Felix Sun
6.867 Final Project

Learning to be Impatient: Modeling the Dynamics of the Inference Engine in Venture

We model the log score in an inference run of Venture (a probabilistic programming language), using a hidden Markov model (HMM). To find parameters for this model, we apply a slight variant of the EM algorithm, along with message passing on the HMM. We use this model to help the Venture inference engine escape local optima and find global optima faster. We find that our model indeed helps Venture find better optima, but also that randomly resetting the sampling algorithm had much the same effect.

1 – Introduction

Venture [1] is a probabilistic programming language, which allows users to describe probability models using the syntax of a programming language. The user can then describe observations of his model, and run inference to get estimates of the non-observed nodes. A simple Venture model may be as follows:

```
(assume rained_today (flip 0.3))  ; There is a 0.3 chance of rain
(assume sprinklers_on (if rained_today
  (flip 0.1)
  (flip 0.8)
)
(assume wet_grass (if (or rained_today sprinklers_on)
  (flip 0.9)
  (flip 0.1)
)
(observe wet_grass true)
(observe rained_today true)
(infer 100)
(predict sprinklers_on)
```

Figure 1: A probabilistic model, as specified in Venture. The predict command should return a distribution over the possible states of sprinklers_on, e.g. the probability that the sprinklers were on.

To run inference on a Venture model, Venture applies the Metropolis-Hastings (MH) algorithm, a sampling algorithm for probability distributions. Roughly speaking, MH starts with some guess of the states of the system. It then makes a small change to the state (flipping a boolean variable, or perturbing the value of a continuous variable), and evaluates the probability of the model generating this state. If this probability increased with the change, MH keeps this change. Otherwise, it may revert to the old guess, or it may still keep the change with some small probability that is a function of how much the likelihood decreased. The "output" of MH is the string of guesses that it makes. In the infinite limit, this string of guesses is guaranteed to match the actual distribution of the un-observed variables, given the observed variables.

The main difficulty with MH is that the convergence time may be very large. It may take MH a long time to find a "good" guess – one that has a high likelihood – and the string of guesses will only be close to the true distribution over the un-observed variables once some good guesses has been found. Before then, MH returns very low-likelihood states.

There is no accepted algorithmic solution to this problem, but there are some successful "hacks". Often, running many separate instances of MH on the same problem is more likely to lead to a global optimum then running one instance for a longer period of time. Periodically resetting the sampler has the same effect. If the user is

[1] http://arxiv.org/pdf/1404.0099v1.pdf

watching the execution of the sampler, he might want to manually reset the sampler whenever the likelihood score appears to plateau. In this project, we implement an automatic version of this last strategy using machine learning – we train a model of how the likelihood score changes over time, and reset the sampler based on the results of the model. This will hopefully allow MH to find a more likely optimum in fewer iterations.

2 – Modeling the time-evolution of the likelihood score

Our "control strategy" will only use one variable: the value of the MH algorithm likelihood score. The goal is to determine when the MH algorithm is stuck in a local maximum from the time-evolution of the likelihood. As such, we model the likelihood score as a hidden Markov model, with each step in the algorithm corresponding to one transition of the HMM. Each hidden state represents a "true value" of the likelihood. The observed value is the true value, plus some Gaussian noise. There are nBins possible true values. At each step, the likelihood has a (1-pTrans) chance of staying at the same true value, or a pTrans chance of switching to a different one. If the likelihood jumps, any other true value is considered equally likely. This model is summarized in Figure 2.

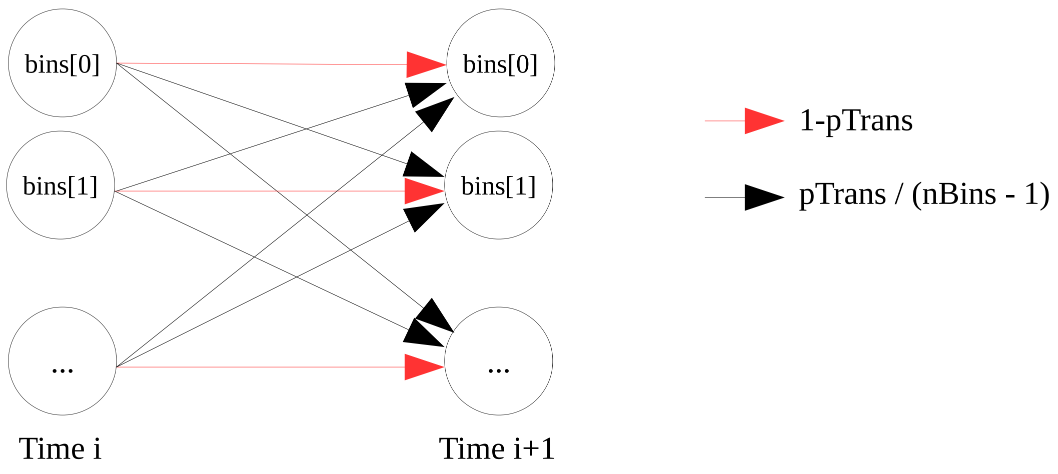


Figure 2: Markov chain for the time-evolution of the likelihood value. There are nBins possible values for the likelihood, and the value changes with total probability pTrans.

Note that these "true values" are not designed to correspond to any physical quantity – they are merely a way to simplify the dynamics of the inference algorithm.

We expect every probabilistic program to have a different set of parameters for this HMM: a different set of bins (true likelihood values), and a different pTrans. Therefore, we need to learn these parameters for every program. To do this, we first run several short inference chains, and fill a matrix likelihood[step, chainIdx] with the likelihood values from each chain. (In my final experiments, I used 10 chains of 100 steps each. I did not try any other possibilities.)

We then use the EM algorithm to find a set of bins and a pTrans that explain the observed likelihoods well. In my experiments, we fixed nBins to 5 bins to reduce the complexity of the inference problem. (I did not try any other values of nBins.) To apply the EM algorithm to this scenario, we start with an estimate of the bin values and of pTrans. We then assign each step in each chain to a bin, based on the estimated parameters. We then re-estimate the parameters based on the bin assignments from the previous step. This process is repeated until the parameters stop changing from iteration to iteration. The labeling and parameter re-estimation steps are described below..

Labeling the steps with a bin value is done using the message passing algorithm. Each node in the Markov chain (each step in the sampler) passes to its neighbors (the steps immediately before and after it in the same chain) a message. The message from node i to node i+1 is the probability that node i+1 is in each bin, given the information from nodes 1 through i:

$$Msg_{i\to i+1} = \begin{bmatrix} P(i+1 \in bin_1|i) \\ P(i+1 \in bin_2|i) \\ \dots \end{bmatrix} = (A \cdot Msg_{i-1\to i}) * \begin{bmatrix} P(step_{i+1}|i+1 \in bin_1) \\ P(step_{i+1}|i+1 \in bin_2) \\ \dots \end{bmatrix}$$

Here, A is the transition matrix:

$$A[y,x] = P(i+1 \in bin_y|i \in bin_x)$$

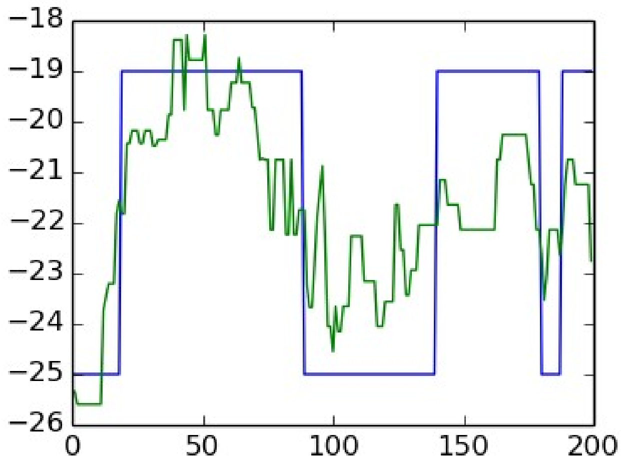
And the observation probability is a normal distribution around the true value of the bin.

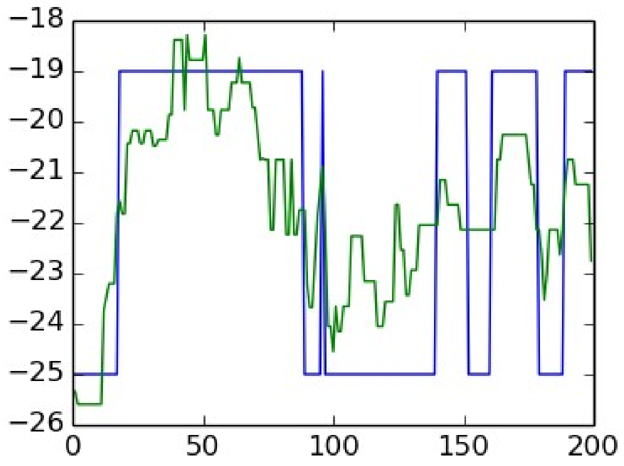
$$P(step_i|i \in bin_b) = N(\text{value}(bin_b) - \text{value}(step_i), \sigma)$$

Finally, each message vector is normalized to sum to 1. To save space, this is not shown in the equations above.

The messages from i+1 to i are defined analogously, and the final bin distribution at each node is equal to the element-wise product of the two messages into the node, normalized to 1.

To compute the bin distribution, we need to compute all messages, starting with Msg[0 to 1] (assuming a uniform prior distribution on bins at node 0) and going forwards to the end of the chain. At the end, we have a best estimate of the bin distribution of the last node. Using this estimate, we compute the backwards messages, until we have every message, and therefore a bin distribution at every node. In Figure 3, this message passing algorithm is used to classify the likelihood score from a run of MH.





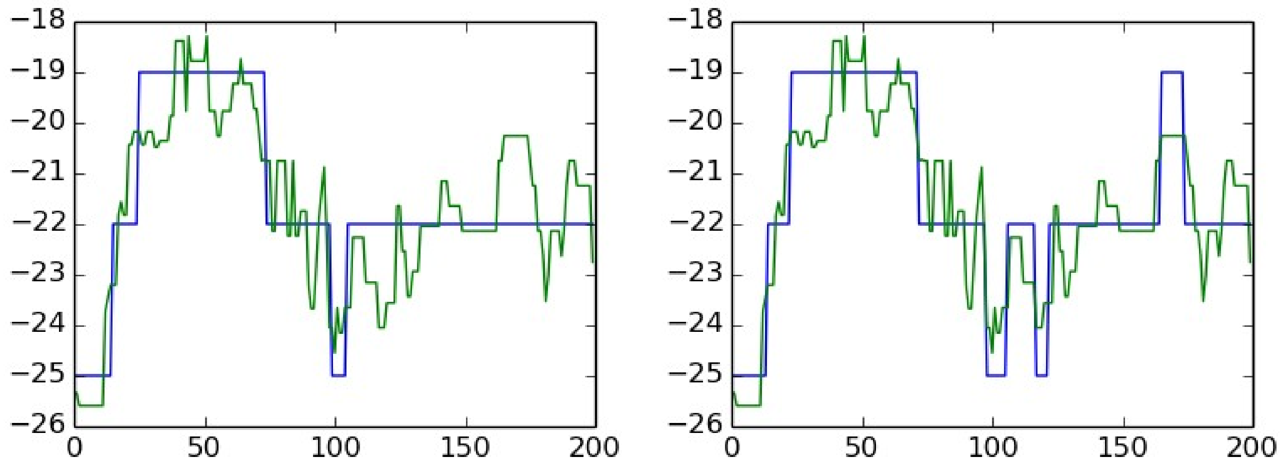


Figure 3: Using the message passing algorithm to classify a series of likelihood scores. Green traces are likelihood scores, blue traces are the HMM's most likely prediction of the "true value" of the likelihood score at each point. Different parameters were used in each plot, showing how these parameters affect the output. Top left: two bins, pTrans = 0.01. Top right: two bins, pTrans = 0.05. Bottom left: three bins, pTrans = 0.05. Bottom right: three bins, pTrans = 0.2. Values for the bins were manually selected.

Given the probability that each step is in each bin, it is easy to re-estimate the bin true values. Simply take a weighted average of the likelihood value of each step, weighted by the probability that the step is in the bin of interest. We also need to estimate the new pTrans, the probability that the value jumps to a new bin. It's not obvious how to estimate pTrans directly from the probabilities, so we first found the maximum likelihood bin for each step. We counted the number of places where the most likely bin changed in a step, and set pTrans to this number divided by the total number of steps. We implemented this process in Python using Numpy, to interface with the Venture front-end.

Using the EM algorithm, we can cluster and label some sample data from a run of MH on a Venture program. Here, we performed EM on a sample of 10 different runs of MH on the same probabilistic program. (An explanation of how test programs were generated is found in the next section.) Each run was 100 steps long, and as always, there are 5 bins.

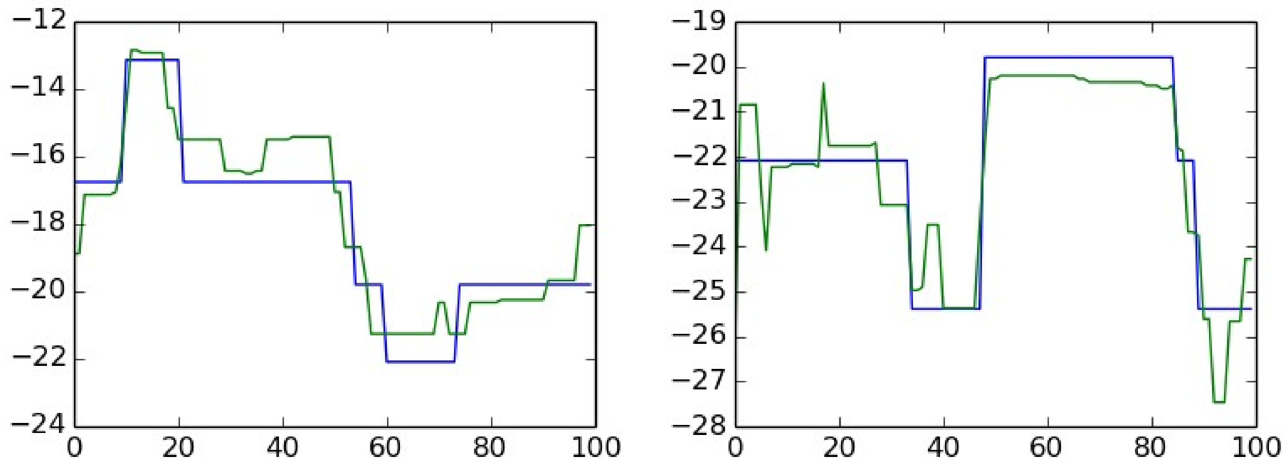


Figure 4: Fitting the HMM with likelihood data. In green are the actual likelihood traces from two runs of MH on the same Venture program. In blue is the "true value" of the likelihood at each point, according to the model. The HMM was trained on 10 different traces in all, including these two. Log likelihood is shown on the y-axis.

The results are shown in Figure 4. Also of note is that the final value of pTrans was 0.042, which means that the MH trace has a 1-in-25 chance of making a major jump at any step, according to the model.

3 – Generating test Venture programs

To test inference algorithms in Venture, we wrote a procedure that randomly generated probabilistic programs of varying difficulty. The procedure makes a directed graphical model with nNodes binary variables. Each binary variable v_i has nParents parents, randomly chosen from $\{v_0, \ldots, v_{i\text{-}i1}\}$. The conditional probability table at each node contains $2^{nParents}$ entries, each one defining the probability that v_i is true, given some combination of the values of its parents. These probabilities are chosen uniformly at random from [0, 1].

The result is a probabilistic program of controllable complexity. Increasing nNodes makes the program harder, by increasing the number of variables Venture has to infer. Increasing nParents also makes the program harder, by making the Markov blanket of each node larger. In the experiments described below, we used 7 nodes, and 3 parents per node. This was empirically the largest class of programs for which running 100 rounds of inference could reliably generate at least one large jump in likelihood. Such jumps are necessary for the EM algorithm to estimate a useful pTrans value.

4 – Using the HMM to control inference in Venture

To apply our model to a probabilistic program in Venture, we first need to train our model on the dynamics of that particular program. To get training data, we run several short inference chains on the program first. In these experiments, we run 10 chains of length 100. (No other possibilities were tried.) Using these chains, we fit an HMM, according to section 2. We then evaluate our model on finding the global maximum in one long attempt, of length 1000 (once again, the only length I tested).

Now, we want to use the HMM to inform our decision of whether to restart the inference algorithm. We could try to infer the bin distribution in the HMM, given the likelihoods coming out of the inference algorithm so far. But, we run into a problem: ideally, the long inference chain will return very high likelihoods, which will exceed the top bin in our HMM. In that case, we can't make any meaningful predictions, because we can only predict that the likelihood will go down. To avoid this, we instead look to see how much likelihoods vary in a sliding window. Suppose the median bin size (the difference between the true values of two consecutive bins) is binSize. Then, our model says that the likelihood spends on the order of 1/pTrans turns inside a range of binSize, before jumping away. Therefore, if we see that the likelihood has been within the range of binSize for the last Θ(pTrans) turns, we can conclude that we are likely stuck at a local optimum, and reset the algorithm. Unfortunately, this method requires manual tuning of the timeout: we wait for k * pTrans turns, where k must be specified ahead of time. In our experiment, we used k = 1.

We test our algorithm, as well as regular MH, on 40 programs. We find that our reset strategy results in significantly more likely optima when testing on randomly generated Venture programs. Figure 5 (in blue) shows a histogram of the difference in score between our algorithm and the Venture MH algorithm. (Since these scores are log likelihoods, a positive score of 10 means that our algorithm found a configuration of hidden variables that is 20,000 times more likely.)

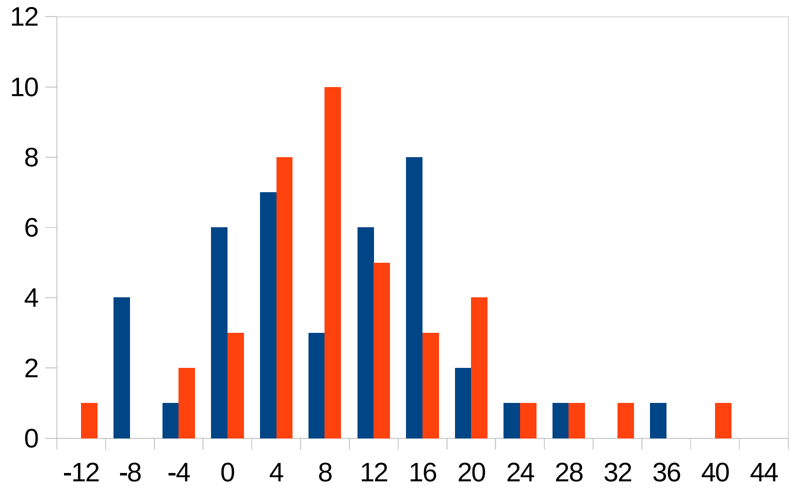
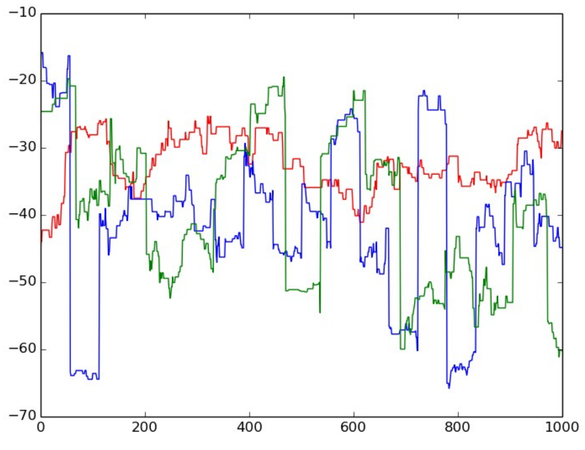


Figure 5: Difference in likelihood of the best state found by resetting, versus vanilla MH. Positive means resetting did better. In blue is our algorithm (mean = 12.2, SD = 10.0). In red is a control, which resets the same number of times as our algorithm, but at regular intervals (mean = 10.4, SD = 10.1).

However, in the course of running these experiments, we found a potential confounding variable. The MH algorithm tended to find better optima when it was reset more often, regardless of the timing. Ideally, we would like to show that our algorithm resets the sampler at “good” times – that it does better than randomly resetting the sampler the same number of times. To test this, we added a second control, which reset the MH sampler at fixed intervals, so that the total number of resets was consistent with our algorithm on each problem. The results of this control, when compared to regular MH, are shown in red in the Figure 5.

The control that resets at a fixed interval achieves an average gain of 10.4 points over regular MH. This is most of the 12.2 point gain that our algorithm achieves. Indeed, a mean difference significance test shows that our algorithm is better than this control only with a p = 0.07. Therefore, while the data so far suggest that our algorithm is better than uninformed resetting, there are not enough samples to definitely tell.

Finally, we are interested in not just finding global optima, but also in sampling the entire distribution over posteriors. We want our sampling algorithm to visit all of the possible states of the posterior in a well-mixed way. While we don't know how to formally test for well-mixing in the context of our programs, we can qualitatively see in Figure 6 that resetting – both periodic (green) and controlled (blue) – helps MH explore states with a much wider range of likelihoods, including many states that are much more likely. A possible direction for future work is formally measuring how well various modifications to the MH algorithm sample all states fairly.



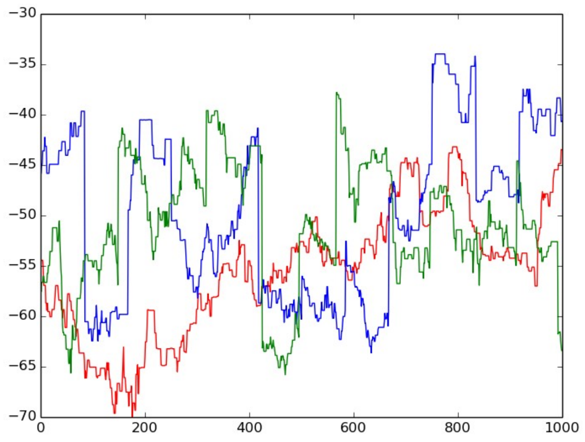


Figure 6: Likelihood values over the course of a long inference operation, for two different problems. Red: default MH. Blue: control with resetting at fixed intervals. Green: our algorithm. Note that both resetting algorithms sampled from states with much higher likelihood than default MH.

5 – Discussion and summary

It is rather concerning that simply resetting the MH sampler more often results in higher likelihood scores. In theory, MH uses gradient ascent to locate states with high likelihood faster than would be possible with random guessing. In Figure 6, it appears that this is not the case. When the sampler is reset to a random state, the likelihood value often makes a big positive jump. This suggests that randomly sampling from the prior distribution is actually more likely to result in a high-likelihood state than using MH, at least for this problem. I am troubled by this conclusion – it may mean that our random problem generator in fact makes problems that are ill-posed for probabilistic programming.

In general, our work shows that it is possible to train a HMM to model the dynamics of a sampling algorithm, and extract useful information from the HMM that reflects the dynamics of the sampling algorithm. We use this information to improve the performance of the sampling algorithm. Future work could involve more sophisticated models of the likelihood score as a function of time. Perhaps the distribution of likelihood scores could be estimated, giving us some idea of what the best possible likelihood score may be. (In contrast, our current method only estimates bins of likelihoods – it has no idea how often each bin is used.) Estimating the distribution of likelihood scores would require a good prior on this distribution, which would in turn require deeper knowledge of the dynamics of MH.

Another possible avenue of approach is to allow a variable number of bins. This would require a non-parametric model of the likelihood, like a Chinese restaurant process. With a non-parametric bin model, we could incrementally update the model as the MH sampler takes each sample. Based on the model so far, we could then decide whether to reset the sampler.