SNAP-Haskell

Abraham Starosta
March 18, 2016
Code: https://github.com/astarostap/cs240h_final_project

Abstract

The goal of SNAP-Haskell is to become a general purpose, high performance system for analysis and manipulation of large networks. SNAP-Haskell can create large networks of thousands of edges, and then perform inference operations on them. Thus far, the main application of SNAP-Haskell is image denoising through Gibbs Sampling. As Bayesian and Markov models become more sophisticated, the need for efficient inference methods becomes more urgent. However, sampling the posterior distribution over a set of random variables is intractable as it is an NP-Hard problem. This is why tractable sampling techniques to approximate posterior distributions such as those based on Markov Chain Monte Carlo (MCMC) models (Gibbs Sampling being one of them) have become increasingly popular statistical tools, both in applied and theoretical work.

Introduction to Gibbs Sampling

The underlying logic of MCMC sampling is that we can estimate any expectation by calculating over averages.

$$\hat{g} = \frac{1}{T}\sum_{t=1}^{T} g(x^t) \rightarrow E_P[g(x)] \text{ for } T \rightarrow \infty$$

For instance, here *g(x)* is the desired expectation, *T* is the number of samples, and *P* is the posterior distribution. As we approach an infinity of samples, our estimate is theoretically bound to approach the exact expectation of *g(x)*.

In order to obtain samples from the posterior distribution, we can use Gibbs sampling. The main idea of Gibbs sampling is to go through each variable, and sample its conditional distribution with the remaining variables fixed on their current values. This is one iteration, which enables us to obtain one sample for each variable.

Pseudocode:

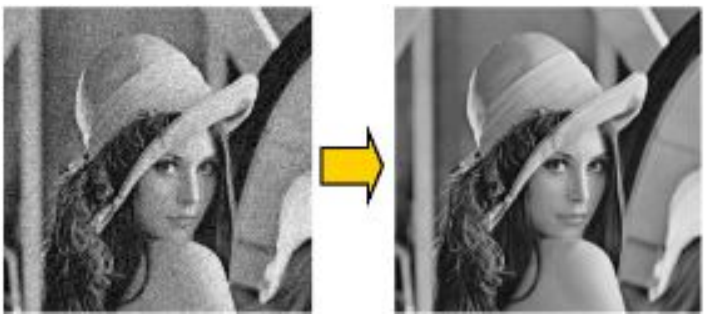
Algorithm 1 Gibbs sampler

Initialize $x^{(0)} \sim q(x)$
for iteration $i = 1, 2, \ldots$ **do**
  $x_1^{(i)} \sim p(X_1 = x_1 | X_2 = x_2^{(i-1)}, X_3 = x_3^{(i-1)}, \ldots, X_D = x_D^{(i-1)})$
  $x_2^{(i)} \sim p(X_2 = x_2 | X_1 = x_1^{(i)}, X_3 = x_3^{(i-1)}, \ldots, X_D = x_D^{(i-1)})$
  $\vdots$
  $x_D^{(i)} \sim p(X_D = x_D | X_1 = x_1^{(i)}, X_2 = x_2^{(i)}, \ldots, X_D = x_{D-1}^{(i)})$
end for

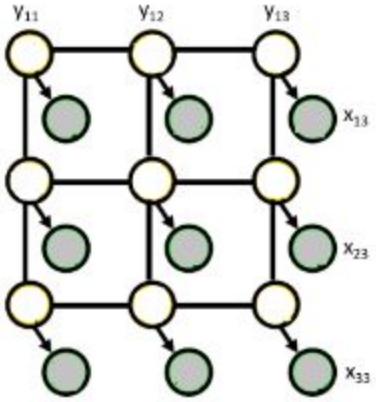
Thus, we can see that we are not sampling directly from the conditional distribution, but instead we go through all the variables, drawing a sample one variable at a time. It is very important to understand that the initial values for the variables is randomized, and thus the first iterations are not representative of the actual posterior distribution. However, MCMC theory guarantees that all there exists a stationary distribution that will be achieved. This is why we generally run Gibbs sampling for many (thousands) iterations until convergence to the stationary distribution (which represents the actual posterior distribution).

Gibbs Sampling application to Image Denoising in SNAP-Haskell

The image denoising problem consists of taking a noisy image, and attempting to recover the original image.



To do so using Gibbs Sampling, we construct the following probabilistic model:



X : noisy pixels
Y : “true” pixels

Where the Markov Random Field X represents the observed image, and Y is another Markov Random Field representing the true (unknown) image before noise is added. Both having dimensions $Nx\,M$. In SNAP-Haskell, the client can perform image denoising for black-and-white images. Thus, we set the possible values of any vertex in X and any vertex in Y to $\{-1,\ +1\}$. Each vertex Yij is linked to four neighbors in Y (top, left, right, and bottom vertices). For those vertices at the borders of the image, we have less that fours neighbors.

We let E represent the corresponding set of edges to the neighbors of Yij. Therefore, we represent the joint probability of X and Y as:

$$p(\boldsymbol{y},\boldsymbol{x}) = \frac{1}{Z}\left\{\prod_{i=1}^{N}\prod_{j=1}^{M}\exp^{\eta y_{ij}x_{ij}}\right\} \times \left\{\prod_{((i,j),(i',j'))\in E}\exp^{\beta y_{ij}y_{i'j'}}\right\}$$

This distribution is the Boltzmann distribution.
Now, we can easily derive the conditional probability of given vertex Yij to be black, given its neighbors to be:

$$\frac{1}{1+e^{-2\eta x_{ij}-2\beta\sum_{y\in y_{M(i,j)}\setminus x_{ij}}y}}$$

Now that we can calculate the conditional probabilities of a pixel being black, given its neighbors, we can implement Gibbs sampling (pseudocode previously provided).

Design and Implementation

Types (TNTypes.hs):

```
-- |@TNGraphType@ holds all the possible types of graph.
-- @TNDGraph@: directed graph
-- @TNUGraph@: undirected graph (for each edge going out a vertex, there is another edge coming into it).
data TNGraphType = TNUGraph | TNDGraph deriving (Eq, Ord, Show)

-- |@TNGraph@ is the TNGraph data structure
-- It stands for "Type Network Graph"
data TNGraph =  TNGraph {
                        table :: TNTable [TNVertex],
                        graphType :: TNGraphType,
                        vertexValues :: TVertexValueMap
                } deriving (Show)
```

For more information on other types, please refer to the documentation in TNTypes.hs.

Creating TNGraphs (TNParser.hs):

Allows the client to create TNGraphs by supplying an input file that specifies the edges in the graph, as well as the values for each vertex. The parser is implemented using Text.ParserCombinators.Parsec.
For instance, the input file must be of this form:

```
#edges
0       4
1       5
2       6
3       7
#values
0       -1
1       -1
2       1
3       1
4       -1
5       1
6       1
7       1
```

SNAP-Haskell comes with several example input files in the folder snap-haskell/examples/

```
-- |Parses file into a TNGraph
parseTNGraphEdgesFile :: String -> Either String TNGraph
```

TNGraphs API (TNGraph.hs):

Allows the client to perform general operations on TNGraphs.

Main operations supported:

```
-- | Convert a graph into another type.
-- Currently only implemented TNDGraph (directed) -> TNUGraph (undirected)
convertGraph:: TNGraph -> TNGraphType -> TNGraph

-- | Get all neighbors for a vertex in a graph
getNeighbors:: TNGraph -> TNVertex -> [TNVertex]

-- | Get the value assigned to a vertex in a graph
getValue:: TNGraph -> TNVertex -> Maybe TNVertexValue

-- | Assign a new value to a vertex in a graph
insertValue:: TNGraph -> TNVertex -> TNVertexValue -> TNGraph

-- | Get the graph's type (TNDGraph, or TNUGraph)
getGraphType:: TNGraph -> TNGraphType

-- | Creates a new graph where the values are the union of the old graph and the
 -- new value map
addValuesToGraph:: TNGraph -> TVertexValueMap -> TNGraph
```

There are other operations, but they were not worth including here because they aren't as useful. Please refer to TNGraph.hs for more information.

Gibbs Sampling API (GibbsSampler.hs):

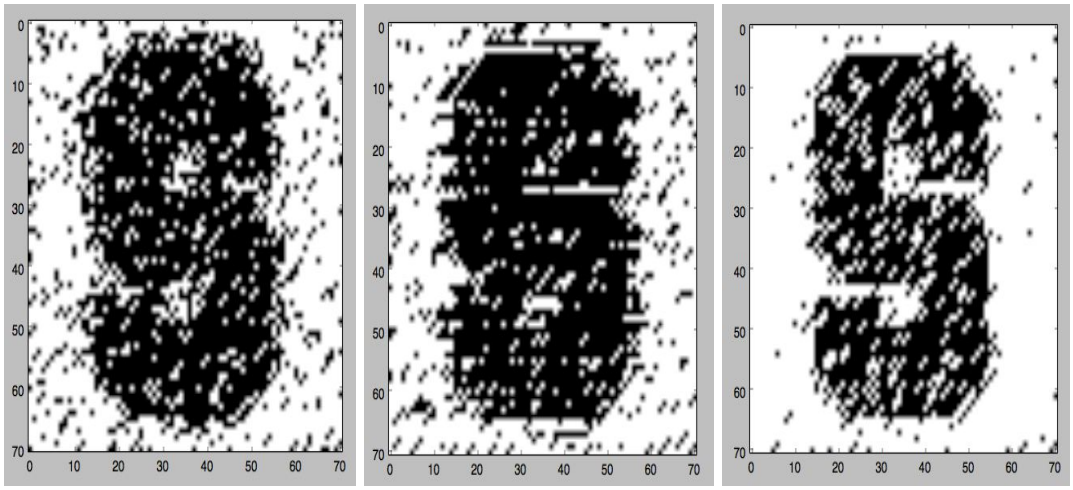
This module allows the client to perform Gibbs Sampling for image denoising, using TNGraphs.

-- |Get posterior probability for all vertices in Y
getPosteriorSampling:: TNGraph -> TNGraph -> Int -> StdGen -> TNGraph
getPosteriorSampling xGraph yGraph numSamples randGen

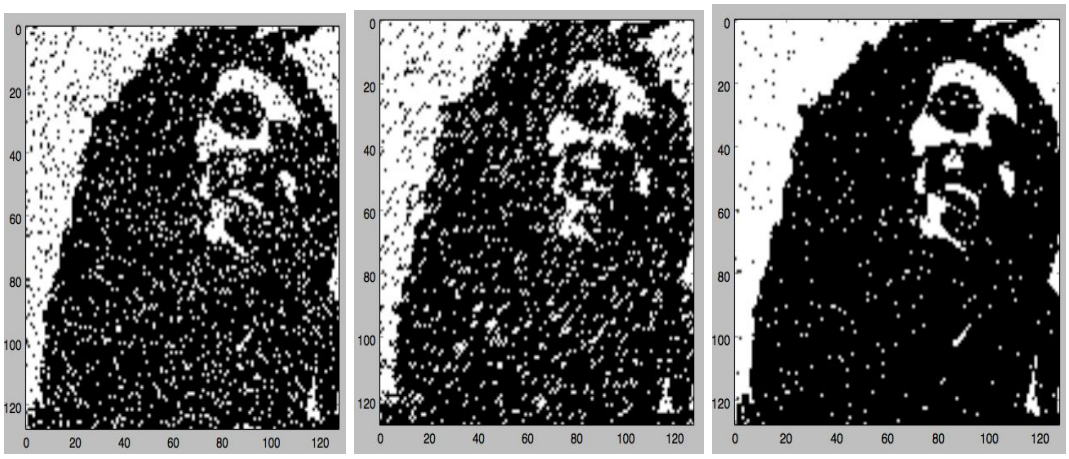
The X graph, Y graph, and numSamples correspond to our previous explanation of the image denoising model. The random generator must be passed in from an impure IO function into our pure function.
The output is the network for the posterior distribution of the sampling iterations.

Examples of Image Denoising

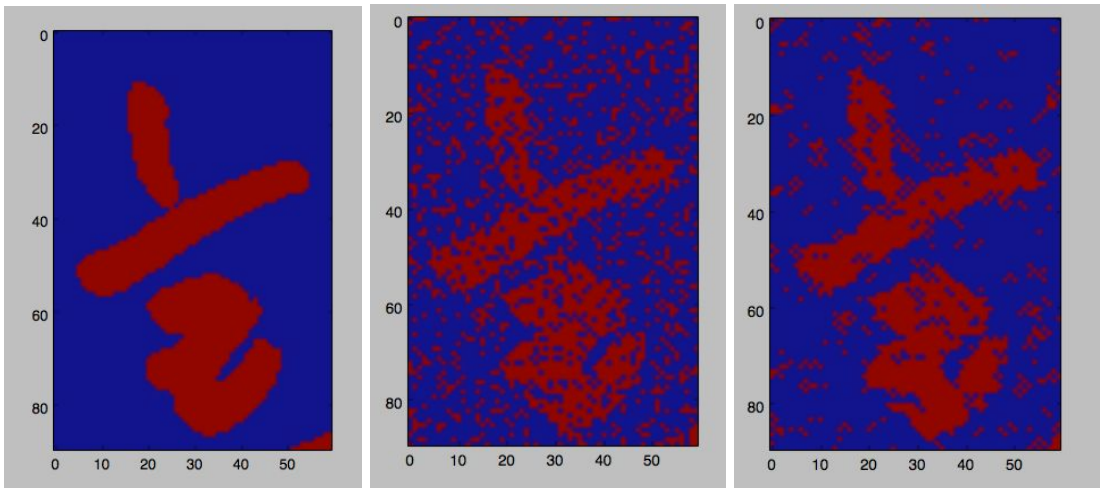
20% noise **100 samples** **1000 samples**



20% noise **100 samples** **1000 samples**



Original **20% noise** **1000 samples**



Conclusion and Future Work

In all of our examples, it is interesting to see that the more samples we get, the closer we get to the MCMC stationary distribution, and thus we get an output posterior distribution that is closer to the original image. It was very interesting to implement probabilistic graphical models and randomized algorithms in a functional language such as Haskell. It was also very rewarding to see the tangible results of fixing noisy images. In the future, I want SNAP-Haskell to become the library where algorithms on networks are implemented functionally.

Gibbs Sampling is just one instance of MCMC sampling techniques, but there are ways to generalize MCMC sampling techniques so that the client can use SNAP-Haskell to implement any MCMC sampling technique they desire. Thus, for future work, I will generalize my implementation of Gibbs Sampling so that the client could specify other types of more complex networks and more distributions (besides Boltzmann's distribution).

Bibliography

Yildirim, Ilker. "Gibbs Sampling." *Bayesian Computation with R* (2012): 211-36. MIT. Web. (http://www.mit.edu/~ilkery/papers/GibbsSampling.pdf)

Casella, George. "Exaplaining Gibbs Sampler." *The American Statistitian* (n.d.): n. pag. Web. (http://www.stat.ufl.edu/archived/casella/OlderPapers/ExpGibbs.pdf)

Ermon, Stefano. "CS228 Assignment 4." *CS228 Assignment 4* (2016): n. pag. Stanford University. Web.