1 Hidden Markov Models

A **Hidden Markov Model** (HMM) is defined by an underlying system that is a Markov process with states that cannot be observed, but gives outputs that are dependent on the hidden state of the system. If we let random variable X_i represent the i-th state of the system, and Y_i represent the i-th output of the system, we have:

$$X_1 \sim \mu(x_1) \text{ and } X_n|(X_{n-1} = x_{n-1}) \sim f(x_n|x_{n-1}) \tag{1}$$

$$Y_n|(X_n = x_n) \sim g(y_n|x_n) \tag{2}$$

where X_1 is defined by some initial conditions $\mu(x_1)$. Note how f obeys the Markov property, so the current state is dependent only on the previous state. Also note that g shows that the current observation is only dependent on the current state.

2 Particle Filters

The main goal is to use our observations to determine the hidden state of the system. That is to say, we want to figure out $p(x_n|y_{1:n})$. Recognizing that this is a marginal is useful because then we can just say:

$$p(x_n|y_{1:n}) = \int p(x_{1:n}|y_{1:n})dx_{1:n-1} \tag{3}$$

Setting this as a marginal of $p(x_{1:n}|y_{1:n})$ is helpful because we can refactor this conditional distribution to see how it comes from our prior distributions of the hidden state and observations. With some refactoring and using our initial HMM conditions we can show:

$$p(x_n|y_{1:n}) = \int \frac{\mu(x_1)\prod_{i=2}^{n} f(x_1|x_{i-1})\prod_{i=1}^{n} g(y_i|x_i)}{\int \mu(x_1)\prod_{i=2}^{n} f(x_1|x_{i-1})\prod_{i=1}^{n} g(y_i|x_i)dx_{1:n}} dx_{1:n-1} \tag{4}$$

This looks really messy, but it's helpful to see that this is just integration of functions we already know: f and g. In other words, we already have all the information we need to arrive at the answer.

The issue now is that this is sometimes intractable. If functions are nice, like guassians or linear distributions, this is integrable and we can arrive at a solution analytically. However, this is not always the case, which is why we sometimes need to derive an answer from numeric approximations.

3 Kalman Filter

So let's say you're really lucky and your initial distribution is something nice. This allows us to get a solution analytically through a method called the Kalman Filter. The algorithm becomes a little more obvious if we recognize the recursive form of our goal:

$$\text{Prediction step: } p(x_n|y_{1:n-1}) = \int f(x_n|x_{n-1})p(x_{n-1}|y_{1:n-1})dx_{n-1} \tag{5}$$

$$\text{Update step: } p(x_n|y_{1:n}) = \frac{g(y_n|x_n)p(x_n|y_{1:n-1})}{p(y_n|y_{1:n-1})} \tag{6}$$

Essentially, we are now assuming that $p(x_{n-1}|y_{1:n-1})$ is known due to recursion. To see how this relates to our earlier definition of $p(x_n|y_{1:n})$ in Eq. (4), note that the numerator of Eq. (4) is the product of all of f and g from Eq. (5) and (6) while the denominator is the product of conditionals from Eq. (6). The denominator simplifies to $p(y_{1:n})$ which is just the marginal distribution of observations on the joint, hence the second integral inside of the denominator of Eq. (4).

4 Sequential Monte Carlo Methods

So what if our distribution isn't nice? We will use Monte Carlo Methods to approximate Eq. (5) and (6):

$$\text{Prediction step: } \pi_{x_{1:t}|y_{1:t-1}}(x_{1:t}|y_{1:t-1}) = f(x_t|x_{t-1})\pi_{x_{1:t-1}|y_{1:t-1}}(x_{1:t-1}|y_{1:t-1}) \tag{7}$$

$$\text{Update step: } \pi_{x_{1:t}|y_{1:t}}(x_{1:t}|y_{1:t}) \propto g(y_t|x_t)\pi_{x_{1:t}|y_{1:t-1}}(x_{1:t}|y_{1:t-1}) \tag{8}$$

5 SIS

To algorithmically find our target distribution, we sample from something simple like the Uniform or Gaussian distribution. We then weight these samples using our observations and g. Essentially, we iterate between (7) and (8), sampling new points and updating our weights each time. While this sounds okay, it turns out that we never use SIS in real life because of something called the "degeneracy problem".

6 Degeneracy Problem

When we are sampling from our reweighted distribution, it is possible that certain points are assigned very low weight. In future iterations, this is likely to compound, making their weights essentially negligable. This manifests as a handful of points having nearly all the weight while the rest of our points have "degenerated". This is a problem because our distribution essentially is now defined by just a few of the original points we started with, which is not enough to get an accurate representation of our target distribution.

7 Bootstrap (SIR)

```
1. Initialization:
   For $i = 1, \dots N$:
       Sample $x_0^{(i)} \sim \mu(x_0)$
       Assign weights $\widetilde{w}_0^{(i)} = g(y_0|x_0^{(i)})$
       Normalize weights $w_0^{(i)} = \frac{\widetilde{w}_0^{(i)}}{\sum_{i=1}^n \widetilde{w}_0^{(i)}}$
2. Importance Sampling:
   For $t = 1, \dots, T$:
       Sample $x_t^{(i)} \sim f(x_t|\widetilde{x}_{t-1}^i)$
       Assign weights $\widetilde{w}_t^{(i)} = g(y_t|x_t^{(i)})$
       Normalize weights $w_t^{(i)} = \frac{\widetilde{w}_t^{(i)}}{\sum_{i=1}^n \widetilde{w}_t^{(i)}}$
       Resample $\widetilde{x}_t^{(i)}$ from $x_t^{(i)}$ according to the weight distribution with replacement.
3. Return $\{x_T^{(i)}, w_T^{(i)}\}_{i=1}^N$
```

While this addresses the degeneracy problem, it turns out that resampling has it's own issue which we call the "sampling impoverishment problem". This comes about because our higher weighted points are more likely to be drawn multiple times, and it's possible that we might get a really bad resampling where all of the new points get drawn from a single point. There are methods that address both of these issues but they require fancy sounding things like the "Epanechnikov Kernel", which our outside the scope of our discussion. It turns out that SIR is generally good enough and is the algorithm that people generally use.