Qualitative Probabilistic Programming

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

In probabilistic programs, sometimes it is difficult to specify the correct parameterized family of distributions. We explore an extension to probabilistic programming languages that allows programmers to mark some distributions as unspecified. Then, we can fill in the distribution with some family and infer parameters.

1 Introduction

By separating model specification and inference, probabilistic programming has made it easier for non-experts to implement and use probabilistic models. Practitioners frequently have strong intuitions about the *structure* of their domain knowledge, such as which latent variables exist and what their causal relations are, and probabilistic programming allows them to encode this knowledge. However, it also requires them to specify the specific parametric shape and parameterization of any distributions used, and intuitions tend to be much less precise there. We present Quipp, a system that does *not* require such specification; instead, random variables and random functions can be left undefined and will automatically be filled in under maximum entropy assumptions based on their types and available datasets.

Our formalism can concisely express a wide variety of models that machine learning practitioners care about, and we provide an expectation maximization algorithm that can learn the parameters for many of these models with reasonable efficiency. This system makes it easy for non-experts to encode their beliefs about the data and to get predictions based on as few additional assumptions as possible.

In an ordinary probabilistic programming language (such as Church), it is possible to treat parameters as random variables. This would allow ordinary inference algorithms to infer parameters. However, there are advantages of having unknown functions as a feature in the language. First, it is easier to write programs without knowing the details of different parameterized distributions. Second, the system can use specialized algorithms to infer parameters faster.

In the following, we first specify the syntax used to write Quipp programs, including the notation for unknown variables and functions. We describe the class of exponential family variables and functions that our system can learn, and present the expectation maximization algorithm used to learn them. We then demonstrate the expressiveness of our language, and the broad applicability of our algorithm, by writing some of the most common machine learning models in Quipp: clustering, naive Bayes, factor analysis, a Hidden Markov model, Latent Dirichlet Allocation, and a neural net.

2 Syntax

Quipp is implemented as a library for webppl programs. Webppl [2] is a probabilistic programming language that is similar to Javascript but also contains features for generating random values, conditioning on values, and estimating expectations. Quipp programs are written as webppl programs that have access to additional special functions.

Here is an example of a Quipp program to cluster 2d points into 2 clusters:

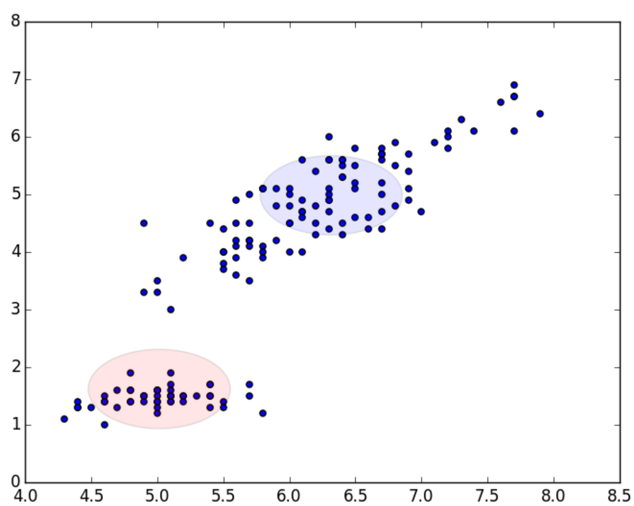
```
var Cluster = Categorical(2);
var Point = Vector(2, Double);
var getPoint = randFunction(Cluster, Point);

var model = function() {
  var cluster = randomValue(Cluster);
  observe(getPoint, cluster);
};
```

We declared two types (`Point` and `Cluster`) and one random function (`getPoint`). Type annotations are necessary for random functions. The type `Vector(2, Double)` expresses the fact that the points are 2-dimensional, and the type `Categorical(2)` expresses the fact that there are 2 possible clusters (so a cluster may be either 0 or 1).

The variable `model` specifies a generative model for a single data point. We use the `randomValue` function to generate a random uniform cluster, and then use `getPoint` to generate the point given the cluster. Since `getPoint` is an unknown random function, we will need to infer its parameters. The `observe` function allows us to observe data; here it says that our observation consist of the result of the call `getPoint(cluster)`.

To demonstrate, let us run this example on a dataset consisting of 150 points (TODO cite). When we run the program on this data, we infer the parameters to the random function `getPoint`. In this case, `getPoint` is a linear function with Gaussian noise, so it will naturally split the data into 2 clusters with equal variance:



The first cluster is at (6.3, 5.0) and the second is at (5.0, 1.6). They both have a standard deviation of 0.54 in the x direction and 0.69 in the y direction. We could use these parameters to fill in the generative model:

```
var model = function() {
  var cluster = randomInteger(nclusters);
  return [gaussian(cluster == 0 ? 6.3 : 5.0, 0.54),
          gaussian(cluster == 0 ? 5.0 : 1.6, 0.69)];
};
```

This model is estimated to assign probability density e^{-393} to the data, yielding a perplexity of $e^{-393/150} = 0.0728$.

3 Family of distributions

In the previous example, we that `randFunction(Categorical(2), Vector(2, Double))` represented 2 axis-aligned 2-dimensional Gaussian clusters with equal variances. In general, `randFunction` returns a randomized function that is a member of some generalized linear model determined by the desired argument types and return type. The distribution of the function's return value y is some exponential family whose natural parameters are determined from the arguments x:

$$p_\eta(y|x) = \exp\left(\eta(x)^T\phi(y) - g(\eta(x))\right)$$

Here, $\eta(x)$ is the natural parameter, $\phi(y)$ is a vector of y's sufficient statistics, and g is the log partition function.

To determine $\eta(x)$, we label some subset of the sufficient statistics of both x and y as *features*. For the gaussian distribution, the sufficient statistics are X and X^2 but the only feature is X. For the categorical distribution `Categorical(n)`, the sufficient statistics and features are both $[X = 1], [X = 2], ..., [X = n - 1]$. The natural parameters corresponding to non-features are constant, while natural parameters corresponding to features are determined as an affine function of the features of the arguments.

Let $\psi(x)$ be the features of x. Then

$$\eta(x) = \mathbf{N}^T \begin{bmatrix} 1 \\ \psi(x) \end{bmatrix}$$

$$p_\mathbf{N}(y|x) = \exp\left(\begin{bmatrix} 1 \\ \psi(x) \end{bmatrix}^T \mathbf{N}\phi(y) - g\left(\mathbf{N}^T \begin{bmatrix} 1 \\ \psi(x) \end{bmatrix}\right)\right)$$

where $\mathbf{N}$ is a matrix containing our parameters. It must have 0 for each entry whose row corresponds to a sufficient statistics of y that is not a feature and whose column is not 1. This ensures that only the natural parameters that are features of y are affected by x.

Additionally, we provide a `randomValue` function for convenience, which returns a sample from the "default" distribution for a given type. The following shows the types in Quipp and their corresponding sufficient statistics, features, and English descriptions.

T	$\phi_T(X)$	$\psi_T(X)$	`randomValue(T)`	Random function class
`Double`	$\begin{bmatrix} X \\ X^2 \end{bmatrix}$	$[X]$	$\mathcal{N}(0, 1)$	Linear regression
`Categorical(n)`	$\begin{bmatrix} [X=1] \\ [X=2] \\ \dots \\ [X=n-1] \end{bmatrix}$	$\begin{bmatrix} [X=1] \\ [X=2] \\ \dots \\ [X=n-1] \end{bmatrix}$	uniform	n-class logistic regression
`Tuple(T1, T2)`	$\begin{bmatrix} \phi_{T1}(X_1) \\ \phi_{T2}(X_2) \end{bmatrix}$	$\begin{bmatrix} \psi_{T1}(X_1) \\ \psi_{T2}(X_2) \end{bmatrix}$	`[randomValue(T1), randomValue(T2)]`	Independent regression

Note that `Vector(n, T)` is shorthand for `Tuple([T, T, ..., T])` (with n copies of T), and `Bool` is shorthand for `Categorical(2)`.

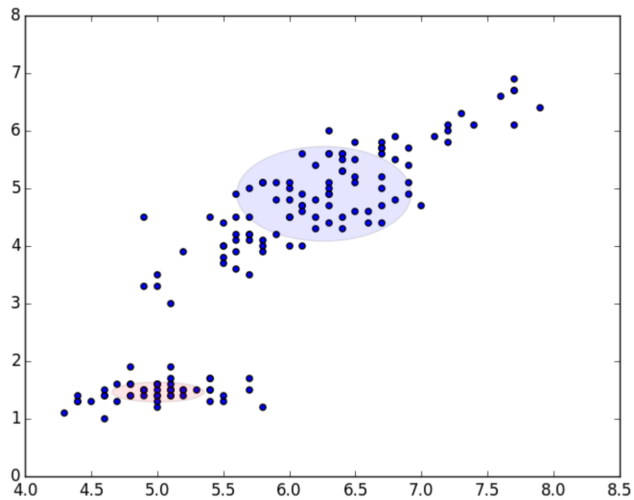
4 Example: improving the clustering model

Using this knowledge about the form of the random functions, we can improve the clustering model from before. As observed in the previous graph, the two clusters are forced to have the same variance. They do not fit the data well, since the data has a different shape in each location. To fix this problem, we can substitute the following model, which uses a separate random function (and therefore a separate axis-aligned Gaussian distribution) for each cluster:

```
var Cluster = Categorical(2);
var Point = Vector(2, Double);
var getPointFunctions = [randFunction(Point), randFunction(Point)];
```

```
var model = function() {
  var cluster = randomValue(Cluster);
  observe(getPointFunctions[cluster]);
};
```

Using this model, we get the following clusters:



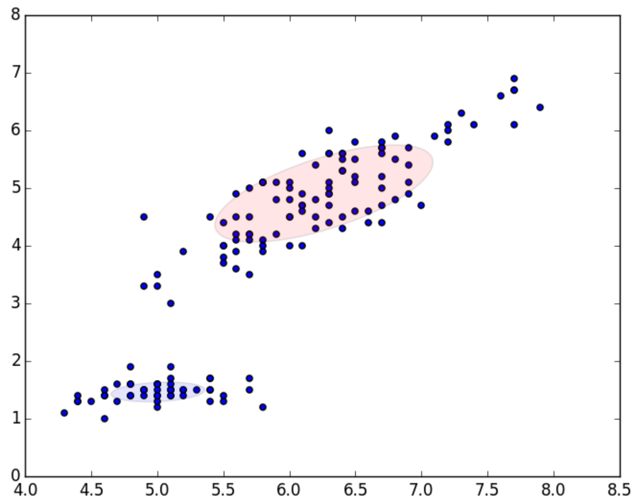
This model (with the parameters filled in) is estimated to assign probability density e^{-337} to the data, yielding a perplexity of 0.1058. This is a significantly improved fit.

The fact that the gaussian distribution for each cluster must be axis-aligned limits the degree to which the model can fit the data. To allow each cluster to be an arbitrary Gaussian distributions (where the x and y coordinates may be correlated), we can use the following model:

```
var Cluster = Categorical(2);
var getX = [randFunction(Double), randFunction(Double)];
var getY = [randFunction(Double, Double), randFunction(Double, Double)];

var model = function() {
  var cluster = randomInteger(nclusters);
  var x = observe(getX[cluster]);
  var y = observe(getY[cluster], x);
};
```

Here, there are 2 random functions for each cluster, one to get the x coordinate and one to get the y coordinate (whose distribution may depend linearly on the x coordinate). This allows x and y to be correlated, improving fit:



This model is estimated to assign probability e^{-269} to the data, yielding a perplexity of 0.166. This is a large improvement from the previous model.

5 Inference

To infer both latent variables and parameters, we use a Monte Carlo expectation maximization algorithm [1] on the probabilistic model, iterating stages of estimating latent variables using Metropolis Hastings and inferring parameters using gradient descent. The first iteration uses randomly generated parameters.

For the expectation step, we must estimate latent variable distributions given fixed values for the parameters. To do this, we can replace unknown random functions in the Quipp program with random functions set to use these fixed parameter values, yielding a probabilistic program. We use the Metropolis Hastings algorithm to perform inference in this program, yielding a distribution of execution traces (where each execution trace specifies the result of every call to a random function). Next, for each random function, we can find all calls to it in the trace to get the training data.

For the maximization step, given samples from each random function, we set the parameters of the function to maximize the likelihood of the samples. To do this we, we use gradient descent.

Given (x, y) samples, parameter estimation to maximize log probability is a convex problem because the log probability function is concave as a function of $\mathbf{N}$:

$$\log p_{\mathbf{N}}(y|x) = \begin{bmatrix} 1 \\ \psi(x) \end{bmatrix}^T \mathbf{N}\phi(y) - g\left(\mathbf{N}^T \begin{bmatrix} 1 \\ \psi(x) \end{bmatrix}\right)$$

This relies on the fact that g is convex, but this is true in general for any exponential family distribution. Since the problem is convex, it is possible to use gradient descent to optimize the parameters. Although the only exponential family distributions we use in this paper are the categorical and Gaussian distributions, we can use the same algorithms for other exponential families, such as the Poisson and gamma distributions.

6 Evaluation

To evaluate performance, for each model, we:

- Randomly generate parameters θ
- Generate datasets x_{train}, x_{test} using θ

- Estimate $\log P(x_{test}|\theta)$
- Use the EM algorithm to infer approximate parameters $\hat{\theta}$ from x_{train}
- Estimate the regret $\log P(x_{test}|\hat{\theta}) - \log P(x_{test}|\theta)$

Estimating $\log P(x_{test}|\theta)$ is nontrivial, given that the model contains latent variables. We use the Sequential Monte Carlo algorithm for this, as described in [2].

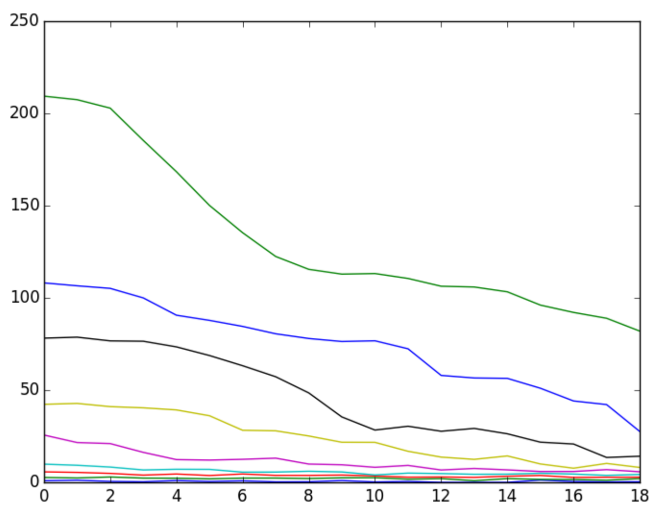
7 Examples

7.1 Clustering

```
var Cluster = Categorical(3);
var Point = Vector(2, Double);
var getPoint = randFunction(Cluster, Point);

var model = function() {
  var cluster = randomValue(Cluster);
  observe(getPoint, cluster);
};
```

In this example, we cluster 2d points into 3 different clusters. Given a cluster, the distribution for a point is some independent Gaussian distribution. This is similar to fuzzy c-means clustering.



7.2 Naive Bayes

```
var Class = Categorical(2);
var Features = Vector(10, Bool);
var classFeatures = [randFunction(Features), randFunction(Features)];

var model = function() {
  var whichClass = randomValue(Class);
  observe(classFeatures, whichClass);
};
```

The naive Bayes model is similar to the clustering model. We have two classes and a feature distribution for each. Since each feature is boolean, we will learn a different categorical distribution for each class.

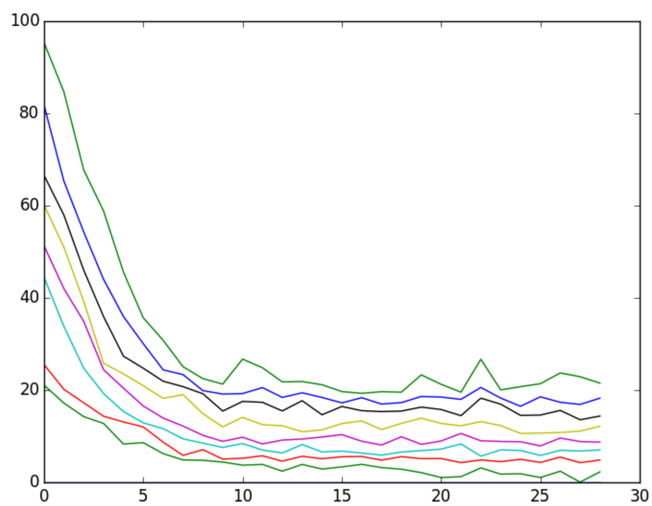
(figure should show average classification accuracy)

7.3 Factor analysis

```
var Factors = Vector(2, Double);
var Point = Vector(5, Double);
var getPoint = randFunction(Factors, Point);

var model = function() {
  var factors = randomValue(Factors);
  return observe(getPoint, factors);
};
```

The factor analysis model is very similar to the clustering model. The main difference is that we replace the categorical `ClusterType` type with a vector type. This results in the model attempting to find each point as an affine function of a vector of standard normal values.



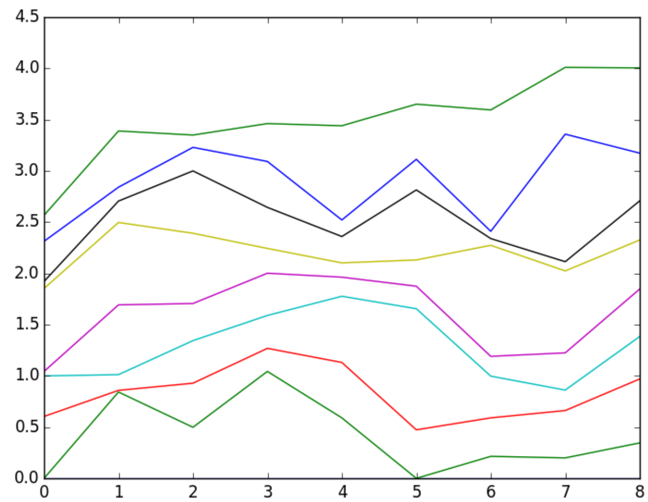
7.4 Hidden Markov model

```
var chainLength = 20;
var State = Categorical(2);
var Obs = Categorical(4);
var transFun = randFunction(State, State);
var obsFun = randFunction(State, Obs);

var observeStates = function(startState, i) {
  if (i == chainLength) {
    return [];
  } else {
    observe(obsFun, startState);
    observeStates(transFun(startState), i+1);
  }
};

var model = function() {
  return observeStates(randomValue(State), 0);
};
```

In this example, we use the unknown function `transFun` for state transitions and `obsFun` for observations. This means that we will learn both the state transitions and the observation distribution.

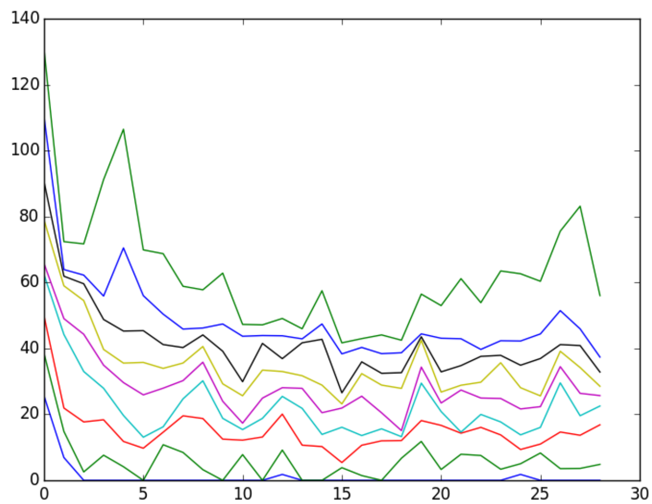


7.5 Latent Dirichlet allocation

```
var maxWordsPerDocument = 100;
var Class = Categorical(3);
var Word = Categorical(10);
var classToWord = randFunction(Class, Word);

var model = function() {
  var whichClass = randomValue(Class);
  var nWords = observe('len' + docIndex, randomIntegerERP, maxWordsPerDocument);
  repeat(nWords, function(wordIndex) {
    observe(classToWord, whichClass);
  });
};
```

In this example, we use the unknown function `classToWord` to map classes to word distributions. Note that each column of the matrix of parameters for `classToWord` will represent a categorical distribution over words, and there will be one column for each class.



7.6 Neural network

```
var Input = Vector(30, Bool);
var Hidden = Vector(10, Bool);
var Output = Bool;

var inputToHidden = randFunction(Input, Hidden);
var hiddenToOutput = randFunction(Hidden, Output);

var model = function() {
  var inputLayer = randomValue(Input);
  var hiddenLayer = inputToHidden(inputs[sampIndex]);
  observe(hiddenToOutput, hiddenLayer);
};
```

8 Discussion

We have found that it is possible to write many useful machine learning models as Quipp programs and then use generic algorithms for inference. Furthermore, performance is ¡???¿. This should make it much easier for non-experts to write useful machine learning models.

In the future, it will be useful to expand the set of types supported. It is possible to define reasonable default distributions for non-recursive algebraic data types, and it may also be possible to define them for recursive algebraic data types using catamorphisms. Also, it will be useful to create a more usable interface to infer parameters and perform additional data pracessing given these parameters.

References

[1] Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and MichaelI. Jordan. An introduction to mcmc for machine learning. *Machine Learning*, 50(1-2):5–43, 2003.

[2] Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014. Accessed: 2015-6-4.