Wilson

October 12, 2017

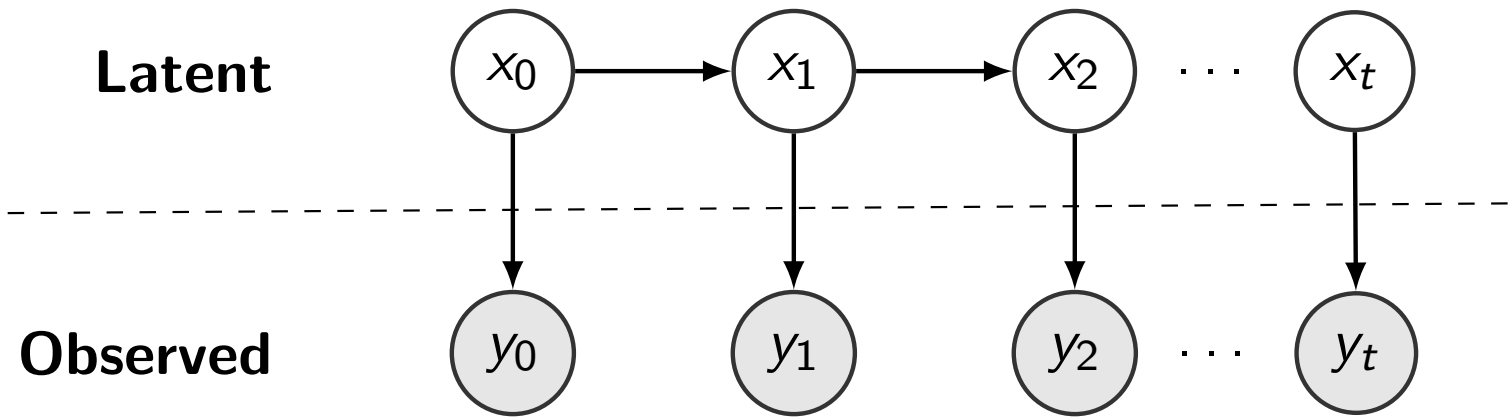
This talk

- Primarily: Understanding the bootstrap particle filter by implementing it in Python*
- Secondarily: Why the bootstrap filter is useful as a building block in other models.

* for a particular model

Sequential Monte Carlo & The Bootstrap Filter

- SMC methods are simulation-based methods for computing posterior distributions
- SMC can approximate $P(x_{0:t}|y_{0:t})$ for the model below.
- The bootstrap filter is a particular SMC algorithm with some nice properties.



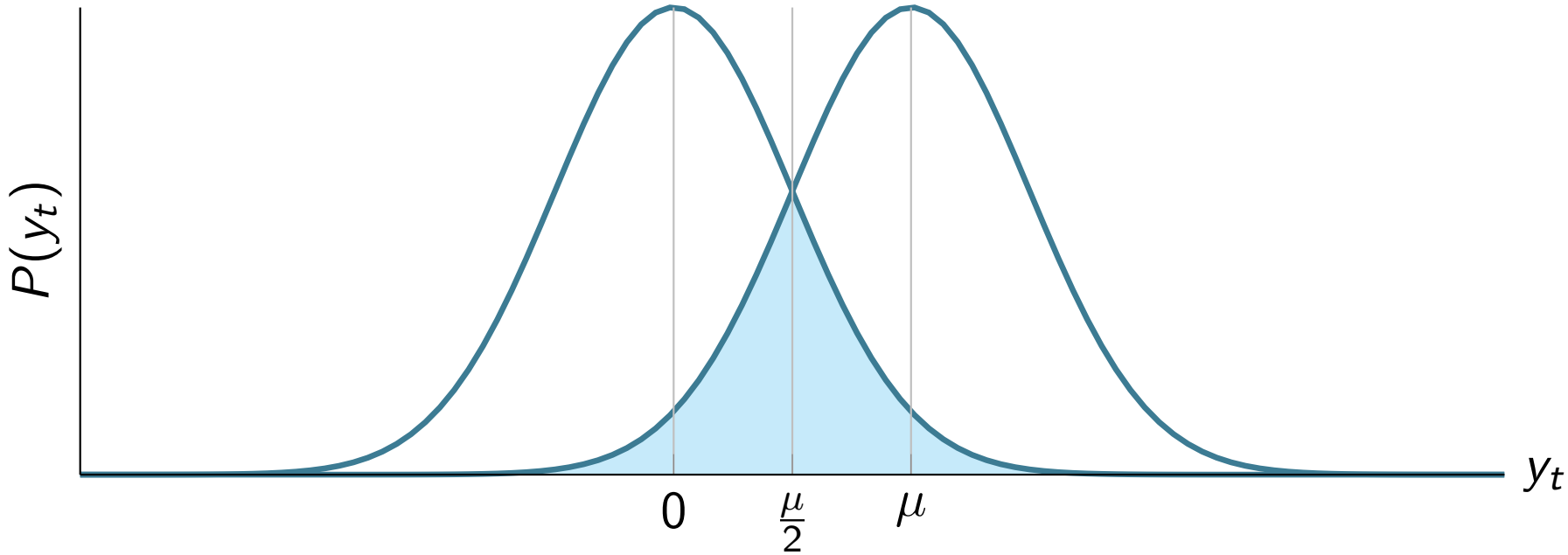
Example application: A bitcoin trading model

- We believe some traders are using 'wash trading' to cause the bitcoin price to artificially rise.
- The market is either in 'wash trade' state, or 'normal' state.
- We want to identify when the market is in 'wash trade' state so we can make a profit.
- Let's use $y_t \in \mathbb{R}$ to denote the change in price from $t-1$ to t.
- When the market is in 'wash trade' state, the average change will be some value $\mu > 0$.
- In 'normal' market conditions, y_t will remain around 0.

Example application: A bitcoin trading model

Lighter regions might represent ‘wash trading’ states

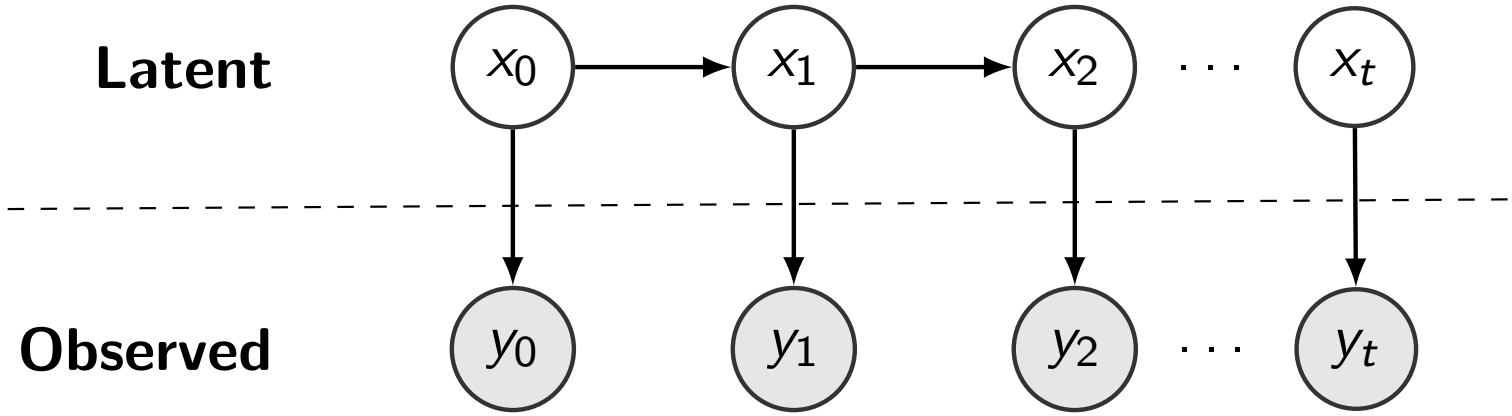
Distribution of price changes in ‘wash‘ and ‘non-wash‘ states.



A simple model is to decide for ‘wash trade‘ state when an observation $y_i > \frac{\mu}{2}$.
However, we want to incorporate time-series information to make better predictions.
We’ll make the assumption that if the market is in ‘wash trade’ or ‘non-wash-trade’ state, it’s likely to stay there with probability p.

The 'sticky-state' hidden markov model

Latent states $x_i \in \{0, 1\}$ generate observations $y_i \in \mathbb{R}$ from a normal distribution with mean μx_i and variance 1.



The transition density $p(x_i|x_{i-1})$ means the system likes to 'stick' in the current state with probability p.

$$p(x_i|x_{i-1}) = \begin{cases} p, & \text{if } x_i = x_{i-1} \\ 1 - p, & \text{otherwise.} \end{cases}$$

Finally, assume $X_0 \sim Bernoulli(\frac{1}{2})$

Generating Data with Python (part 1)

This code specifies the model dynamics. It'll come in handy later to plug in to the bootstrap filter.

```
# Draw the initial state by choosing uniformly at random from {0, 1}
def draw_x0(size):
  return r.choice(xs, size=size)


# Draw a random transition from state x_t to x_{t+1}
def draw_transition(xt, p=p):
  # if "changes" is 1, then xt will switch state.
  changes = r.choice([0, 1], p=[p, 1 - p], size=len(xt))

  # XOR "xt" with "changes" to get x_{t+1}.
  xt_next = xt.astype(int) ^ changes
  return xt_next


# Probability of observation gven state
def py_bar_xt(y, x):
  return norm.pdf(y, loc=mu1*x, scale=1)
```

Generating Data with Python (part 2)

We can use the previous code to generate sequences of states and observations from the model.

```
def simulate_forward(num_obs):
  X = np.zeros(num_obs) # list of states
  Y = np.zeros(num_obs) # list of observations

  # Draw initial state
  X[0] = draw_x0(1)

  # sample initial observation
  Y[0] = r.normal(loc=X[0] * mu1, scale=1)

  for i in range(1, num_obs):
    x = np.array([X[i - 1]])  # previous state
    X[i] = draw_transition(x) # random transition
    Y[i] = r.normal(loc=X[i] * mu1, scale=1) # generate observation

  return (X, Y) # set of states + observations.
```

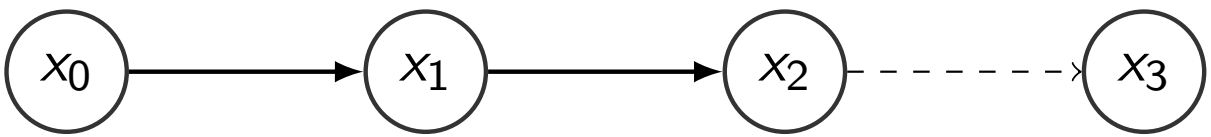
The Bootstrap Particle Filter

The idea:

- Maintain a list of N particles
- Each particle is a trajectory of states $\mathbf{x}_{0:t}^{(i)}$.
- Simulate each particle one step forward
- Weight each particle by the likelihood of the observed data, $P(y_{t+1}|x_{t+1}^{(i)})$
- Resample the list of particles according to their weights.

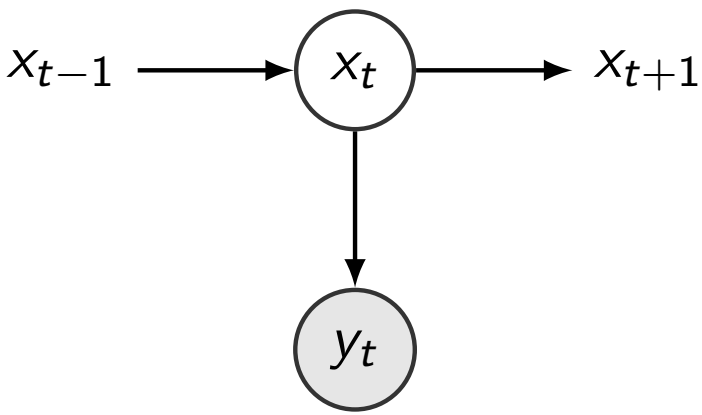
The Bootstrap Particle Filter: Simulate forward

- "Simulating forward" means generating $x_t \sim p(x_t|x_{t-1})$
- This diagram shows a single trajectory (particle), with time moving left to right.



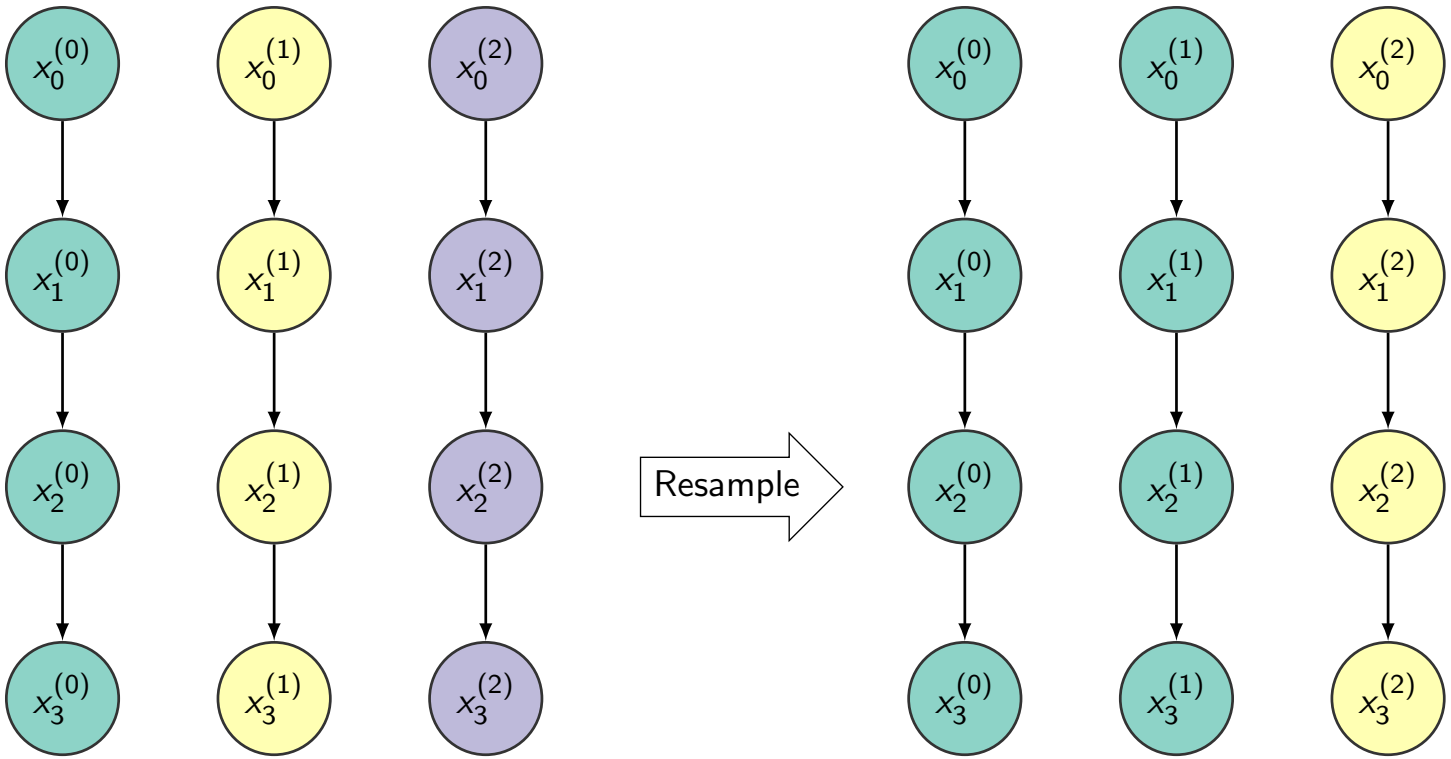
The Bootstrap Particle Filter: Weight particles

- The weight of each trajectory is the likelihood for y_t given the current state.
- Weights are normalised before the resampling step



The Bootstrap Particle Filter: Resampling

- Three trajectories are shown, with time moving downwards
- Colours denote particle ancestry
- After resampling, the first particle has been propagated twice.



$w_3^{(0)} = 0.6 \quad w_3^{(1)} = 0.3 \quad w_3^{(2)} = 0.1$

The Bootstrap Particle Filter: Pseudocode

The Bootstrap Filter

1. *Initialization*, $t = 0$.
   - For $i = 1, ..., N$, sample $\mathbf{x}_0^{(i)} \sim p(\mathbf{x}_0)$ and set $t = 1$.
2. *Importance sampling step*
   - For $i = 1, ..., N$, sample $\tilde{\mathbf{x}}_t^{(i)} \sim p(\mathbf{x}_t | \mathbf{x}_{t-1}^{(i)})$ and set $\tilde{\mathbf{x}}_{0:t}^{(i)} = (\mathbf{x}_{0:t-1}^{(i)}, \tilde{\mathbf{x}}_t^{(i)})$
   - For $i = 1, .., N$, evaluate the importance weights

$$\tilde{w}_t^{(i)} = p(\mathbf{y}_t | \tilde{\mathbf{x}}_t^{(i)})$$

   - Normalise the importance weights
3. *Selection Step*
   - Resample with replacement N particles $\left(\mathbf{x}_{0:t}^{(i)}; i = 1, ..., N\right)$ from the set $\left(\tilde{\mathbf{x}}_{0:t}^{(i)}; i = 1, ..., N\right)$ according to the importance weights.
   - Set $t \leftarrow t + 1$ and go to step 2.

An introduction to Sequential Monte Carlo Methods, (Doucet, de Freitas, Gordon, p 11)

The Bootstrap Particle Filter: Python implementation

```
def bootstrap(ys, num_particles):
  # 1. Initialization
  T       = len(ys)
  # A grid T "tall", and num_particles "wide".
  X       = np.zeros(shape=(T, num_particles))
  # draw from x0 N times.
  X[0]    = draw_x0(num_particles)

  indexes = np.array(range(0, num_particles))

  # weight each particle based on probability of observing y_0
  # then normalise + select
  wt_tmp = normalise( py_bar_xt( ys[0], X[0] ) )
  ixs    = r.choice(indexes, p=wt_tmp, size=num_particles)
  X      =  X[:, ixs]

  for t in range(1, T):
    # 2. Importance sampling step
    X[t]   = draw_transition(X[t - 1])
    wt_tmp = normalise( py_bar_xt( ys[t], X[t] ) )

    # 3. Selection step: pick particles weighted by importance.
    ixs = r.choice(indexes, p=wt_tmp, size=num_particles)
    X = X[:, ixs]

  return X
```

Discussion

- The bootstrap filter doesn't need to 'hard-code' model dynamics
- in the previous Python code you could easily factor out model-specific code.
- PMCMC methods exploit this, so inference can be made for arbitrary models with *"limited design effort on the users' part."*
- PMCMC can be useld, for example, to infer distributions over parameters μ and p in our example model.

Experiments!

Three models are compared on simulated data of 40 observations.

- The Bootstrap particle filter implemented above
- The Viterbi algorithm
- The 'Baseline' model - deciding for state $x_i = 1$if $y_i > \frac{\mu}{2}$.

I measured accuracy as the percentage of states x_i correctly guessed, averaged over $10,000$ runs for the parameters $p = 0.95$, $\mu = 1$, and $N = 1000$ particles.

algorithm	accuracy
Baseline	69%
Viterbi	86%
Bootstrap	87%

Experiments Discussion

- Minor difference between Viterbi and Bootstrap might be due to metric chosen.*
- Viterbi returns the MAP estimate of the whole sequence, not each state individually.
- Baseline result is equal to $f(x = 0.5|\mu = 1, \sigma^2 = 1)$ where f denotes the normal CDF.

* or a bug