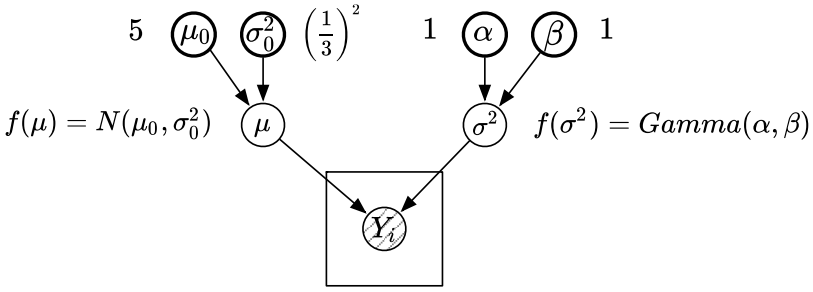
Nathan Hadfield, Daniel Gardner, David Grimsman
CS 677 (Spring 2014)

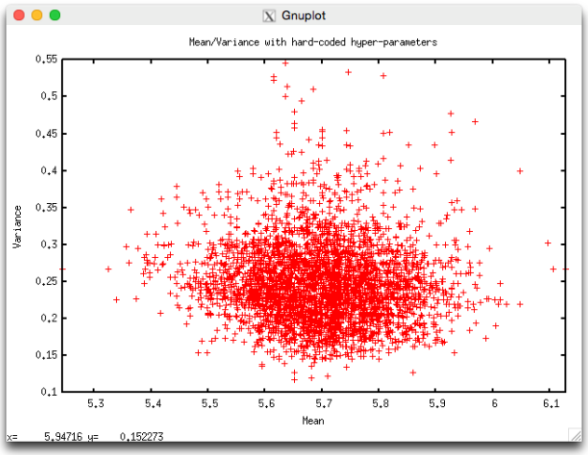
MCMC Lab 3: Parameter Learning

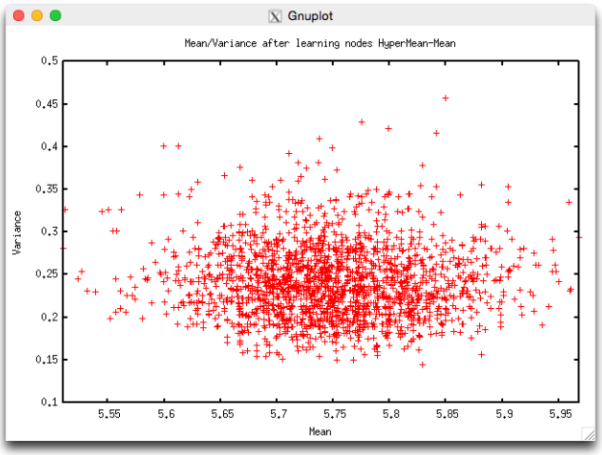
Faculty Data

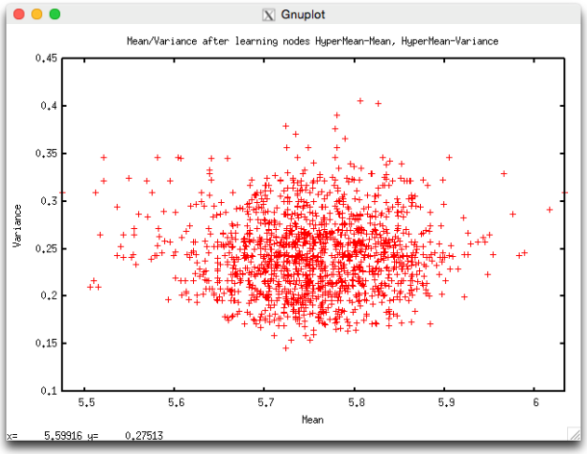
We started the lab by first looking at the faculty data from MCMC Part 2 lab. We added 4 hyper-parameters as shown in the figure below.

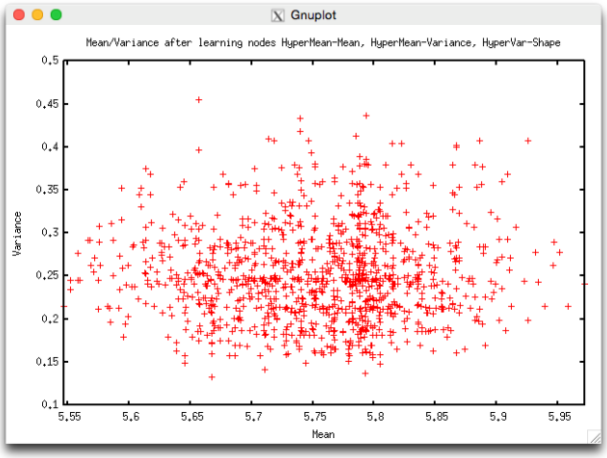


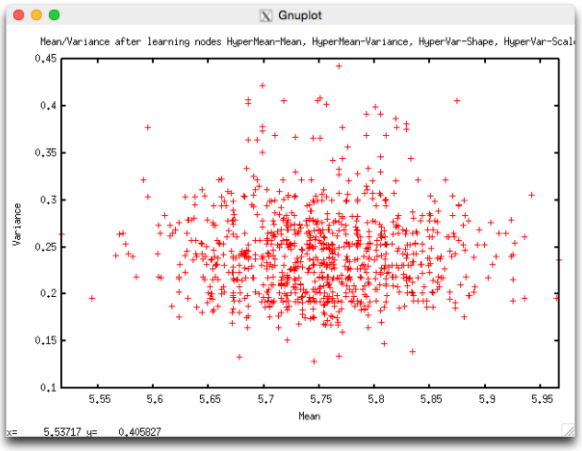
We set out to explore how the hyper-parameters' learning affects the posterior for mean and variance (μ and σ^2).





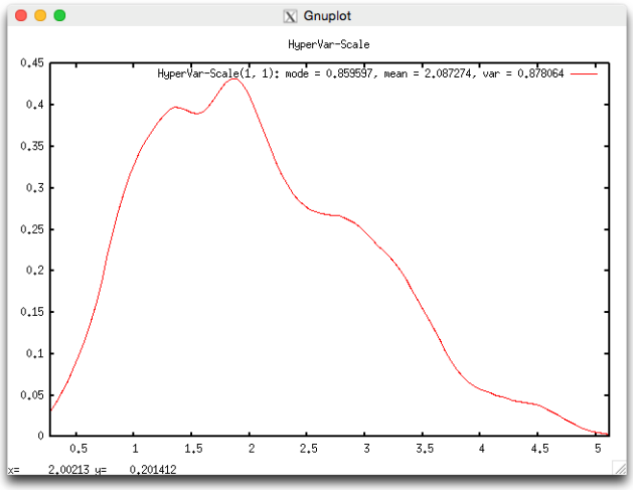




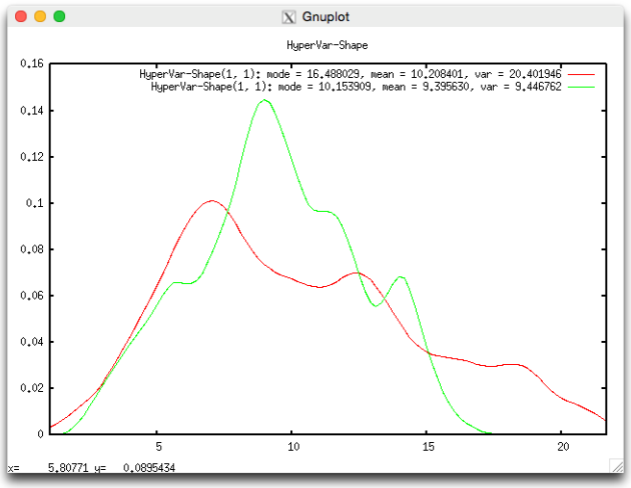


As can be seen from the plots above, when we hard-code all of the hyper-parameters, the values of μ and σ^2 tend to clump. As we progressively moved all 4 nodes associated with hyper-parameters to be learning nodes, we can see the clump begins to disseminate. This is because more of the network is unknown, and while we can still infer what the true values of μ and σ^2 are, we can infer those values with less certainty. The tradeoff, though, is that we don't need to know as much information a priori.

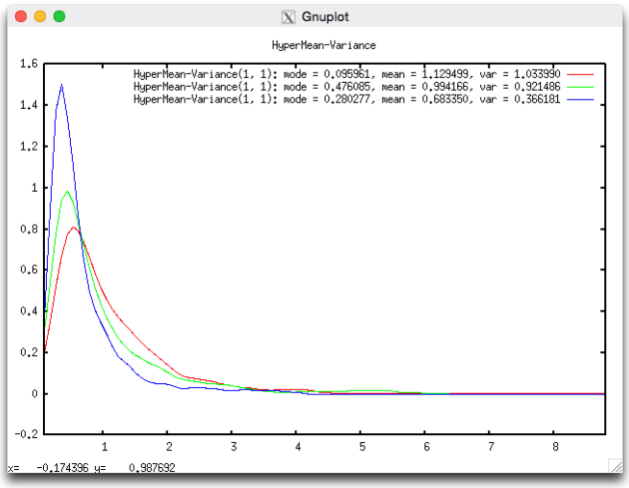
As we varied how many hyper-parameters were hard-coded, we also noticed how that affected the value distributions for the nodes in the network.



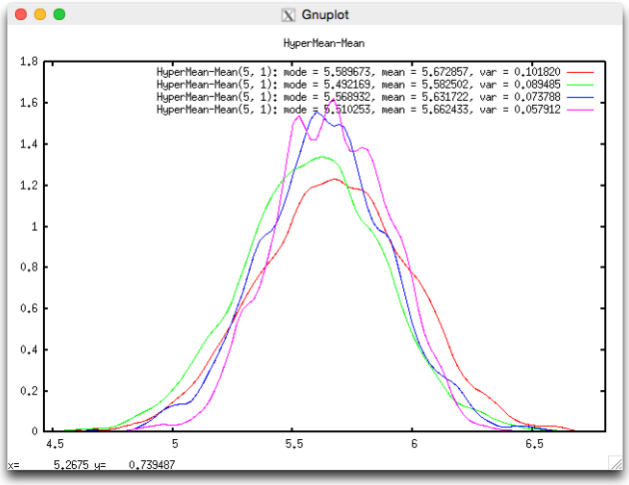
We started with the value for the HyperVar-Scale node. This is the distribution as we let the node learn.



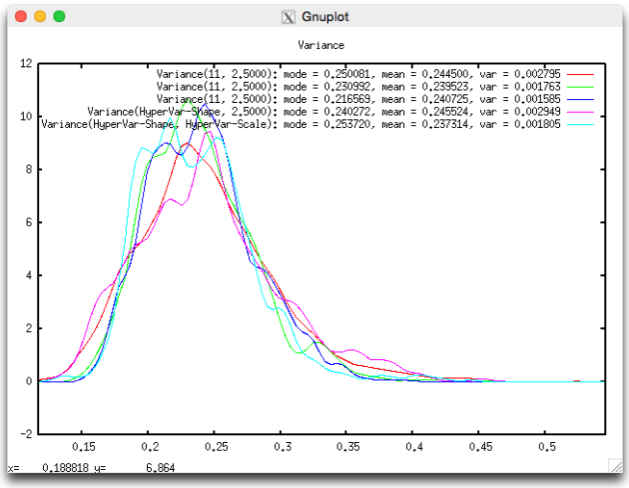
This, then is the value for the HyperVar-Shape node. The red line represents the distribution when one of these 2 nodes are learners and the green represents when they both are.



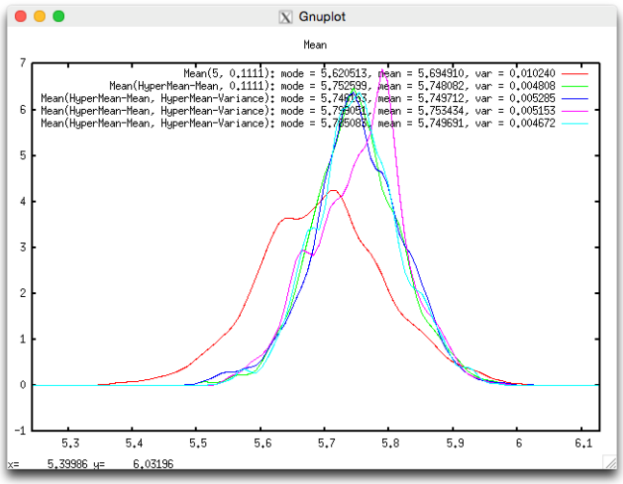
The same goes here: as we move from red to green to blue, we simply make one more node a learner.



This is the last of the hyper-parameter nodes, varied similarly as above



This is how the Variance node distribution behaves as more nodes become learners. As we go from orange to light blue, we can see that the variance in the distribution shrinks.



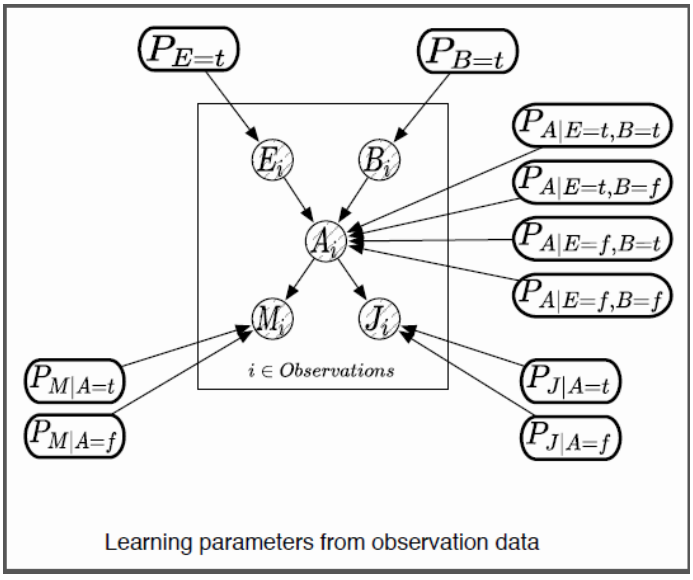
The same observation goes for the Mean node: as we increase the number of learner nodes, we can predict the value of Mean with more confidence.

We see this phenomenon where more learners help us predict values more accurately because it means we are relying on our observations more than we are relying on our predetermined values. This is helpful when the values for hyper-parameters are unknown, which is generally the case.

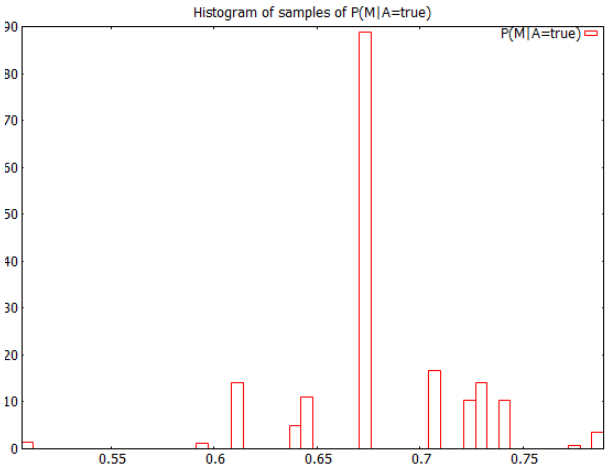
Alarm Model

At this point we turned our attention to the alarm model from MCMC Part 2 lab. We adjusted the parameters as required in the lab specs, essentially making sure that no probability was less than 0.05. As will be explained, this allowed our code to converge faster.

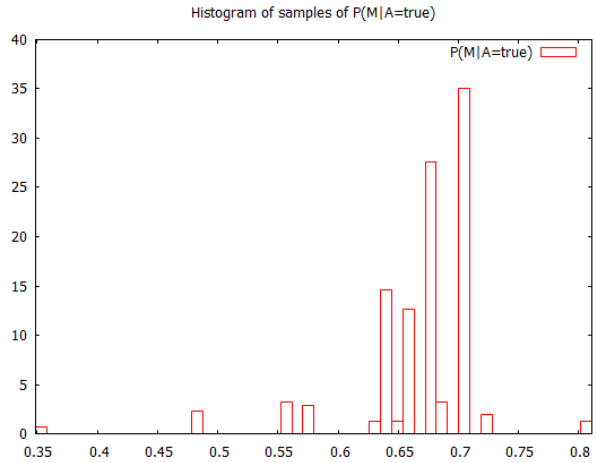
In order to get some observation data, fixed all the values for all the nodes and then sampled the entire network. To start we gathered 1000 samples. Once we had our observations, we experimented with our network to see if, given the observations, we could figure out what the network parameters are. This meant that we needed to augment the Bayesian network by adding a node for each table value (our hyper-parameter nodes) and a node for each observation. We hooked it all together similar to the golfer example in MCMC Part 2.



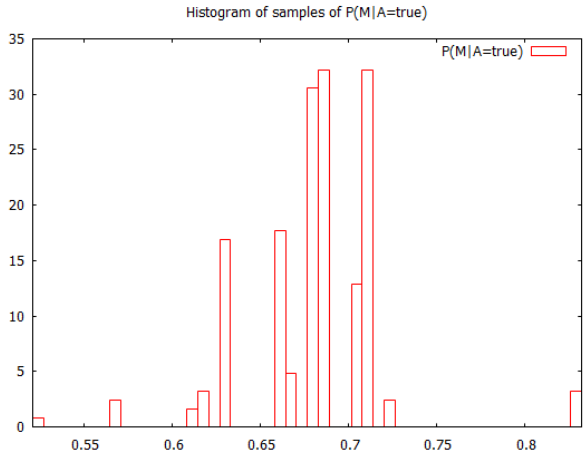
To get a feel for how adding hyper-parameter nodes affected the network, we'll look at the histogram for the values at a single node, *P(M=true | A=true)*.



Histogram when we have 4 learned nodes



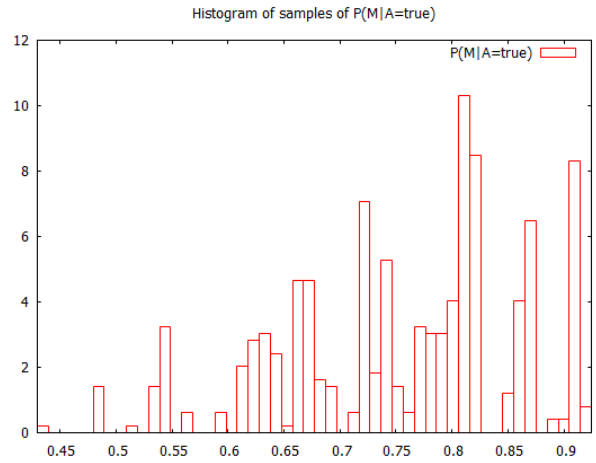
Histogram when we have 6 learned nodes



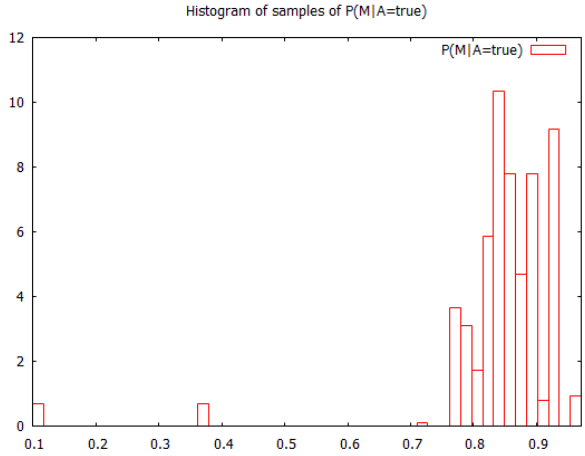
Histogram when we have 10 learned nodes

Recall that the correct value for this node is 0.7. Remarkably, in each case, we get a pretty good estimator for this value. However, notice that we are seeing the opposite affect from the faculty network: as we increase the number of learning hyper-parameter nodes, the variance increases. This happens because we now know the correct values to place in the parameters. In the faculty network, we were guessing, so the learners did a better job finding the true value. In the alarm network, we know what the correct hyper-parameter values should be, so we are more confident without the noise from the learners.

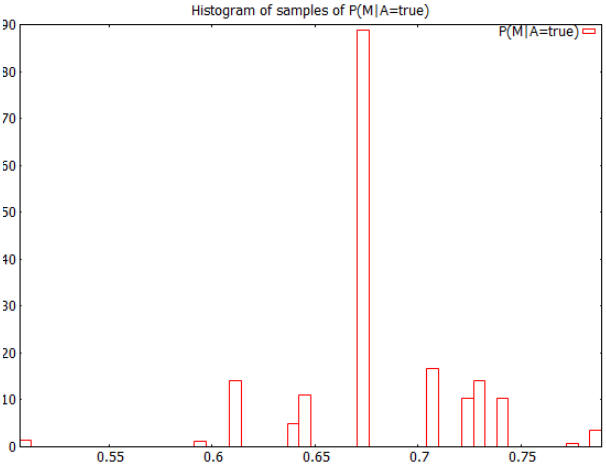
Next we tested how the number of observations affects the network. We started with the same node, with a total of 4 learned nodes, but this time we varied how much data we observed beforehand.



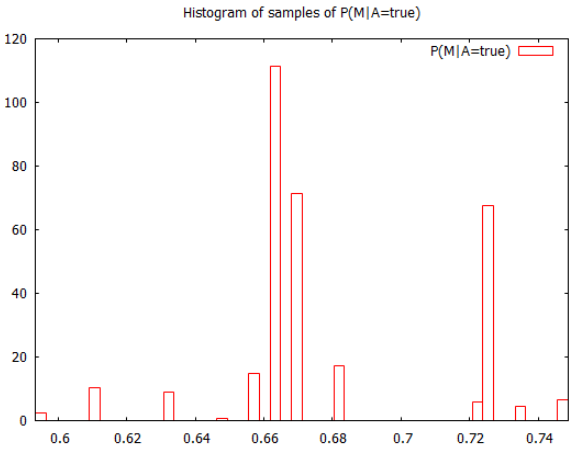
Histogram with 50 observations



Histogram with 100 observations



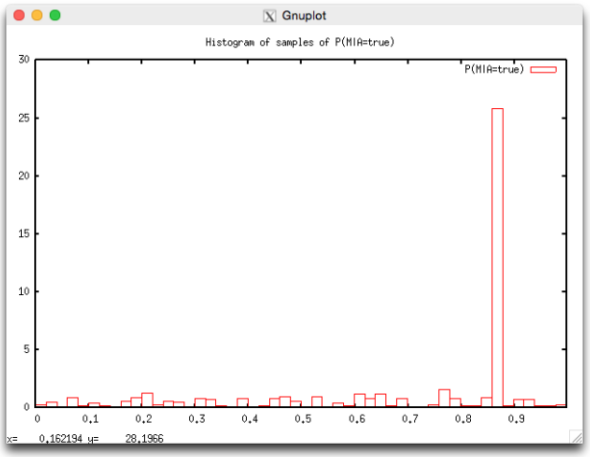
Histogram with 1000 observations



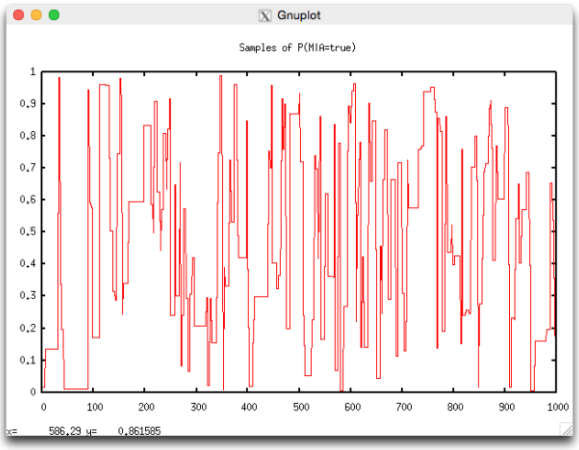
Histogram with 5000 observations

As we expect, with more observations, we become increasing accurate and decreasingly variant (note the scale on each *x*-axis). However, since each observation also becomes a node in the network, more observations means a longer runtime. So, as with most things we have a tradeoff: accuracy costs time and computation.

We can also look at what else might slow our sampling down. For example, what if we went back to using our original parameters from MCMC Part 1? We went back, changed the parameters, re-observed (1000 observations), and ran our sampler (1000 samples). The correct values were more difficult to obtain:



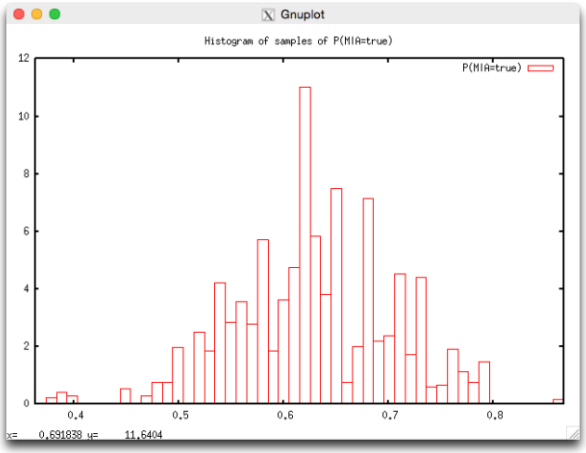
Histogram with original parameters



Mix plot with original parameters

Notice that the histogram is not as accurate, suggesting that the correct value for *P(M=true | A=true)* is about 0.85 instead of 0.7. The mix plot also indicates a high variance. This is because the node we are watching assumes that *A=true*. However, *A* is generally only true when either *B* or *E* is true. *B* and *E* are only true with a small probability, so to get an accurate guess at *P(M=true | A=true)* we would need more observations, which would then take more time to sample. In general, we can say that small parameters make the whole network difficult to learn.

Another natural question: why wouldn't we just add hyper-hyper-parameters? We can certainly do this, but with marginal improvement (if any) given that it takes longer to compute:



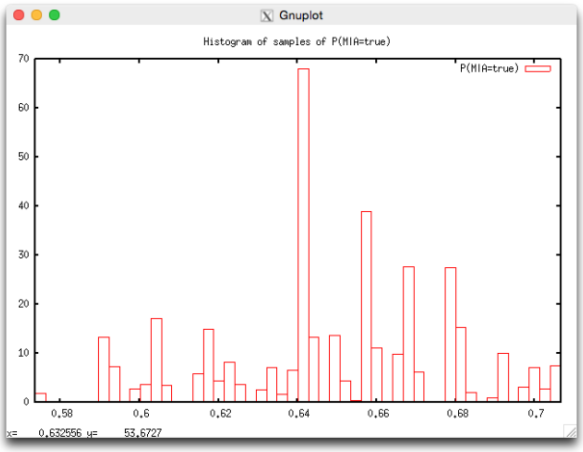
In this case, we can see it just added more noise to our plot. Adding more learner nodes requires more sampling to get the parameters correct. We could probably see improvement if we had more sampling. However, as it stands, this made our estimation worse.

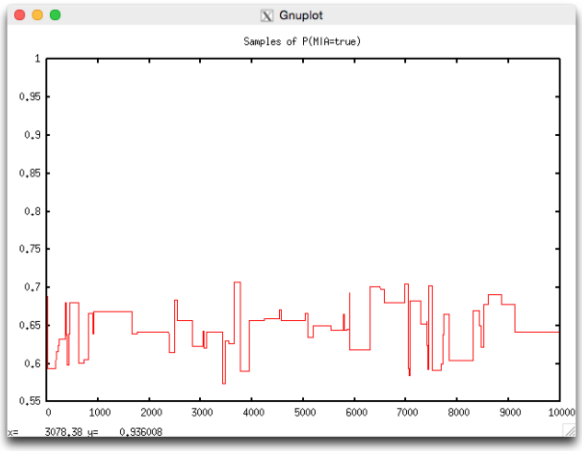
Putting It All Together

Continuing to work with the alarm network, we ran some additional tests. First, we wanted to see what would happen if we had imperfect data, a common problem in the real world. We reran our observations, but this time some of the nodes were unobserved. The percentages indicate what % of observations had a missing data value. We ran these tests with the following settings:

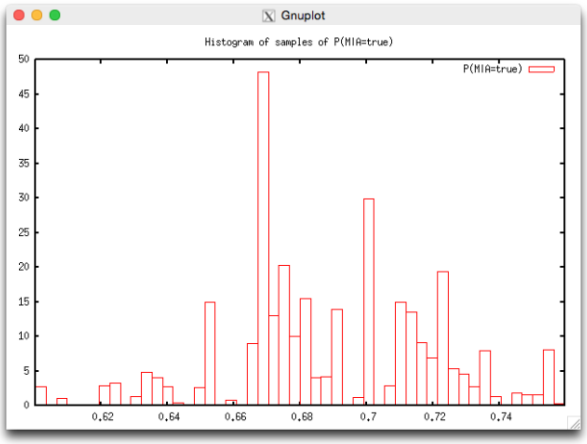
- 1,000 observations
- 1,000 burn
- 10,000 samples

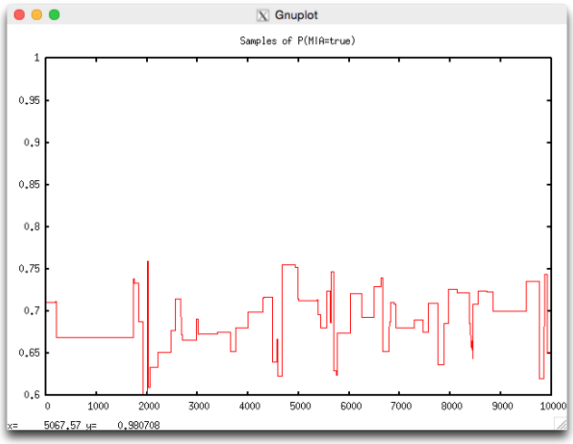
Missing 1%



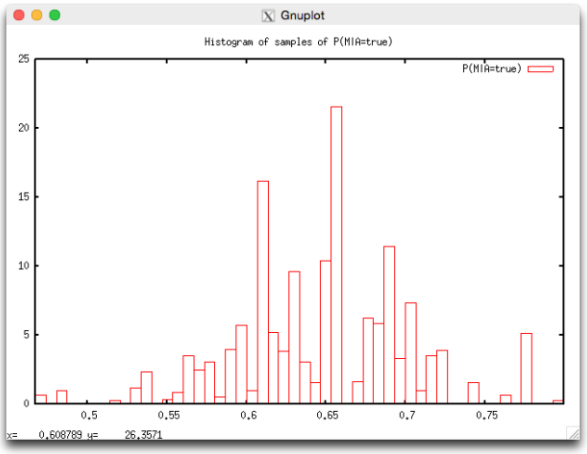


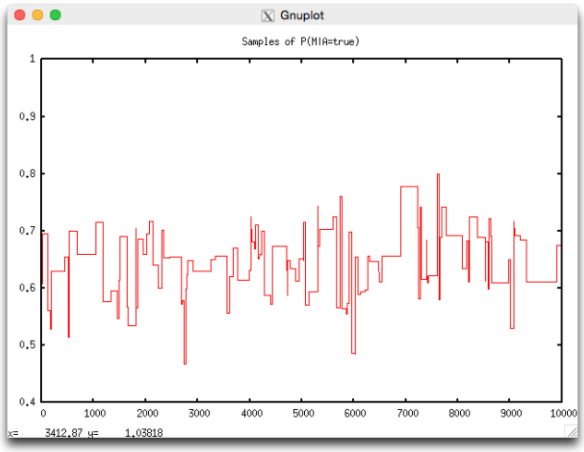
Missing 10%



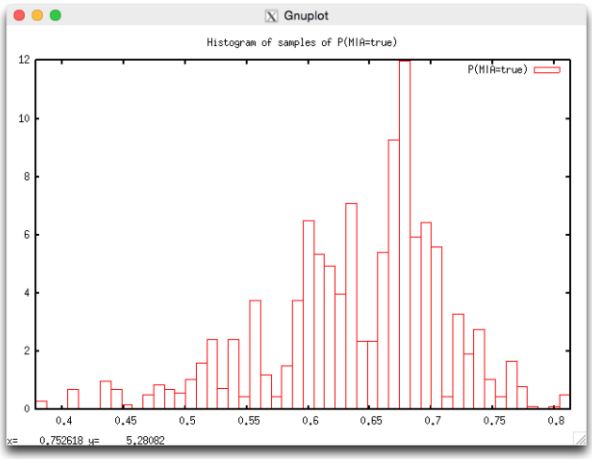


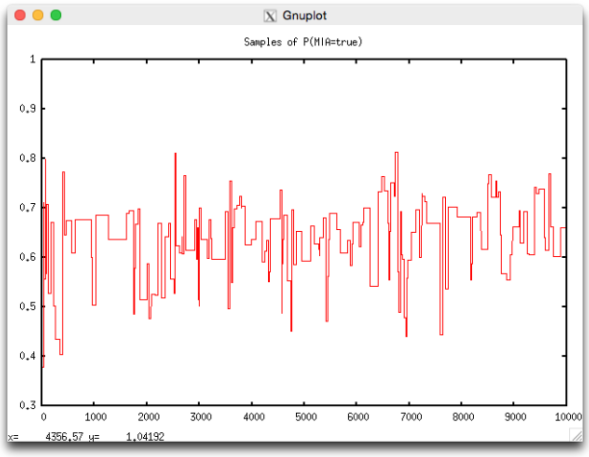
Missing 33%





Missing 50%



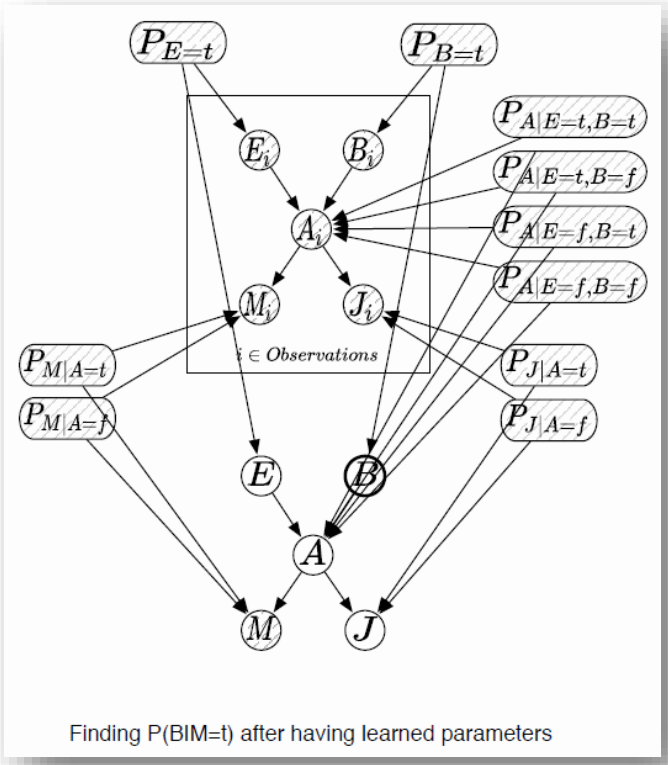


Somewhat surprisingly, our network does a good job of estimating our parameter here despite missing data. The nodes are able to continue learning even when the data is imperfect. However, as we expected, the more data that we miss, the less accurate our learner gets with more variance (again, note the axis scales on the plots).

Lastly, we wanted to see how our new learned network parameters would fair with a query. So, we did the following:

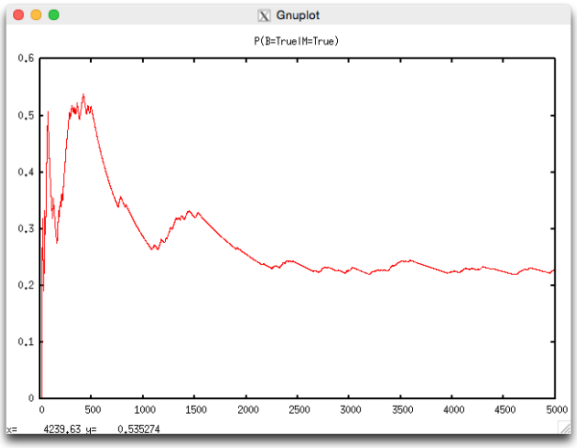
- Gathered 1,000 observations
- All 10 hyper-parameters as learners
- Ran 10,000 samples with 1,000 burn. This set our network with our learned parameters
- Additionally sampled (5,000 samples, 1,000 burn) with the query *P(B=true | M=true)*.

Our new network looked like the following:



Finding P(BIM=t) after having learned parameters

As we sampled, we got the following graph for the value of *B*:



Notice that it starts to settle at just about 0.22. This is close to the correct value – about 0.2 as confirmed by running the query on our network from MCMC Part 1. Therefore, we were able to use a noisier network to obtain a similar result.