Inductive Program Synthesis in BLOG

Jared Rulison

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-107
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-107.html

August 9, 2018

Acknowledgement

I'd like to thank my advisor Stuart Russell and Yi Wu, Yusuf Erol, and Karthika Mohan for their patience and guidance.

I'd like to thank Prof. Laura Waller for providing me my first research experience and Zack Phillips for acting as a mentor.

I want to thank the UC Berkeley chapter of IEEE and all its members, past and present. I'd like to thank Joel, Ben, Bobby, Willem, Chase, and Huu for sticking around so long.

Lastly, I'd like to thank Mom, Daddy, and Noah for their support in everything I've ever done.

Inductive Program Synthesis in BLOG

Jared Rulison
rulison@berkeley.edu

August 10, 2018

Abstract

I seek to discover relationships within data via synthesizing BLOG models to describe them. BLOG (Bayesian LOGic) is a first-order probabilistic programming language that details probability distributions over worlds containing sets of objects. This sort of learning has previously been done with Probabilistic Relational Models, which are a highly restricted special case of BLOG models. I synthesize programs using a local search algorithm that maximizes over the likelihood of a model with the given data while penalizing complexity. I apply the algorithm to learning the generative model describing how parts of citations are written and show that it is capable of learning accurate and useful relationships.

Acknowledgements

I'd like to thank my advisor Stuart Russell and Yi Wu, Yusuf Erol, and Karthika Mohan for their patience and guidance.

I'd like to thank Prof. Laura Waller for providing me my first research experience and Zack Phillips for acting as a mentor.

I want to thank the UC Berkeley chapter of IEEE and all its members, past and present. I'd like to thank Joel, Ben, Bobby, Willem, Chase, and Huu for sticking around so long.

Lastly, I'd like to thank Mom, Daddy, and Noah for their support in everything I've ever done.

1 Introduction

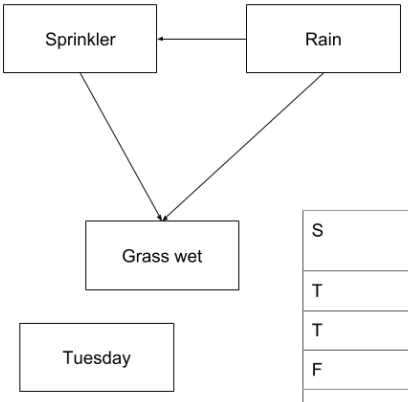
With the continuing explosion in the amount of data available for analysis comes the desire to find structure and relationships within it. Early and effective ways of modeling these relationships include Bayesian networks (BNs), which codify discrete or continuous random variables and their dependencies. Bayesian networks relate propositions, which are either true or false and are not predicated on any sort of object. There are also Probabilistic Relational Models (PRMs), which are capable of modeling relational data between object instances. PRMs are a restricted form of first-order languages, in which predicates can be quantified by variables (but not predicates). See the next section for a more in-depth examination of these models.

In this report I will use BLOG (Bayesian LOGic) [15], a first-order probabilistic programming language that describes probability distributions over worlds containing sets of objects of possibly unknown quantity. Probabilistic programming languages (PPLs) describe probabilistic models and perform inference over these models. BLOG is first-order in the same sense that PRMs are, where predicates can be quantified by objects. My goal is to produce a generative BLOG model that describes a given dataset while minimizing model complexity. BLOG can represent any PRM, as well as introduce built-in and user-defined conditional distributions that allow for a much wider space of probabilistic models. In addition, BLOG allows for an unknown number of objects and uncertainty regarding what observations correspond to which object.

The problem of probabilistically modeling datasets can also be tackled as filling in "holes" in a provided program template while maximizing likelihood over a dataset [17]. Here I instead allow for any part of a provided program to be altered rather than restricted sections. This expands the allowed search space while still taking advantage of provided human knowledge.

The standard method of learning a PRM to describe a dataset is local search maximizing likelihood of a candidate model with the given data [9]. No changes that decrease likelihood are allowed. Beginning with no relationships between objects, each possible addition, removal, or reversal of a dependency is attempted until the score no longer improves. While this is prone to getting stuck in local maxima, compared to MCMC methods, this allows the ultimate model to be formed via discrete and easy understood steps. I adapt this algorithm to BLOG to create Algorithm 1.

BLOG's sampling algorithms allow us to approximate the likelihood of a model given data without having to explicitly compute it. In PRM search, when a dependency is added between two variables, the conditional probability distribution (CPD) is assumed to take the form of a table where each value the parent variable can assume corresponds to a distribution over the possible values of the child variable. When searching over BLOG models, in addition to introducing



P(R = T)	P(R = F)
0.2	0.8

P(T = T)	P(T = F)
0.14	0.86

R	P(S = T | R)	P(S = F | R)
T	0.01	0.99
F	0.4	0.6

S	R	P(G = T | S, R)	P(G = F | S, R)
T	T	1	0
T	F	0.9	0.1
F	T	0.8	0.2
F	F	0	1

Figure 1: A Bayesian network modeling the wetness of grass. In the CPTs, the random variables are abbreviated to their first letters.

possible dependencies, Algorithm 1 also considers a larger space of distributions. Also, when the algorithm find multiple models that are effective at modeling some parts of the data but not others, it combine these models to produce one that is able to harness the expressive power of each of the constituent models.

An often used domain for demonstrating the capabilities of BLOG is citation matching, where given a set of citation strings the goal is to reconstruct the original set of papers and authors being cited [11]. I evaluate Algorithm 1 by having it model the sub-tasks of how author's names and journal names are cited in paper citations and show that is indeed able to capture common citation methods.

2 Technical Preliminaries

2.1 Bayesian Networks

Bayesian networks are a family of graphical models used to quantify uncertainty among random variables and their dependencies [1]. Specifically, a Bayesian network B is defined by a directed acyclic graph G over continuous or discrete random variables $X_1, X_2, ...X_n$, and parameters θ which describe the CPD parameters $\theta_{x_i|\pi_i} = P_B(x_i|\pi_i)$ [21]. Here x_i refers to an assignment of some value for X_i, and π_i refers to some assignment to the parents Π_i of X_i. A variable X_j is a parent of X_i if G specifies an edge from X_j to X_i.

Figure 1 is an example of a Bayesian network along with the associated CPTs. This model represents the knowledge that if it is raining, $R = T$, then the sprin-

klers are likely not to activate. Also, if it is raining or the sprinklers are on, $S = T$, then the grass is likely to be wet. Notice the variable Rain has no incoming edges, meaning it has no parents. As a result, its probability distribution is not conditioned on the other variables. The variable Tuesday is disconnected from the other variables, meaning its assignment has no correlation with the other variable assignments.

In general, the probability of a given assignment to the variables of a Bayesian network is calculated as

$$P_B(X_1 = x_1, X_2 = x_2, ..., X_n = x_n) = \prod_{i=1}^{n} P_B(X_i = x_i | \Pi_i = \pi_i) = \prod_{i=1}^{n} \theta_{x_i|\pi_i}.$$

In the case of Figure 1, this is

$$P_B(R = r, S = s, G = g) = P(R = r)P(S = s|R = r)P(G = g|S = s, R = r).$$

2.2 Probabilistic Relational Models

PRMs are designed with relational databases in mind. Their purpose is to be able to learn from the relationships between object instances. In the context of a bayes net, each individual object instance would an individual rule to describe its probabilistic relationships to other objects. A PRM, being first-order, can instead use a predicate quantified by any object in an object class, resulting in a much more concise representation [9]. A PRM is composed of two components: a description of the domain over which it operates and a graphical model describing probabilistic relationships. See Figure 2 for an example domain and a high-level overview of the relationships.

A PRM is defined over a schema describing a domain with classes $\mathcal{X} = \{X_1, X_2, ..., X_n\}$. In Figure 2, these classes are **Movie**, **Actor**, and **Appears**. Additionally, each class X has a set of attributes $\mathcal{A}(X)$, with a specific attribute denoted as $X.A$. The space of values of $X.A$ is denoted as $\mathcal{V}(X.A)$. For example, the **Movie** class has attributes Title, Process, Decade, and Genre, each with a fixed domain.

As in relational databases, objects can refer to other objects with what is known as a reference slot. The set of reference slots for a class X is called $\mathcal{R}(X)$, and $x.\rho$ refers to a specific reference slot. Each slot ρ has a domain and range, Dom[ρ] and Range[ρ]. The domain is the class which holds the reference to another object, and the range is the class it can reference. For example, the **Appears** class has one reference slot with domain **Appears** and range **Movie**, and another with domain **Appears** and range **Actor**. A slot chain is the chaining together of multiple reference slots.

The other portion of a PRM is a probability distribution over instances of a schema fitting the above description. A PRM defines a set of parents for each attribute $X.A$ as $Pa(X.A)$. One possible type of parent is simply another attribute

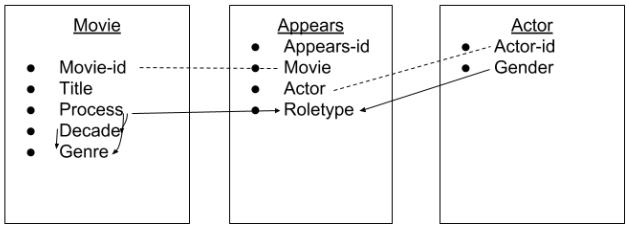


Figure 2: An example PRM describing actors, movies and roles played by actors in movies. An arrow denotes a probabilistic dependency and a dotted line denotes an equivalency.

of X, $X.B$. An example of this is **Movie**.Genre depending on **Movie**.Decade. The parent can also be reached via a reference slot, as **Appears**.Roletype depends on **Appears**.**Actor**.Gender.

The other kind of parent involves an aggregation. It is sometimes the case that a reference slot will refer to a set of instances of a class. For example, an **Actor** might appear in many movies, and therefore the database may contain many **Appears** instances whose **Actor** reference slots point to the same actor. In these cases, some form of aggregation is used over the relevant attribute of the set of instances, such as an average or minimum (in the case of a sorted domain).

The assignment of parents to attributes forms a structure $\mathcal{S}$. A PRM defines CPTs over each $X.A$ given assignments to $Pa(X.A)$. These are similar to the tables shown in Figure 1, but each CPT is universally quantified, meaning it applies to each instance of an object class rather than having a separate CPT for each instance. The parameters of the CPTs forming $\mathcal{S}$ are called $\theta_{\mathcal{S}}$.

A relational skeleton σ_r defines a set of objects fitting some schema, but does not specify the values of the attributes. An instantiation $\mathcal{I}$ of a relational skeleton σ_r is the same set of objects as σ_r but with the values for the attributes set. See Figure 3 for an example skeleton in the movie domain. An instantiation differs by having defined values for each object's attribute.

The joint distribution for the assignment to the variables of a PRM is very similar to that of a Bayesian network, but instead of multiplying over every variable one multiplies over every attribute of every instance of every class. See Figure 4 for an illustration of how some object instances in a PRM instantiation can be expanded as a BN.

$$P(\mathcal{I}|\sigma_r, \mathcal{S}, \theta_{\mathcal{S}}) = \prod_{X_i} \prod_{A \in \mathcal{A}(X_I)} \prod_{x \in \sigma_r(X_i)} P(\mathcal{I}_{x.A}|\mathcal{I}_{Pa(x.A)})$$

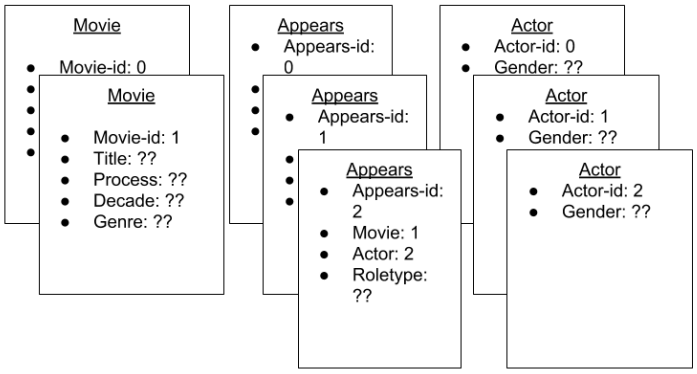


Figure 3: An example PRM skeleton in the movies domain. In this skeleton there are two **Movie** distances, two **Actor** instances, and three **Appears** instances. The references between each object instance is defined, but the attributes of each object instance are not.

2.3 BLOG Models

BLOG is a typed first-order probabilistic programming language that defines probability distributions over varying numbers of objects and varying relations among them [15]. A BLOG model defines a generative process for constructing worlds, which are instantiations of declared random variables.

2.3.1 Syntax

Figure 5 is an example BLOG model. Three types of objects are declared: Ball, Draw, and Color. Each name is an identifier that acts as the symbol representing each type. Lines 5 and 6 then introduce objects guaranteed to exist, namely two Colors and ten Draws. The two declared Colors are given specific names Blue and Green, while the ten declared Draws are distinct but unnamed.

Line 8 is a number statement that adds some number of Balls, drawn from a Poisson distribution, to the world. Note that this implies the potential number of Balls in unbounded.

Lines 10 through 19 are random variable dependency definitions that provide the conditional probability distribution (CPD) for a random variable conditioned on a tuple of arguments. Line 10 defines the variable TrueColor, which for each

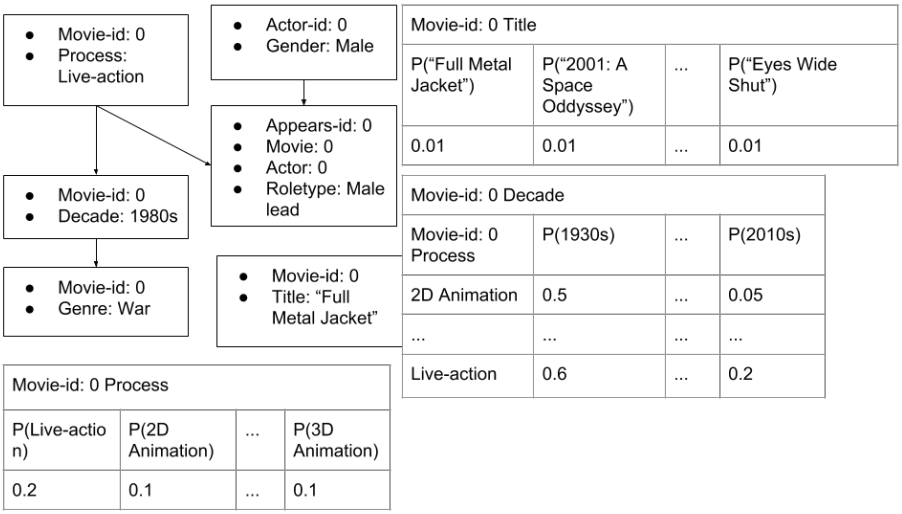


Figure 4: An example of how a set of **Movie**, **Actor**, and **Appears** instances can be expanded into BN form, where each object attribute is its own variable. Each CPT in the PRM has to be replicated to individually describe each created variable. Each shown CPT, and those omitted, would be copied for each object instance in the PRM skeleton.

Ball has value Blue with 50% probability and value Green with 50% probability. Lines 14 through 19 specify exactly how the observed color for a draw depends on the true color of the ball drawn.

Not shown are fixed functions. BLOG allows the user to define deterministic fixed functions which act on the built-in or declared types. These can be used to define mathematical functions not included in BLOG or to represent known information about declared objects.

Lines 21 through 32 specify observe and query statements. These are not part of the model but rather indicate how the user would like to interact with the model. Observe statements specify given variable assignments, called evidence, and query statements specify which variables' posterior probabilities the user would like to infer. Notice that the example query statement does not specify a value, as BLOG will return the posterior probability of all or many possible values of the queried variable. BLOG's inference algorithms are based on random sampling, so if a possible value is not assigned a posterior, its posterior is assumed to be zero.

BLOG's built-in types include `String`, `Integer`, `Boolean`, `Real`, `Character`, `Timestep`, and `RealMatrix` [12]. BLOG has built-in syntax for performing basic functions such as `3+4`. See Figure 6 for an abbreviated example of how a

```
type Ball;
type Draw;
type Color;

distinct Color Blue, Green;
distinct Draw Draw[10];

#Ball ~ Poisson(6);

random Color TrueColor(Ball b) ~ Categorical({Blue -> 0.5, Green -> 0.5});

random Ball BallDrawn(Draw d) ~ UniformChoice({b for Ball b});

random Color ObsColor(Draw d) ~
  if (BallDrawn(d) != null) then
    case TrueColor(BallDrawn(d)) in {
      Blue -> Categorical({Blue -> 0.8, Green -> 0.2}),
      Green -> Categorical({Blue -> 0.2, Green -> 0.8})
    };

obs ObsColor(Draw[0]) = Blue;
obs ObsColor(Draw[1]) = Green;
obs ObsColor(Draw[2]) = Blue;
obs ObsColor(Draw[3]) = Green;
obs ObsColor(Draw[4]) = Blue;
obs ObsColor(Draw[5]) = Green;
obs ObsColor(Draw[6]) = Blue;
obs ObsColor(Draw[7]) = Green;
obs ObsColor(Draw[8]) = Blue;
obs ObsColor(Draw[9]) = Green;

query size({b for Ball b});
```

Figure 5: An example BLOG model describing drawing a ball from an urn ten times with replacement. The number of balls in the urn is unknown and drawn from a Poisson distribution. Each Ball is Blue or Green with equal probability, and has an equal chance of being drawn. The color of each drawn ball is misread with known probability.

PRM could be represented as a BLOG model.

2.3.2 Semantics

Let M refer to a BLOG model. I call the outcome space, that is all possible worlds, Ω. Call the set of objects, of possibly unknown size, in a BLOG model $\mathcal{O}_M = \{O_1, O_2, ...\}$. Each object has a type a, e.g. `String` or `Integer`. The model M also contains a set of variables $\mathcal{V}_M$ defined by random variable dependency definitions, akin to attributes of classes in PRMs. Each dependency definition has a signature $(r, a_1, ..., a_k)$ where r is the return type and a_i is the type of the ith argument.
A BLOG model's random variables are: (i) a number variable for each number statement as well as (ii) a random variable for each possible assignment of arguments to each function random variable dependency definition.

A model M can be said to contain a skeleton σ, which describes the domain-specific language. This includes declared types, as well as the return types and names of declared variable definitions, but not the definition bodies. See Figure 7 for an example.

I also define a structure $\mathcal{S}$, which fills in the bodies of the random variable dependency definitions, and therefore describes the relationships between the variables. This would be like Figure 5 with the question marks filled in. The body of a variable dependency definition can reference other random variables, as in Figure 5, in which `BallDrawn(d)` is referenced within `ObsColor(Draw d)`. I define the set of random variables $\{v_1, ...v_k\}$ called in the body of variable v's dependency definition template to be its parents, or $Pa(v)$.

While BLOG is capable of representing some cyclic models, here I will only consider models whose underlying structure $\mathcal{S}$ is acyclic with regard to the graph created by adding an edge from v_i to v_j if $v_i \in Pa(v_j)$.

Let an instantiation $\mathcal{I}$ be a set of object instances as well as a set of data containing the values of each random variable conditioned on each set of object instances. The first part of an instantiation $\mathcal{I}$ is

$$\mathcal{I}_O = \{\mathbf{o_1}, \mathbf{o_2}, ..., \mathbf{o_n}\},$$

where $\mathbf{o_i}$ is a vector of objects in O_i. The second part is

$$\mathcal{I}_V = \{v(o_1, ..., o_k) \forall v \in \mathcal{V}_M, \{o_1, ..., o_k\} \in [o_i]^k \text{ s.t. } v(o_1, ..., o_k) \text{ is valid},$$

where by "valid" I mean the types of the passed-in objects match the types in the dependency definition signature. Let $\mathcal{I}_{v(o_1,...,o_k)}$ be the value of a random variable conditioned on inputs $\{o_1, ..., o_k\}$ given by the instantiation $\mathcal{I}$.

```
1  type Movie;
2  type Appears;
3  type Actor;
4
5  type Gender;
6  distinct Gender Male, Female, Other;
7
8  type Roletype;
9  distinct Roletype Lead, ComedicRelief, LoveInterest, ...;
10 ...
11
12 distinct Movie movies[30];
13 distinct Appears appears[50];
14 distinct Actor actors[40];
15
16 random String Title(Movie m) ~
17   Categorical({
18     "Moana" -> 0.0333,
19     "Pulp Fiction" -> 0.0333,
20     ...
21   })
22 ;
23 ...
24 random Actor Actor(Appears ap) ~
25   Uniform({ac for ac in actors})
26 ;
27 random Roletype Roletype(Appears ap) ~
28   if Process(Movie(ap)) == 3DAnimation & Gender(Actor(ap)) == Male then
29     Categorical({
30       Lead -> 0.6,
31       ComedicRelief -> 0.2,
32       LoveInterest -> 0.05,
33       ...
34     })
35 ;
36 ...
37 obs Title(Movie(appears[0])) = "Full Metal Jacket";
38 ...
39 query Genre(movies[0]);
```

Figure 6: A PRM represented as a BLOG model. Each class is defined as a type, as well as attributes that exist in a discrete domain. Domains are defined in lines 6 and 9. Class attributes are defined as dependency statements that take a class instance as an argument, as in lines 16 and 27. Assignments from a dataset are incorporated in the form of `obs` statements as in line 37, and specific variables can be queried as in line 39.

```
1  type Ball;
2  type Draw;
3  type Color;
4
5  distinct Color, Blue, Green;
6  distinct Draw Draw[10];
7
8  #Ball ~ ???
9
10 random Color TrueColor(Ball b) ~ ???;
11 random Ball BallDrawn(Draw d) ~ ???;
12 random Color ObsColor(Draw d) ~ ???;
```

Figure 7: An example skeleton for the urn-ball program listed in Figure 5.

2.3.3 Performing Inference in BLOG

There currently exist two compilers for BLOG, one that is Java-based, and one called Swift [23] which compiles BLOG code into a C++ executable. BLOG can perform inference via three algorithms: rejection sampling, likelihood weighting, and Metropolis-Hastings. Here I will not alter any of these methods and but instead treat them as block-box algorithms to evaluate candidate BLOG models.

3 Related Work

3.1 Learning Bayesian Networks

Learning Bayesian networks is the task of estimating a network structure and parameters given a dataset while minimizing complexity. Bayesian network learning tasks can be partitioned into four categories [1]. The first case is when the graph structure G is known and the dataset is complete and fully observable, meaning the values of every variable in the graph are known, so only the parameters θ need be learned. Define the dataset as $\Sigma = \{\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_m}\}$, where a vector $\mathbf{x_i}$ is one of m assignments to every variable in G, $\mathbf{x_i} = \{x_{i1}, x_{i2}, ..., x_{in}\}$.

First the log of the likelihood function is taken resulting in

$$\log L(\theta|\Sigma) = \sum_{j=1}^{m} \sum_{i=1}^{n} \log P(x_{ji}|\pi_i, \theta_i).$$

Now the scoring function is decomposed such that each term can be maximized individually by simply counting the instances of x_{ji} in Σ and normalizing.

The second case is where the graph structure G is known, but the dataset

Σ is missing values or is not fully observed. One technique to tackle this is the expectation–maximization [5] algorithm which alternately optimizes the parameters θ and estimates of the missing data.

In the third case, there is a complete and fully observable dataset but an undetermined graph. The problem of finding the best graph G is NP-hard, as the number of possible DAGs is superexponential in the number of variables [1]. Here we need to introduce a prior probability over the model space to the scoring function to avoid simply connecting every variable. One such prior scales linearly with the number of connections in the model, thereby preferring simpler, less connected. One approach to learning in this case is to begin with a graph with no edges, such that every variable is absolutely independent of others, then do a local search by adding or removing edges until a local optimum is found [8]. One convenience in this approach is that, as the score is decomposable as above, if an arc is added from X_i to X_j, only $\log P(X_j|\pi_j, \theta_j)$ and θ_j need be recomputed. To avoid getting stuck in local maxima, one can augment this algorithm with random restarts or simulated annealing.

In the fourth and final case, there is both an incomplete dataset and an undetermined graph. The standard approch is to substitute the Bayesian score used above with an approximation to the posterior known as the Bayesian information criterion (BIC). Similar to the Bayesian score, the BIC trades off between maximizing likelihood and minimizing model complexity. Another approach is to apply EM, but in the Maximization step, perform a local search step across graph structures. This is known as Structural EM [10].

None of the presented algorithms are able to introduce hidden variables, which could reduce model complexity by reducing the number of dependencies without affecting likelihood. These learning algorithms therefore do not search the complete space of BNs able to generatively produce a given dataset.

3.2 Learning PRMs

The goal of learning PRMs is to find a structure that best describes the data in an instantiation, given the instantiation and a skeleton. The local search algorithm in PRM space, similar to learning Bayesian networks, uses the posterior probability of a model given the data as the score, and iteratively improves each model using small local changes [4]. Namely, this score is

$$P(\mathcal{S}, \sigma_r|\mathcal{I}) \propto P(\mathcal{I}|\sigma_r, \mathcal{S})P(\mathcal{S}|\sigma_r).$$

The prior probability $P(\mathcal{S}|\sigma_r)$ is a prior probability over structures. It is often assumed the structure is independent of the skeleton, so $P(\mathcal{S}|\sigma_r) = P(\mathcal{S})$. The term $\log P(\mathcal{S})$ is then typically made to be proportional to the sum of the lengths of slot chains contained in S. This punishes relationships between objects that are only very indirectly related, as well as dense networks making use

of short chains of length one or two.

The other term is the marginal likelihood

$$P(\mathcal{I}|\sigma_r, \mathcal{S}) = \int P(\mathcal{I}|\sigma_r, \mathcal{S}, \theta_{\mathcal{S}}) P(\theta_{\mathcal{S}}|\mathcal{S}) d\theta_{\mathcal{S}}.$$

Here the parameters $\theta_{\mathcal{S}}$ are typically drawn from a parameter-independent Dirichlet prior. See [4] for details.

The local search algorithm for PRMs is similar to local search over Bayesian networks. The null model starts with every class attribute being independent, which is the first candidate model. Small local changes are then applied to the candidate model, maximizing the score until a local optimum is reached. These local changes are adding an edge, deleting an edge, or changing the direction of an edge. As with search over Bayesian networks, this can be augmented with random restarts or simulated annealing to overcome local maxima, or with Structural EM when the data are not fully observed. The log-likelihood is decomposable into a sum of terms, such that each local change only necessitates the recomputing of terms corresponding attributes who lost or gained a parent.

3.3 Learning Probabilistic Programs

In general, deterministic program synthesis is the task of creating a program that satisfies user intent [6]. The standard way of expressing this intent is through logical specifications. Other ways this intent can be expressed are in the form of input-output examples or natural language. This intent can be paired with a template program to restrict the search algorithm's search space. As with PRM search, search over the space of programs in some language, even with a provided template, still is searching over a number of programs exponential in the allowed program size. Recent advancements in program synthesis usually involve exploiting domain-specific knowledge to reduce the search space.

Learning probabilistic programs differs from learning deterministic programs in that probabilistic programs specify distributions over values, rather than deterministic outputs. One approach to synthesizing probabilistic programs is detailed in [17]. Nori et al. use Metropolis–Hastings to fill in holes in a provided program skeleton S with code H with the aim of generatively modeling a dataset D. Let $S[H]$ refer to the program skeleton S with holes completed by code blocks H. A state used in in this MCMC algorithm is a tuple H of possible hole completions. The next proposed tuple of hole completions H' is obtained by applying random changes to H, such as swapping out one variable for another or replacing a constant c with the result of sampling from a normal distribution centered around c. Instead of directly computing $P(D|S[H])$, which is computationally expensive, they approximate difficult to integrate CPDs with mixtures of Gaussians (MoGs). MoGs have a universal approximation property [13], which makes them suitable for this kind of substitution. This substitution

allows for computation of $P(D|S[H])$ in linear time.

Another approach is given in [18]. Similar to the previous approach, Perov et al. use a MCMC algorithm to create a program that generates a given set of data. Let $\mathcal{T}$ refer to some program text, $\mathcal{X}$ refer to the given data, and $\hat{\mathcal{X}}$ refer to samples of data generated by running $\mathcal{T}$ some number of times. Let $\pi(\mathcal{X}|\hat{\mathcal{X}})$ be a distance metric between summary statistics of $\mathcal{X}$ and $\hat{\mathcal{X}}$. Their goal is to optimize

$$\pi(\mathcal{X}|\hat{\mathcal{X}})p(\hat{\mathcal{X}}|\mathcal{T})p(\mathcal{T}).$$

The first term measures similarity between given and generated data, the second term measures the probability of model $\mathcal{T}$ generating the data it did, and the third term is a prior over program texts.

4 Learning BLOG Models

Learning BLOG models is the task of synthesizing a BLOG model on some domain that generates a dataset with high probability while minimizing model complexity. To learn BLOG models, I adapt the local search algorithm used to learn PRMs.

4.1 Scoring

The score of a BLOG model, as when evaluating PRMs, is the posterior probability of the model given the data $P(M|\mathcal{I}) = P(\mathcal{S}, \sigma|\mathcal{I})$. By Bayes rule, this score can be written as the probability of the data given the model and the prior probability of the model as

$$P(\mathcal{S}, \sigma|\mathcal{I}) \propto P(\mathcal{I}|\mathcal{S}, \sigma)P(\mathcal{S}|\sigma).$$

The first term is the probability of a model generating the data given by instantiation $\mathcal{I}$, which I can approximate using BLOG's inference algorithms. The second term is the probability of some structure given the skeleton. As with PRMs, I will assume these are independent, so $P(\mathcal{S}|\sigma) = P(\mathcal{S})$. I would like this prior to punish complexity to avoid overfitting, so I set it to be proportional to the sum over all random variables of the number of random variables each depends on.

The term $P(\mathcal{I}|\mathcal{S}, \sigma)$ regarding a model M can be decomposed as

$$P(\mathcal{I}|\mathcal{S}, \sigma) = \prod_{v \in \mathcal{V}_M} \prod_{\{o_1,...,o_k\}} P\Big(v(o_1, ..., o_k) = \mathcal{I}_{v(o_1,...,o_k)} | \{\mathcal{I}_{v'(o_1,...,o_k)}\}\forall v' \in Pa(v)\Big).$$

Algorithm 1: BLOG model search

```
input  : Instantiation 𝓘, Structure 𝒮, Model M
output : A locally optimal BLOG model
training set 𝓘_T, validation set 𝓘_V ← split(𝓘);
current model M_0 ← M;
best score ← score(M_0, 𝓘_V);
best new model M_+ ← M_0;
while score is improving do
    training subset 𝓘_T' ← random subset of 𝓘_T;
    for each variable v ∈ V_𝒮 do
        for each node n ∈ G_v do
            for each type of local change c do
                candidate model M' ← incorporate c into M_0 via node n;
                if c has a parameter then
                    optimize c's parameter in M' on 𝓘_T';
                end
                if score(M', 𝓘_𝒱) > best score then
                    M_+ ← M';
                    best score ← score(M', 𝓘_𝒱);
                end
            end
        end
    end
    M_0 ← M_+
end
return M_0;
```

This is the probability of model $M = (\mathcal{S}, \sigma)$ generating each piece of data in $\mathcal{I}$. As in Bayesian networks and PRMs, the log of the score is a decomposable sum.

It is not guaranteed that a valid assignment to a variable is assigned a non-zero probability by a BLOG model. As a result, if a single variable assignment in $\mathcal{I}$ cannot be generated by a BLOG model, the entire score becomes 0, or $-\infty$ under the log-likelihood. This is not ideal, since I would prefer a model that assigns non-zero probability to fewer answers to a model that is zero everywhere. To deal with this, I soften the zeros by replacing them in the probability calculations with a small ϵ. In this way, models for which 0 probabilities occur less often, all else being equal, score over models in which they occur more often.

4.2 The Search Algorithm

See Algorithm 1. It takes as input (i) an instantiation I, (ii) a skeleton, or program template, $\mathcal{S}$, and (iii) a base model M. The algorithm's search begins at the provided model M. For dependency definitions with no provided bodies,

```
random String Process(String input) ~
 Categorical({
        Capitalize(input) -> 0.5,
        Reverse(input) -> 0.5
 })
;
```

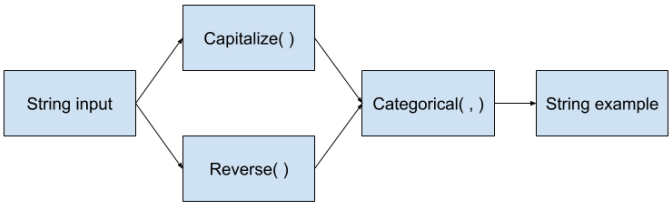


Figure 8: A dependency definition and its corresponding graph.

the initial null model assumes no dependencies on any other variables. For dependency definitions with a return type that is a user-defined discrete and finite domain, it uses a Categorical distribution which is uniform over the domain. For Integers, it uses a UniformInt distribution with the lower end being the smallest value in I and the upper end being the largest in I. For Strings, it models them as being generated from unigram model `LetterUnigram` where each character is equally likely to be appended to the returned string until stopping.

The space of conditional distributions used to construct each dependency definition includes those pre-defined in BLOG, such as $UniformInt$(int a, int b), as well as user-defined Java distributions and functions, such as `RemoveLetter(String s)` and `Abbreviate(String s, Integer i)`. Note that user-defined functions, unlike distributions, are entirely deterministic. These conditional distributions and functions are categorized by their type signature, or the input types of the variables they are conditioned upon and their output type. For example, $UniformInt$ is categorized as having two integer dependencies and one integer output, and *RemoveLetter* is categorized as having one String dependency and one String output.

4.2.1 Dependency Definition Body Representation

The conditional distribution that makes up a random variable v's dependency definition is stored in a directed graph where the random variables upon which v is conditioned are source nodes, and the value of the v is the only sink node. Each node is either a variable conditioned upon, v itself, a deterministic function,

```
1  random String NoisyCite(Listing l) ~
2    DeleteFirstLetter(LastName(l))
3  ;
```

(a) Before

```
1  random String NoisyCite(Listing l) ~
2    DeleteLastLetter(DeleteFirstLetter(LastName(l)))
3  ;
```

(b) After

Figure 9: A demonstration of the second type of local change. In part (a), the path of functions connecting `LastName(l)` to `NoisyCite(l)` is solely the `DeleteFirstLetter` function. In part (b), the `DeleteFirstLetter` function is composed with the `DeleteLastLetter` function.

or some conditional distribution. There is an arc (i, j) from one node to another if node j depends on node i. See Figure 8 for a sample dependency definition and its function graph. I refer to this graph for some variable v as G_v.

4.2.2 Local Changes

The search space is the subset of BLOG models that can be constructed starting from the starting model with any number of the following changes applied.

Change 1: Replace a fixed function or distribution with a different fixed function or distribution. For this type of change, only functions and distributions with the same type signature as the original function/distribution are considered. See Figure 9 for an example.

Change 2: Insert a fixed function or distribution. Similarly, for the second type of change, only functions and distribution whose input(s) match the outputs of its proposed parents, if any, and whose output match the inputs of the proposed children are considered. See Figure 10.

Change 3: Introduce a dependency on another variable. For the third type of change, the new variable is introduced by replacing a distribution or function with a new distribution or function that takes exactly one more input. This additional input slot is taken by the new variable. See Figure 11.

Change 4: Remove a dependency on another variable. For the fourth type of change, the variable is removed if and only if one of its descendants is a function or distribution that takes more than one input. The nodes between the

```
1  random String NoisyCite(Listing l) ~
2    DeleteFirstLetter(LastName(l))
3  ;
```

(a) Before

```
1  random String NoisyCite(Listing l) ~
2    DeleteLastLetter(LastName(l))
3  ;
```

(b) After

Figure 10: A demonstration of the first type of local change. In part (a), the path of functions connecting `LastName(l)` to `NoisyCite(l)` is solely the `DeleteFirstLetter` function. In part (b), the path is replaced by the `DeleteLastLetter` function.

```
1  random String NoisyCite(Listing l) ~
2    LastName(l)
3  ;
```

(a) Before

```
1  random String NoisyCite(Listing l) ~
2    Concat(FirstName(l), LastName(l))
3  ;
```

(b) After

Figure 11: A demonstration of adding the random variable `FirstName(l)` as a parent to variable `NoisyCite(l)`.

```
1  random String NoisyCite(Listing l) ~
2    Concat(FirstName(l), LastName(l))
3  ;
```

(a) Before

```
1  random String NoisyCite(Listing l) ~
2    StringIdentity(FirstName(l))
3  ;
```

(b) After

Figure 12: A demonstration removing the random variable `LastName(l)` as input to the CPD for `NoisyCite(l)`. Here the `Concat` function is replaced by the `StringIdentity` function.

```
random String NoisyCite(Listing l) ~
  LastName(l)
;
```

(a) Before

```
random String NoisyCite(Listing l) ~
  Categorical({
        LastName(l) -> 0.2,
        Concat(FirstName(l), LastName(l)) -> 0.8
  })
;
```

(b) After

Figure 13: Suppose our current model is simply the CPD described in part (a). Suppose also `Concat(FirstName(l), LastName(l)` perfectly describes 80% of our training batch and `LastName(l)` the other 20%. The algorithm will then consider this new CPD and combine it as in part (b) to describe the entire training batch.

removed variable and this descendant are removed and the multi-input node is replaced with a function or distribution that takes one fewer input. See Figure 12.

When a change has been incorporated, it is possible that the altered CPD explains some data instances better than the original CPD for some training examples, but not others. To handle this, the algorithm uses a Categorical distribution that selects between the new and old CPDs. The weights of the CPDs in this Categorical distribution are determined by the proportion of training examples a CPD gives a better score. See Figure 13 for an example.

4.2.3 Parameter Optimization

Some distributions and functions have parameters that can be optimized. For example, a class of Abbreviate functions can be defined as all functions $f(s; k) = s[: k]$ in which a string s is abbreviated to its first k characters. In general, such a k is optimized over the training subset. In the case of distributions, the k is chosen to maximize the likelihood over the training subset. In the case of fixed functions, two models are scored on the validation set. See Figure 14 for an example of these two cases.

The first case is where the k is simply selected from a Categorical distribution over the values of k maximizing over each example in the training subset. In the Abbreviate example, how each word is abbreviated does not depend on

```
random String Abbreviate(String s) ~
  Categorical({
        Abbreviate(s, 1) -> 0.4,
        Abbreviate(s, 4) -> 0.2,
        Abbreviate(s, 6) -> 0.1,
        s -> 0.3
  }
;
```

(a) Without value-specific distributions.

```
random String Abbreviate(String s) ~
  if s == "Journal" then
        Categorical({
              Abbreviate(s, 1) -> 1.0
        })
  else if s == "Applied" then
     Categorical({
              Abbreviate(s, 4) -> 0.7.
              s -> 0.3
     })
    ...
  Categorical({
        Abbreviate(s, 1) -> 0.4,
        Abbreviate(s, 4) -> 0.2,
        Abbreviate(s, 6) -> 0.1,
        s -> 0.3
  })
;
```

(b) With value-specific distributions

Figure 14: Two models that are tested with the introduction of `Abbreviate`, an optimizable function.

the word itself, only the distribution of words seen in the training subset. The second case is where the Categorical distribution associates input values with a distribution of values for k. If the word "Journal" is seen as always abbreviated to "J" in the training subset, this model will only consider abbreviating to the first letter when "Journal" is seen. In this case each unique input value in the training set has its own Categorical distribution. For unseen input values, the function is drawn from a Categorical distribution which is an average over the observed input value's distributions, similar to the first case.

4.2.4 Model Space

Having examined the possible local changes, we can define the model space being explored. This space is all BLOG models whose variable dependency definitions can be represented as dependency graphs (i) composed of functions and distributions built into BLOG or included by the user and where (ii) each node has in-degree at most two.

I prove by induction Algorithm 1 can fully explore this space. For the base case, the algorithm begins with either a user-provided model or a null model with a simple dependency definition for each random variable template.

For the inductive step, consider any model satisfying the conditions above. We can consider models with the same general graph structure but different nodes as the same model because Change 1 guarantees we will be able to explore any of these models from an equivalent model. Then consider some model with any node in any variable template's dependency graph removed. It suffices to show that one of the considered local changes undoes this removal. If a node is removed from a composition of functions or distributions, this can be undone with Change 2. If a dependency on another variable is removed, this can be undone with Change 3.

4.2.5 Model Evaluation

The algorithm chooses between two methods for evaluating models, depending on the model to be evaluted. The first method is BLOG's built-in Metropolis–Hastings algorithm. I use this over likelihood weighting as MH is more efficient for models with large numbers of variables. I use the Java BLOG backend over Swift since Swift takes more time just to compile than it takes Java BLOG to compile and run once. Although the runtime of Swift's compiled executable is very fast, because the algorithm only evaluates each BLOG model once, it makes more sense to use Java.

To evaluate the probability of each instance of a variable in an instantiation $\mathcal{I}$ being assigned its value in $\mathcal{I}$, $P(v(o_1, ..., o_k) = I_{v(o_1,...,o_k)} | I_{v'(o_1,...,o_k)} \forall v' \in Pa(v))$, the algorithm must set values for each of the variable parents using `obs` statements. A result of this is that, in the same BLOG run, it cannot evaluate

probabilities of any variable in $Pa(v)$, as BLOG cannot provide a probability distribution for an evidence variable set to be a constant value in all generated worlds.

As I am limiting the search space to acyclic BLOG models, there must exist at least one topological sorting of variables. Each model is therefore evaluated in stages, beginning by evaluating variables with no parents, and continuing with variables whose parents have already been evaluated. This goes on until all variables have been evaluated.

The second method is exactly calculating the score without using any kind of sampling. This is specific to string domains and is used when the model contains at least one `LetterUnigram` distribution. Suppose I am calculating the probability of a variable of type String taking on the value "schapire" and this variable's CPD is defined as a `LetterUnigram` distribution. There is only one value among all the strings that can be sampled from this distribution that matches the given value, so any kind of sampling algorithm will require very many samples to be able to approximate the desired probability.

This exact calculation is done by processing each node in the graph in a topologically sorted order and enumerating each's possible values and their respective probabilities. In the case of a string domain, this is done for all nodes excepting any LetterUnigrams, as they have infinite possible outputs. In such cases the CPD is evaluated as a list of regular expressions where `LetterUnigram`s are represented as `(.*)`. If the expression matches the variable's given value, then there is some string that can have been sampled from `LetterUnigram` that produces the desired value.

5 Experimental Evaluation: Citations

Table 1: Example citations. The first example is a standard citation pattern, having the first two names abbreviated followed by the last name. Notice there is no space between the initials. The hyphen in the second example could be due to the last name being split across lines. The third example shows an author likely accidentally switching the first and last names before abbreviating.

FirstName	MiddleInit	LastName	CitedName	FirstWord	SecondWord	...	FifthWord	CitedJournal
robert	e	schapire	r.e. schapire	International	Journal	...	Engineering	Int. J. Electr. Eng.
david		haussler	d. haus-sler	Irrigation	And	...		Irrig. Drain.
harris		drucker	harris, d.	F1000Research		...		F1000Research

Citation matching is a commonly used problem that demonstrates BLOG's capabilities. The specific task is to reconstruct the original set of publications being cited and identity the referenced authors given a set of citations[11].

To test Algorithm 1, I used two subtasks of generatively modeling both how authors and journals are cited in papers, which can be part of a larger model that solves the full citation problem. I used the cited authors in the CORA dataset [14], a set of 1295 citation instances with common fields such as authors, journal, year, etc. labeled. These citations contain the original mistakes they were printed with, such as misspellings. They are also all entirely lowercase. I augmented CORA with each author's true first names, middle initials, and last names. These were programmatically extracted and manually adjusted from the dataset. The result is a complete dataset containing 4940 tuples of the form (`FirstName`, `MiddleInitial`, `LastName`, `CitedName`). For authors with no middle name, I set `MiddleInitial` to be the empty string.

For journals, I used the listed journals and abbreviations provided by [22]. Unlike CORA, this list is regularly updated with new journal names. Also, each journal is associated with exactly one abbreviation and is unlikely to contain a misspelling. The dataset contains some journal names in English and some in German. Lastly, likely because this list is curated by a single organization, each word for the most part is abbreviated the same way across different journal abbreviations. Typically, a journal in this dataset is cited as taking an abbreviation of each word and concatenating with a space. In a few cases, the journal is abbreviated as an acronym. Here I only considered journal names with five or fewer words to reduce the complexity of the problem and limit time needed to search. For journals with fewer than five words, unused slots are set as the empty string.

These journal names were combined with the CORA dataset by replacing the journal each citation was actually published in with an abbreviation from the abridged journals dataset. Each citation was also edited to list exactly one author. See Table 1 for some example entries and how these citations were fed as input. Other citation attributes not shown are the title and date.

The input provided to the algorithm took the form of a template BLOG model, a dataset, and a set of descriptors of the columns of data in the dataset. My template BLOG model uses a basic letter unigram model for each variable except for `CitedJournal`, where we start with a CPD that concatenates the FirstWord with a space and a letter unigram distribution. The dataset resembled Table 1 but with 2000 generated entries. The descriptors file was a csv file that lists each kind of declared type in our domain and which types each dependency definition template takes as arguments. See Table 2 for the descriptors file I used for this domain. The information in these auxiliary files also could have been included as parts of the original BLOG model input.

In addition, I also defined some user-defined String manipulation conditional distributions and functions detailed in Table 3 in the Appendix.

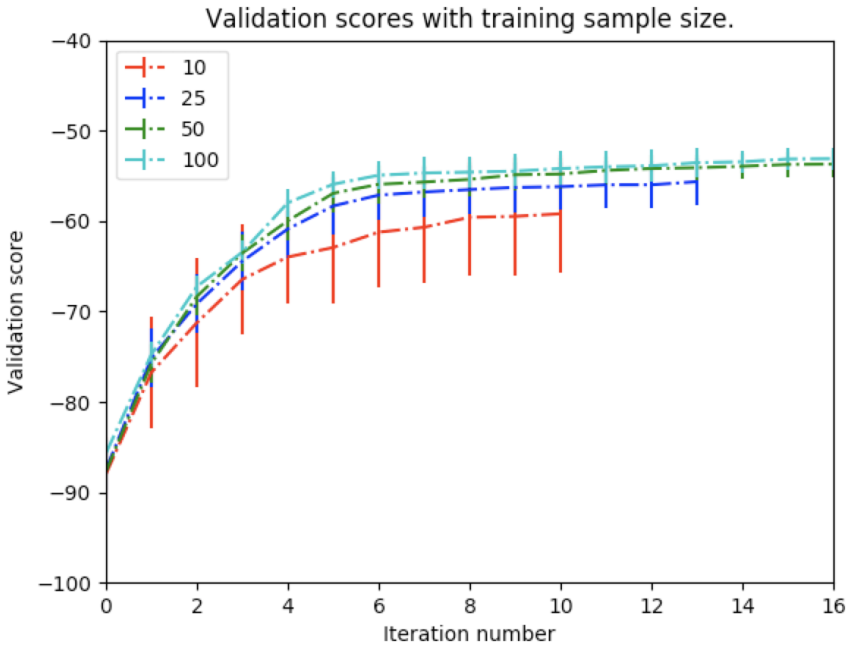


Figure 15: Model scores with size of data. These are the averaged results of five runs. Each line is labeled with the size of the training data subset used to optimize models. The score is the average log-likelihood of each data point in the subset. When exposed to more examples, the system is more likely to encounter a wider variety of author citation methods and word-specific abbreviations.

Table 2: The descriptors file for our citations application. Here I inform the learner of the existence of `FirstName`, `MiddleInitial`, and `LastName` but I do not ask it to fill in the dependency definition bodies beyond the default implementations, nor do I ask it to score their posterior probabilities. I do, however, ask it to fill in the dependency definition body for `CitedName` and ask its posterior to be included in the score.

Variable	Type	Should Fill	Scored
FirstName	String	False	False
MiddleInitial	String	False	False
LastName	String	False	False
CitedName	String	True	True
FirstWord	String	False	False
...			
FifthWord	String	False	False
CitedJournal	String	True	True
Date	String	False	False
Title	String	False	False

5.1 Results

I evaluated the algorithm using four different training batch sizes and a validation set size of 400. See Figure 15 for the system's performance. As expected, the scores were best when the model was allowed to evaluate candidate models on the largest amounts of data. When exposed to more entries, the system can learn more citation formats for names and journals as well as how more words are abbreviated in the journals case. The general shape of the curves shows that initially the system is able to make relatively large gains in probability. This is a result of being able to explain large parts of the citation string, such as each author's `LastName`. In later iterations, only one or two character gains are made, such as initials and punctuation, resulting in smaller score increases. See Table 5 in the Appendix to see how the system progressively increases its understanding of a given citation example.

The system is able to capture all the author citation methods described in Milch's solution [15]. See Figure 16 for exactly what rules the best-scoring model contained.

The system is also able to capture the general citation method for the given journals, that is possibly abbreviating each word then concatenating with a space. Milch's solution only handles authors and citations so we cannot directly compare these results with his solution. See Figure 17 in the Appendix for how a CPD appears as BLOG code. The learned model contains a table of abbreviations for each word come across during training. It was able to learn that some words, such as "and" or "of" are always omitted. As part of the dataset is in

German or Italian, this was also true for foreign prepositions like “di” and “für”.

As mentioned in Figure 16, some rules still contained `LetterUnigram` distributions. This means the system was not able to fully describe each example. Some of the unexplained examples include those with spelling errors. While the model space searched by Algorithm 1 includes models that can generate text with these mistakes, such as those using a `DeleteLetter`, `AddLetter`, or `ReplaceLetter` distribution, the algorithm is not able to generalize a single spelling mistake. For example, suppose in some iteration the system has an existing rule `Abbr(FirstName)+'. '+LastName`. Say it tries to include an `AddLetter` distribution applied to `LastName` and the training example batch happens to include example `d. haus-sler`. This new rule `Abbr(FirstName)+'. '+AddLetter(LastName)` explains this example better than the old rule. However, while we would expect a last name could be misspelled regardless of the context, the algorithm has only learned that a specific kind of error can occur in the specific format of `Abbr(FirstName)+'. '+LastName`.

Similarly, the tables of abbreviations learned for each word in the corpus is distinct for each word slot. `FirstWord` has a different set of abbreviations than `SecondWord`, and so on. This sort of separation of distributions makes sense in the case of authors, where `FirstName` is often abbreviated but `LastName` never is, but not in the domain of journals. To be able to find a model where each word slot shares a table of abbreviations would require an additional kind of local change that allows for shared groups of nodes within distinct CPDs.

6 Conclusion

Taking inspiration from methods of learning Bayesian networks and PRMs from data, I have provided an algorithm for synthesizing BLOG models to fit a given dataset using built-in and user-defined functions and conditional distributions. I have demonstrated the use of Algorithm 1 on a simplified version of the citations problem was able to learn compositions of functions and conditional distributions that mirror common citation methods. In doing so I have also identified key areas for potential improvement.

6.1 Future Work

Iterating through every applicable distribution is quite naive. Ideally one could implement a way to limit the pool of possible functions or distributions based on the input and output for which we are trying to find a relation. For example, if one has the input “robert” and output “r”, one would like to consider the `Abbreviate` function sooner rather than later. One possible solution is to have a set of recognizers, which are easier to compute than the functions/distributions

CitedName CPD Description	Prob
`LastName`	0.34
`FirstName LastName`	0.1
`Abbr(FirstName). LastName`	0.14
`FirstName MiddleInitial. LastName`	0.12
`Abbr(FirstName). MiddleInitial. LastName`	0.08
`LastName, FirstName`	0.08
`LastName, FirstName MiddleInitial.`	0.02
`LastName, Abbr(FirstName). MiddleInitial.`	0.02
`LastName, Abbr(FirstName).MiddleInitial.`	0.04

(a) Citation formats described for `CitedName`.

CitedJournal CPD Description	Prob
`Abbr(FirstWord).`	0.14
`Abbr(FirstWord). Abbr(SecondWord).`	0.16
`Abbr(FirstWord). Abbr(SecondWord). Abbr(ThirdWord).`	0.27
`... Abbr(FourthWord).`	0.29
`... Abbr(FourthWord). Abbr(FifthWord).`	0.14

(b) Citation formats described for `CitedJournal`.

Figure 16: Most of the possible citation formats present in the highest-scoring model produced. Not explicitly written is that each `Abbr` is a `Categorical` distribution selecting to what index to abbreviate. Also, each instance of punctuation or whitespace, such as "." or " " in most cases is also a `Categorical` distribution. Omitted are additional rules that contain a `LetterUnigram` distribution.

themselves, but can rule out functions as candidates before they are ever computed on the input. For example, a Length recognizer could be used to partition the function/distribution space into those that create an output shorter than the input and those that do not. Applying this recognizer to our input/output example could rule out candidates such as `AddLetter` and `ReplaceLetter`, which do not shorten the input, before ever evaluating them.

Functions and distributions necessary for the citation problem, see Table 3, were included as input to the system, but had any of these not been provided, many derived citation methods could not have been found. Ideally, a system could invent new distributions to describe relationships between data. This is analogous to predicate invention in inductive logic programming [16].

Here we only explore a subset of the space of possible BLOG models. The next logical extension would be to provide for the creation of hidden variables to reduce model complexity. For example, regarding the task of citing journal names, a Boolean hidden variable that corresponds to whether or not the `FirstWord` is abbreviated could inform the `Categorical` CPD that corresponds to what punctuation follows the first word. Currently, whether or not the `FirstWord` is abbreviated and the punctuation following have no connection.

Throughout this report I have assumed complete datasets, but often this is not always a realistic assumption to make. One could adapt the Structural EM algorithm found in [5] to make use of the best model found so far to impute the missing data. To do this, one could at the start of the algorithm use an MCMC algorithm where each state computed is an assignment to the missing values in the dataset as the expectation step in EM. Then in the maximization step, one could use these values when searching over models adjacent in the search space.

References

[1] I. Ben-Gal *Bayesian Networks* In Encyclopedia of Statistics in Quality and Reliability, Wiley and Sons, 2007.

[2] D. M. Chickering. *Learning Bayesian networks is NP-complete.* In Learning from Data: Artificial Intelligence and Statistics V. Springer Verlag, 1996.

[3] J. Fodor *The language of thought.* Cambridge, MA: Harvard University Press, 1975.

[4] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. *Learning Probabilistic Relational Models.* In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, 1999.

[5] T. L. Griffiths and A. Yuille. *A primer on probabilistic inference.* In Trends in Cognitive Sciences, 2006.

[6] S. Gulwani, O. Polozov, and R. Singh. *Program Synthesis* Foundations and Trends in Programming Languages, vol. 4, no. 1-2, pp. 1-119, 2017.

[7] W. Hastings *Monte Carlo Sampling Methods Using Markov Chains and Their Applications* In Biometrika, vol. 57, no. 1, pages 97—109, 1970.

[8] D. Heckerman. *A tutorial on learning with Bayesian networks*. In M. I. Jordan, editor, Learning in Graphical Models. MIT Press, Cambridge, MA, 1998.

[9] D. Koller and A. Pfeffer. *Probabilistic frame-based systems*. In Proceedings of the Fifteenth Conference of the American Association for Artificial Intelligence, pages 580–587, Madison, Wisconsin, 1998.

[10] D. Koller and N. Friedman. *Probabilistic Graphic Models*. Cambridge: The MIT Press, 920—925, 2009.

[11] S. Lawrence, C. L. Giles, and K. D. Bollacker. *Autonomous citation matching*. In Proc. 3rd Intl Conf. on Autonomous Agents, pages 392393, 1999.

[12] L. Lei, S. Russell. *The BLOG Language Reference*. BLOG Programming Language, 2014.

[13] V. Mazya and G. Schmidt. *On approximate approximations using gaussian kernels*. IMA Journal of Numerical Analysis, 16, pages 13–29, 1996.

[14] A. McCallum, K. Nigam, J. Rennie, and K. Seymore. *Automating the construction of internet portals with machine learning*. Information Retrieval, 3, 127–163, 2000.

[15] B. Milch *Probabilistic Models with Unknown Objects*. PhD thesis, Computer Sccience Division, University of California, Berkeley, 2006.

[16] S. Muggleton and W. Buntine. *Machine invention of first-order predicates by inverting resolution*. In Proc. 5th Intl Conf. on Machine Learning, pages 339352, 1988.

[17] A. V. Nori, S. Ozair, S. K. Rajamani, and D. Vijaykeerthy. *Efficient synthesis of probabilistic programs*. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015, pages 208–217, 2015.

[18] Y. Perov and F. Wood. *Learning Probabilistic Programs*. arXiv e-print arXiv:1407.2646, 2014.

[19] S. Piantadosi *LOTlib: Learning and Inference in the Language of Thought*. Available from https://github.com/piantado/LOTlib, 2014.

[20] S. Piantadosi, J. Tenenbaum, and N. Goodman. *Bootstrapping in a language of thought: A formal model of numerical concept learning*. In Cognition 123(2), pages 199-217, 2012.

[21] S. Russell. *Artificial Intelligence: a Modern Approach*. Upper Saddle River, N.J. :Prentice Hall, 2010.

[22] *Science and Engineering Journal Abbreviations* Woodward Library, The University of British Columbia, accessed June 2018.

[23] Y. Wu, L. Li, S. Russell, and R. Bodik. *Swift: Compiled Inference for Probabilistic Programming Languages* In Proceedings of IJCAI-16, 2016.

7 Appendix

Table 3: User-defined functions and distributions used in the citations domain.

Name	Inputs	Parameter	Output	Description
Identity	String	None	String	Returns the input String unchanged.
Abbreviate	String	Integer	String	Returns the first [Integer] letters of the input String.
Concat	Strings	None	String	Concats the input Strings.
LetterUnigram	None	None	String	Generates a String where each character, or the decision to stop, is drawn uniformly at random.
ReplaceLetter	String	None	String	Uniformly at random replaces a character with a different character, also selected uniformly at random.
DeleteLetter	String	None	String	Uniformly at random deletes a character.
AddLetter	String	None	String	Uniformly at random inserts a new character.
Constant	None	String	String	Returns the String parameter.

```
random String CitedJournalHidden0_0(Citation c) ~
        if FirstWord(c) == "berichte" then
                Categorical({
                        Abbreviate(FirstWord(c), 3) -> 1.0
                })
        else if FirstWord(c) == "forensic" then
                Categorical({
                        FirstWord(c) -> 1.0
                })
        else if FirstWord(c) == "methods" then
                Categorical({
                        FirstWord(c) -> 1.0
                })
        ...
        else Categorical({
                        FirstWord(c) -> 0.201183431953,
                        Abbreviate(FirstWord(c), 7) -> 0.0532544378698,
                        Abbreviate(FirstWord(c), 6) -> 0.0236686390533,
                        Abbreviate(FirstWord(c), 5) -> 0.0828402366864,
                        ...
            })
;
random String CitedJournalHidden1_0(Citation c) ~
    Categorical({
            ". " -> 0.79881656804,
            " " -> 0.201183431953
    })
;
random String CitedJournalHidden2_0(Citation c) ~
        if SecondWord(c) == "and" then
                Categorical({
                        Abbreviate(SecondWord(c), 0) -> 1.0
                })
        else if SecondWord(c) == "ions" then
                Categorical({
                        SecondWord(c) -> 1.0
                })
        else if SecondWord(c) == "biologia" then
                Categorical({
                        Abbreviate(SecondWord(c), 4) -> 1.0
                })
        ...
        else Categorical({
                        Abbreviate(SecondWord(c), 10) -> 0.0350877192982,
                        SecondWord(c) -> 0.140350877193,
                        Abbreviate(SecondWord(c), 3) -> 0.166666666667,
                        Abbreviate(SecondWord(c), 0) -> 0.114035087719,
                        ...
                })
;
random String CitedJournalHidden3_0(Citation c) ~
    Categorical({
            ``." -> 0.8596491228,
            "" -> 0.140350877193
    })
;
random String CitedJournalHidden4_0(Citation c) ~
        Concat(CitedJournalHidden2_0(c), CitedJournalHidden3_0(c))
;
random String CitedJournalHidden5_0(Citation c) ~
        Concat(CitedJournalHidden1_0(c), CitedJournalHidden4_0(c))
;
random String CitedJournalHidden6_0(Citation c) ~
        Concat(CitedJournalHidden0_0(c), CitedJournalHidden5_0(c))
;
random String CitedName(Citation c) ~
        Categorical({
                CitedJournalHidden5_0(c)->0.1431255544, //Only this option shown here
                CitedNameHidden0_4(c)->0.1351656986,
                ...
        })
;
```


Figure 17: A truncated representation of a generated CPD for `CitedJournal`.

CitedName	CitedJournal	Citation
(u)	FirstWord+(u)	s. a. goldman. how to use expert advice. mar. chem., 1993”
(u)+LastName	FirstWord+(u)	s. a. goldman. how to use expert advice. mar. chem., 1993”
(u)+LastName	Abbreviate(FirstWord)+(u)	s. a. goldman. how to use expert advice. mar. chem., 1993”
(u)+LastName	Abbr(FirstWord)+(u)+Abbr(SecondWord)+(u)	s. a. goldman. how to use expert advice. mar. chem., 1993”
Abbr(FirstName)+(u)+LastName	Abbr(FirstWord)+(u)+Abbr(SecondWord)+(u)	s. a. goldman. how to use expert advice. mar. chem., 1993”
Abbr(FirstName)+(u)+MiddleInitial+(u)+LastName	Abbr(FirstWord)+(u)+Abbr(SecondWord)+(u)	s. a. goldman. how to use expert advice. mar. chem., 1993”
...	...	...
Abbr(FirstName)+”. ”+MiddleInitial+”. ”+LastName	Abbr(FirstWord)+”. ”+Abbr(SecondWord)+”.”	s. a. goldman. how to use expert advice. mar. chem., 1993”

Table 4: This table demonstrates the progressive understanding of an example citation as the CPDs for `CitedName` and `CitedJournal` develop. `Categorical` distributions are omitted and replaced by the choice in each `Categorical` that matches the given example. `LetterUnigram` distributions are replaced by (u). Plus signs denote concatenation. Only one of many created rules are shown in each row.