
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 [1] 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.

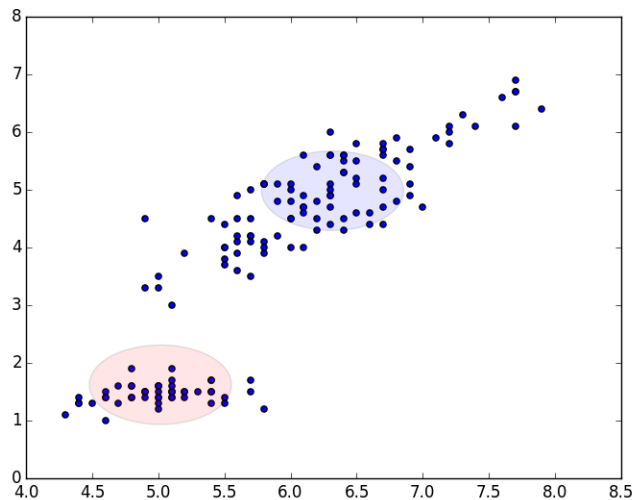
```

054 Here is an example of a Quipp program to cluster 2d points into 2 clusters:
055
056 var n = 100;
057
058 var Cluster = Categorical(2);
059 var Point = Vector(2, Double);
060 var getPoint = randFunction(Categorical(2), pointType);
061 return function() {
062   repeat(n, function(i) {
063     var cluster = randomValue(Cluster);
064     observe(getPoint, cluster);
065   });
066 };
067

```

It is written as a generative model for the observations. Notice that 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). We use the randomValue function to generate a random uniform cluster. We do not know the distribution of points in each cluster, represented by the fact that getPoint is an unknown random function. We will fill in the getPoint function with a learned function that will take a random sample from the given cluster. The observe function allows us to observe data; here it says that the observation named point3 is equal to the result of calling getPoint(cluster), where cluster is cluster number 3.

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:

```

105 return repeat(n, function(i) {
106   var cluster = randomInteger(nclusters);
107   return [gaussian(cluster == 0 ? 6.3 : 5.0, 0.54),
           gaussian(cluster == 0 ? 5.0 : 1.6, 0.69)];

```

```

108     });
109
110

```

111 This model is estimated to assign probability density e^{-393} to the data, yielding a perplexity of
 112 $e^{-393/150} = 0.0728$.

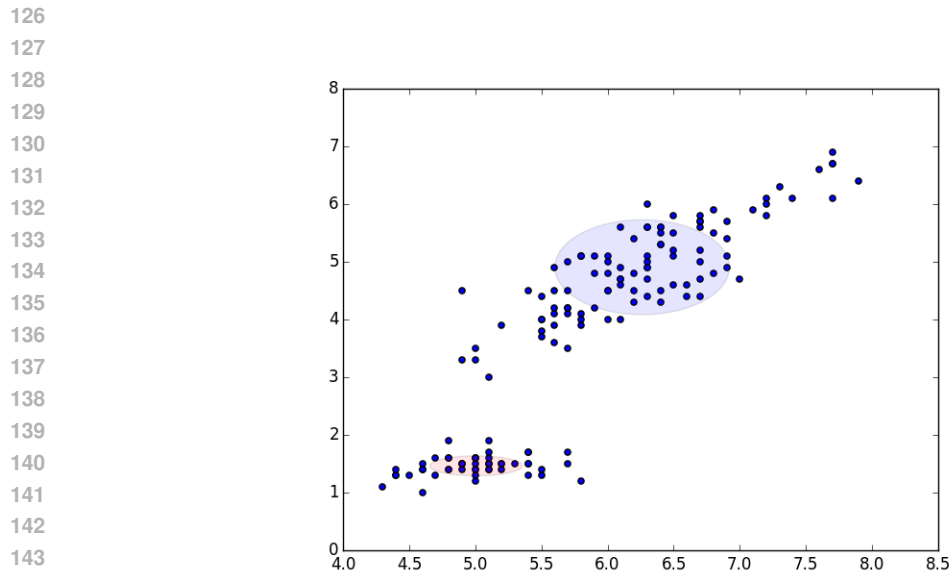
113 Note that, because the two clusters are forced to have the same variance, they do not fit the data
 114 well, since the data has a different shape in each location. To fix this problem, we can substitute the
 115 following model, which uses a separate random function for each cluster:
 116

```

117 var Cluster = Categorical(2);
118 var Point = Vector(2, Double);
119 var getPointFunctions = [randFunction(Point), randFunction(Point)];
120 return repeat(n, function(i) {
121   var cluster = randomValue(Cluster);
122   observe('point' + i, getPointFunctions[cluster]);
123 });
124

```

125 Using this model, we get the following clusters:



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

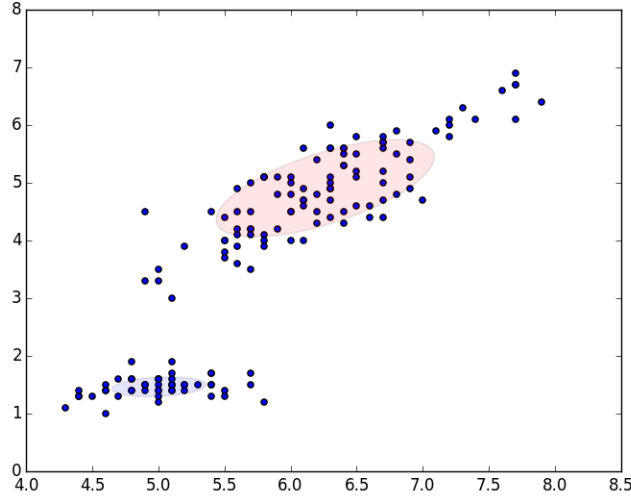
148 Note that the clusters are still restricted to have x and y be independent. To allow these coordinates
 149 to be correlated within a cluster, we can use the following model:
 150

```

151
152 var Cluster = Categorical(2);
153 var getX = [randFunction(Double), randFunction(Double)];
154 var getY = [randFunction(Double, Double), randFunction(Double, Double)];
155
156 return function() {
157   return repeat(n, function(i) {
158     var cluster = randomInteger(nclusters);
159     var x = observe(getX[cluster]);
160     var y = observe(getY[cluster], x);
161   });
162 };

```

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 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.

3 Family of distributions

For unknown functions, the family of random functions used is a kind of generalized linear model. We assume that the distribution of the function's return value y is some exponential family whose natural parameters are determined from the arguments x :

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

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 $[X = 1], [X = 2], \dots, [X = n - 1]$. 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 .

T	$\phi_T(X)$	$\psi_T(X)$	Random function class
Double	$\begin{bmatrix} X \\ X^2 \end{bmatrix}$	$[X]$	Linear regression with Gaussian noise
Categorical(n)	$\begin{bmatrix} [X = 1] \\ [X = 2] \\ \dots \\ [X = n - 1] \end{bmatrix}$	$\begin{bmatrix} [X = 1] \\ [X = 2] \\ \dots \\ [X = n - 1] \end{bmatrix}$	n -class logistic regression
Tuple(T1, T2)	$\begin{bmatrix} \phi_{T1}(X_1) \\ \phi_{T2}(X_2) \end{bmatrix}$	$\begin{bmatrix} \psi_{T1}(X_1) \\ \psi_{T2}(X_2) \end{bmatrix}$	Independent prediction of each component

4 Inference

To infer both latent variables and parameters, we run the expectation maximization algorithm on the probabilistic model, iterating stages of estimating latent variables using Metropolis Hastings and inferring parameters using gradient descent.

For the expectation step, we must estimate latent variable distributions given fixed values for the parameters. To do this, we can run the Quipp program with random functions set to use these fixed parameter values to generate their results. We use the Metropolis Hastings algorithm to perform inference in this program, yielding traces. 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:

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

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.

5 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 $\log P(x_{test}|\hat{\theta})$ and compare to $\log P(x_{test}|\theta)$

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

6 Examples

6.1 Clustering

```
var nclusters = 3;
var dim = 2;
```

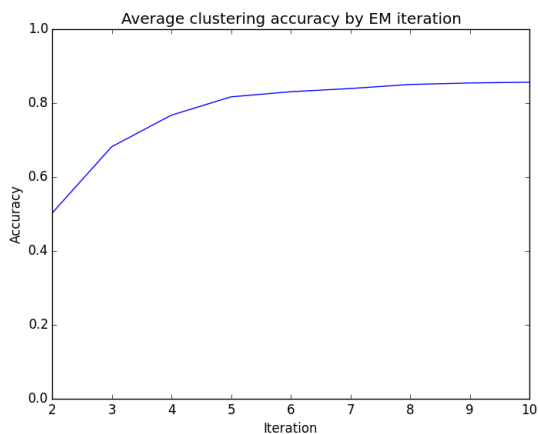
```

270 var n = 100;
271
272 var pointType = Vector(dim, Double);
273 var getPoint = randFunction(Categorical(nclusters), pointType);
274 return function() {
275     return repeat(n, function(i) {
276         var cluster = randomInteger(nclusters);
277         return observe('point' + i, getPoint, cluster);
278     });
279 };

```

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.

We randomly generated parameters for this example 100 times, and each time took 100 samples and then ran 10 EM iterations. The accuracy is defined as the maximum percentage of points assigned to the correct cluster, for any permutation of clusters. On average, accuracy increased in each EM iteration, as shown in this graph:



6.2 Naive Bayes

```

304 var nfeatures = 10;
305 var featuresType = Vector(nfeatures, Bool);
306 var nclasses = 2;
307 var n = 50;
308
309 var class0Features = randFunction(featuresType);
310 var class1Features = randFunction(featuresType);
311
312 var getFeatures = randFunction(Categorical(nclusters), pointType);
313 return function() {
314     repeat(n, function(i) {
315         var whichClass = randomInteger(2);
316         if (whichClass == 0) {
317             observe('features' + i, class0Features);
318         } else {
319             observe('features' + i, class1Features);
320         }
321     });
322 };

```

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)

6.3 Factor analysis

```
var nfactors = 2;
var dim = 4;

var n = 50;

var factorType = Vector(nfactors, Double);
var pointType = Vector(dim, Double);
var getPoint = randFunction(factorType, pointType);
var getFactors = function() {
  return repeat(nfactors, function() { return gaussian(0, 1); });
};
return function() {
  return repeat(n, function(i) {
    var factors = getFactors();
    return observe('point' + i, 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.

6.4 Hidden Markov model

```
var nstates = 2;
var nobs = 3;

var chainLength = 8;

var n = 30;

var stateType = Categorical(nstates);
var obsType = Categorical(nobs);
var transFun = randFunction(stateType, stateType);
var obsFun = randFunction(stateType, obsType);

var observeStates = function(sampIndex, startState, i) {
  if (i == chainLength) {
    return [];
  } else {
    observe('obs' + sampIndex + '_' + i, obsFun, startState);
    return [startState].concat(observeStates(sampIndex, transFun(startState), i+1));
  }
};

return function() {
  return repeat(n, function(sampIndex) {
    return observeStates(sampIndex, randomInteger(nstates), 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.

6.5 Latent Dirichlet allocation

```
var nClasses = 2;
var nWords = 100;
```

```

378 var maxWordsPerDocument = 1000;
379 var nDocuments = 20;
380
381 var classType = Categorical(nClasses);
382 var wordType = Categorical(nWords);
383
384 var classToWord = randFunction(ClassType, WordType);
385 return function() {
386   return repeat(nDocuments, function(docIndex) {
387     var whichClass = randomInteger(nClasses);
388     var nWords = observe('len' + docIndex, randomInteger, maxWordsPerDocument);
389     repeat(nWordsPerDocument, function(wordIndex) {
390       observe('word' + doc + '_' + wordIndex, classToWord, whichClass);
391     });
392     return whichClass;
393   });
394 };

```

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.

(figure should show accuracy over time. Accuracy can be measured as distance between the learned categorical distributions, for some permutation of classes)

6.6 Neural network

```

403 var inputDim = 100;
404 var hiddenDim = 20;
405 var outputDim = 2;
406
407 var inputType = Categorical(inputDim);
408 var hiddenType = Categorical(hiddenDim);
409 var outputType = Categorical(outputDim);
410
411 var inputs = [...];
412
413 testParamInference(function(randFunction) {
414   var inputToHidden = randFunction(inputType, hiddenType);
415   var hiddenToOutput = randFunction(hiddenType, outputType);
416
417   return function() {
418     return repeat(inputs.length, function(sampIndex) {
419       var hiddenLayer = inputToHidden(inputs[sampIndex]);
420       observe('output' + sampIndex, hiddenToOutput, hiddenLayer);
421     });
422   };
423 });

```

7 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 $\approx 10^3\times$ faster. 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 processing given these parameters.

432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485

References

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