Learning Probabilistic Relational Dynamics for Multiple Tasks

Ashwin Deshpande, Brian Milch, Luke S. Zettlemoyer and
Leslie Pack Kaelbling

Massachusetts Institute of Technology CSAIL
32 Vassar St. Building 32, Cambridge, MA 02139, USA
{ashwind,milch,lsz,lpk}@csail.mit.edu

Abstract. The ways in which an agent's actions affect the world can often be modeled compactly using a set of relational probabilistic planning rules. This extended abstract addresses the problem of learning such rule sets for multiple related tasks. We take a hierarchical Bayesian approach, in which the system learns a prior distribution over rule sets. We present a class of prior distributions parameterized by a *rule set prototype* that is stochastically modified to produce a task-specific rule set. We also describe a coordinate ascent algorithm that iteratively optimizes the task-specific rule sets and the prior distribution. Experiments using this algorithm show that transferring information from related tasks significantly reduces the amount of training data required to predict action effects in blocks-world domains.

Keywords. Hierarchical Bayesian models, transfer learning, multi-task learning, probabilistic planning rules

1 Introduction

One of the most important types of knowledge for an intelligent agent is that which allows it to predict the effects of its actions. For instance, imagine a robot that retrieves items from cabinets in a kitchen. This robot needs to know that if it grips the knob on a cabinet door and pulls, the door will swing open; if it releases its grip when the cabinet is only slightly open, the door will probably swing shut; and if it releases its grip when the cabinet is open nearly 90 degrees, the door will probably stay open. Such knowledge can be encoded compactly as a set of *probabilistic planning rules* [1,2]. Each rule specifies a probability distribution over sets of changes that may occur in the world when an action is executed in a certain context. To represent domains concisely, the rules must be relational rather than propositional: for example, they must make statements about cabinets in general rather than individual cabinets.

Algorithms have been developed for learning relational probabilistic planning rules by observing the effects of actions [3,4]. But with current algorithms, if a robot learns planning rules for one kitchen and then moves to a new kitchen where its actions have slightly different effects, it must learn a new rule set from

Dagstuhl Seminar Proceedings 07161
Probabilistic, Logical and Relational Learning - A Further Synthesis
http://drops.dagstuhl.de/opus/volltexte/2008/1384

$$pickup(X,Y): on(X,Y), block(Y), clear(X), inhand\text{-}nil, \neg wet \rightarrow \begin{cases} .7: & inhand(X), \neg clear(X), \\ & \neg on(X,Y), clear(Y), \\ & \neg inhand\text{-}nil \\ .2: & on(X, \text{TABLE}), \neg on(X,Y) \\ .05: & \text{no change} \\ .05: & \text{noise} \end{cases}$$

$$pickup(X,Y): on(X,Y), block(Y), clear(X), inhand\text{-}nil, wet \rightarrow \begin{cases} .2: & inhand(X), \neg clear(X), \\ & \neg on(X,Y), clear(Y), \\ & \neg inhand\text{-}nil \\ .2: & on(X, \text{TABLE}), \neg on(X,Y) \\ .3: & \text{no change} \\ .3: & \text{noise} \end{cases}$$

Fig. 1. Two rules for the *pickup* action in the "slippery gripper" blocks world.

scratch. Current rule learning algorithms fail to capture an important aspect of human learning: the ability to transfer knowledge from one task to another. We address this transfer learning problem in this extended abstract. A more complete treatment is given in our conference paper [5] and in the master's thesis by Deshpande [6].

2 Probabilistic Planning Rules

A set of probabilistic planning rules defines a distribution $p(s_t|s_{t-1}, a_t)$ for the state reached by taking action a_t in state s_{t-1}. A state is an interpretation of a set of predicates over a fixed domain of objects. For instance, the sentence,

$$\begin{aligned} &inhand\text{-}nil \wedge on(\text{B-A}, \text{B-B}) \wedge on(\text{B-B}, \text{TABLE}) \wedge clear(\text{B-A}) \\ &\wedge block(\text{B-A}) \wedge block(\text{B-B}) \wedge table(\text{TABLE}) \wedge \neg wet \end{aligned} \tag{1}$$

describes a blocks-world state where the gripper holds nothing, block B-A is on top of block B-B which is on the table, and the gripper is not wet. An action a_t is a ground atom: for example, $a_t = pickup(\text{B-A}, \text{B-B})$ represents an attempt to pick block B-A up off of block B-B.

Fig. 1 shows two rules for the $pickup(X,Y)$ action. Each rule r has two parts that determine when it is applicable: an action term z and a context formula Ψ, which is a conjunction of literals. Given a particular state s_{t-1} and action a, we can determine whether a rule *applies* by computing a variable binding θ that unifies z with a, and then testing whether the context formula holds for this binding. For example, for a state s described by sentence (1) and an action $a = pickup(\text{B-A}, \text{B-B})$, only the first rule in Fig. 1 applies, because *wet* is not true in s. A default rule handles cases where no other rule applies; we disallow *overlapping rules* that apply to the same (s, a) pair.

Given the applicable rule r, we can look to the right of the $\rightarrow$ to find a discrete distribution $\mathbf{p}$ over a set of outcomes O, defining what changes may happen from s_{t-1} to s_t. Each non-noise outcome $o \in O$ is a conjunction of

literals. If outcome o occurs, then the resulting state s_t is formed by taking s_{t-1} and changing the values of the relevant literals to match $\theta(o)$. For instance, in the first rule in Fig. 1, the first outcome is where the picking up succeeds: it sets five truth values, including setting $on(\text{B-A}, \text{B-B})$ to *false*. We enforce the restriction that outcomes do not overlap: for each pair of outcomes o_1 and o_2 in a rule r, there cannot exist a state–action pair (s, a) such that r is applicable and the resulting states for o_1 and o_2 are the same. The *noise outcome* is a special case: it just gives a small probability $p_{\min}$ to every outcome, allowing the rule learner to avoid modeling overly complex, rare action effects [4]. The probability $p(s_t|s_{t-1}, a_t)$ is thus equal to p_o if s_t is the result of an outcome o in the applicable rule, and $p_{\text{noise}}p_{\min}$ otherwise.

3 Hierarchical Bayesian Model

We formalize our transfer learning problem by assuming that we have training sets from K related *source tasks*, plus a limited set of examples from a *target task* $K+1$, and our goal is to find the best rule set R^*_{K+1} for the target task. Our approach is based on *hierarchical Bayesian models*, which have long been used to transfer predictions across related data sets in statistics [7]. The basic idea, as illustrated in Fig. 2, is to regard the task-specific models $R_1, \ldots, R_K, R_{K+1}$ as samples from a global prior distribution G. This prior distribution over models is not fixed in advance, but is learned by the system; thus, the learner discovers what the task-specific models have in common.

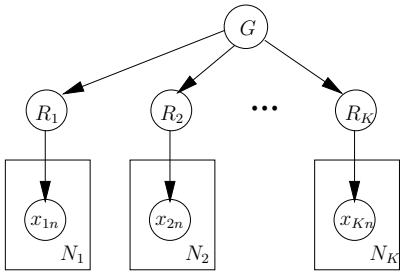


Fig. 2. A hierarchical Bayesian model with K tasks, where the number of examples for task k is N_k.

More precisely, by observing data from the first K tasks, the learner gets information about $R_1, \ldots, R_K$ and hence about G. Then when it encounters task $K+1$, its estimates of R_{K+1} are influenced by both the data observed for task $K+1$ and the prior $p(R_{K+1}|G)$, which captures its expectations based on the preceding tasks.

Although the hierarchical Bayesian approach has been used in a variety of transfer learning settings [8,9,10], applying it to sets of relational probabilistic

planning rules poses both conceptual and computational challenges. In most existing applications, the models R_k are represented as real-valued parameter vectors, and the hypothesis space for G is a class of priors over real vectors. But what family of G-values could parameterize a class of priors over rule sets, which are complicated discrete structures?

We propose to let the possible values of G be *rule set prototypes* that are modified stochastically to create task-specific rule sets. A rule set prototype is a set of *rule prototypes*. Each rule prototype is like an ordinary rule, except that rather than specifying a probability distribution over its outcomes, it specifies a vector Φ of Dirichlet parameters that define a prior over outcome distributions.

3.1 Overview of Model

Our model defines a joint distribution $p(G, R_1, \ldots, R_{K+1}, x_1, \ldots, x_{K+1})$. In our setting, each example x_{kn} is a state s_t obtained by performing a known action a_t in a known initial state s_{t-1}. Thus, $p(x_{kn}|R_k)$ can be found by identifying the rule in R_k that applies to (s_{t-1}, a_t), as discussed in Sec. 2.

The distribution for G and $R_1, \ldots, R_{K+1}$ is defined by a generative process that first creates G, and then creates $R_1, \ldots, R_{K+1}$ by modifying G. Note that this generative process is purely a conceptual device for defining our probability model: we never actually draw samples from it. Instead, in Sec. 4, we will use the joint probability as a scoring function for finding the best rule set prototype.

Two difficulties arise in using a generative process to define our desired joint distribution. One is that the process can yield rule sets that are invalid, in the sense of containing overlapping rules or outcomes. The other is that many possible runs of the process may yield the same rule set. For instance, as we will see, a rule set is generated by choosing a number m, generating a sequence of m rules given the rule set prototype, and then returning the set of distinct rules that were generated. In principle, a set of m^* distinct rules could be created by generating a list of any length $m \geq m^*$ (with duplicates); we do not want to sum over all these possibilities to compute the probability of a given rule set. Both of these problems can be solved by excluding certain invalid or non-canonical runs of the generative process.

Thus, we will define unnormalized measures $P_G(G)$ and $P_{\text{mod}}(R_k|G)$ that give the probability of generating a rule set prototype G, or a rule set R_k, through a "valid" sampling run. The resulting joint distribution is:

$$p(G, R_1, \ldots, R_{K+1}, x_1, \ldots, x_K) = \frac{1}{Z} P_G(G) \prod_{k=1}^{K+1} P_{\text{mod}}(R_k|G) p(x_k|R_k) \qquad (2)$$

The normalization constant Z is the total probability of valid runs of our generative process. Since we are just interested in the relative probabilities of hypotheses, we never need to compute this constant.

The remainder of this section gives a brief description of the distributions $P_{\text{mod}}(R_k|G)$ and $P_G(G)$. Details are available in our conference paper [5].

3.2 Modifying the Rule Set Prototype

We begin by describing how a rule set prototype G is modified to create a rule set R. The first step is to choose the rule set size, m, from a distribution centered on the number of rule prototypes in G. Next, for $i = 1$ to m, we generate a local rule r_i. The first step in generating r_i is to choose which rule prototype it will be derived from. With a certain probability, the rule is generated from scratch; otherwise, its prototype is selected uniformly from G. Since this choice is made independently for each local rule, a single rule in G may serve as the prototype for several rules in R, or for none. Next, given the chosen prototype r^*, the local rule r_i is generated according to the distribution $P_{\text{rule}}(r|r^*)$ discussed in Sec. 3.3.

This process generates a list of rules $r_1, \ldots, r_m$. We consider a run to be invalid if any of these rules are overlapping; in particular, this prohibits cases where the same rule occurs twice. So the probability of generating a set $\{r_1, \ldots, r_m\}$ on a valid run is the sum of the probabilities of all permutations of this set. This is $m!$ times the probability of generating the rules in any particular order.

3.3 Modifying and Creating Rules

We now define the distribution $P_{\text{rule}}(r|r^*)$, where r^* may be either a rule prototype, or the value NIL, indicating that r is generated from scratch. The first step is to choose the action term in r. If r^* is not NIL, we assume the action term is unchanged; otherwise, we choose an action predicate uniformly at random and introduce a distinct logical variable for each argument. Next, we generate the context for r using the formula-modification process described in Sec. 3.4. The input to that process is r^*'s context, or an empty formula if r^* is NIL.

To generate the outcome set in r from that in r^*, we use essentially the same method we used to generate the rule set R from G. We begin by choosing n, the size of the outcome set. Then, for $i = 1$ to n, we choose which prototype outcome serves as the source for the ith local outcome. As in the case of rules, there is some probability that an outcome is generated from scratch. To generate the outcomes from their chosen prototypes, we again use the formula-modification process in Sec. 3.4. A list of outcomes is considered valid if it contains no repeats and no overlapping outcomes. Since repeats are excluded, the probability of a set of n outcomes is $n!$ times the probability of any corresponding list.

The last step is to generate the outcome probabilities in r. These probabilities are sampled from a Dirichlet distribution whose parameters depend on the prototype parameters Φ in r^* and the mapping from local outcomes to prototype outcomes. The Dirichlet weight for each prototype outcome is divided among the local outcomes derived from it; see [5] for details.

3.4 Modifying Formulas

Since we restrict our formulas to be conjunctions of literals, we can think of a given formula ϕ^* simply as a set of literals. To obtain a new formula ϕ, we

begin by choosing which literals in ϕ^* will remain in ϕ. Each literal in ϕ^* is kept independently with a certain probability β_{literal}.

Next, we choose how many new literals (if any) to add to ϕ: this number is chosen from a geometric distribution with parameter α_{literal}. For each new literal, we choose a predicate uniformly at random, and then choose each argument uniformly from the set of constant symbols plus the logical variables that are currently in scope. The run is considered invalid if the resulting atomic formula was already in ϕ. Finally, the polarity of this atomic formula in ϕ (positive or negative) is chosen uniformly at random.

3.5 Generative Model for Rule Set Prototypes

The process that defines the hyperprior $P_G(G)$ is similar to the process that generates local rule sets from G, but all the rule prototypes are generated from scratch — there are no higher-level prototypes from which they could be derived. We assume that the number of rule prototypes in G has a geometric distribution.

The distribution for generating a rule prototype is the same as that for generating a local rule from scratch, except that we must generate a vector of Dirichlet weights Φ rather than a probability distribution over outcomes. We use a hyperprior on Φ in which the sum of the weights has an exponential distribution.

4 Learning

Given data for tasks $1, \ldots, K+1$, our hierarchical Bayesian model tells us that the best target-task rule set R^*_{K+1} can be obtained from Eq. 2 by integrating out the prototype G and the source-task rule sets $R_1, \ldots, R_K$, and then maximizing over R_{K+1}. Rather than attempting the intractable integration over all rule set prototypes, we approximate by maximizing over G. Thus, we work in two stages: first, we find the best rule set prototype G^* given the data for the K source tasks; then, holding G^* fixed, we find the best rule set R^*_{K+1} given G^* and x_{K+1}.

Our goal in the first stage, then, is to find the prototype G^* with the greatest posterior probability given $x_1, \ldots, x_K$. In this optimization, we regard each rule set R_k as consisting of a structure R^{S}_k and parameters R^{P}_k, namely the outcome probability vectors for all the rules. To avoid intractable sums, we maximize over $R^{\text{S}}_1, \ldots, R^{\text{S}}_K$. However, we integrate out the parameters R^{P}_k using standard Dirichlet-multinomial formulas [11]. Maximizing over these parameters would be inappropriate because their dimensionality varies with the number of rules and outcomes, and the heights of density peaks in spaces of differing dimension are not necessarily comparable.

4.1 Scoring Function

We thus find ourselves searching for values of G and $R^{\text{S}}_1, \ldots, R^{\text{S}}_K$ that maximize:

$$P(G, R^{\text{S}}_1, \ldots, R^{\text{S}}_K) \propto P_G(G) \prod_{k=1}^{K} \int_{R^{\text{P}}_k} P_{\text{mod}}(R_k|G) P(x_k|R_k) \tag{3}$$

This equation trades off three factors: the complexity of the rule set prototype, represented by $P_G(G)$; the differences between the local rule sets and the prototype, $P_{\text{mod}}(R_k|G)$; and how well the local rule sets fit the data, $P(x_k|R_k)$.

Computing the value of Eq. 3 for a given choice of G and $R_1^{\text{S}}, \ldots, R_K^{\text{S}}$ is expensive because it involves summing over all possible mappings from local rules to global rules and from local outcomes to prototype outcomes. Thus, as a final approximation, we compute this score using a single correspondence for each local rule and local outcome. Since our model assumes that a prototype is chosen for each rule independently, we can optimize the correspondence for each rule separately, and likewise for outcomes.

4.2 Coordinate Ascent

We find a local maximum of Eq. 3 using a coordinate ascent algorithm. We alternate between maximizing over local rule set structures given an estimate of the rule set prototype G, and maximizing over the rule set prototype given estimates of the rule set structures $(R_1^{\text{S}}, ..., R_K^{\text{S}})$. We begin with an empty rule set prototype, and use a greedy local search algorithm (described below) to optimize the local rule sets. Since $R_1^{\text{S}}, \ldots, R_K^{\text{S}}$ are conditionally independent given G, we can do this search for each task separately. When these searches stabilize — that is, no search operator improves the objective function — we run another greedy local search to optimize G. We repeat this cycle until no more changes occur.

4.3 Learning Local Rule Sets

Our search algorithm for finding the highest-scoring local rule set structure R_k^{S} given the prototype G is a modified version of the rule set learning algorithm from Zettlemoyer *et al.* [4]. The search starts with a rule set that contains only the default rule. At every step, we take the current rule set and apply a set of search operators to create new rule sets. Each of these new rule sets is scored using Eq. 3; the highest scoring set is selected as the new R_k^{S}. The following operators are used to create candidate rule sets:

- Add/Remove Rule. Rules can be created by an *ExplainExamples* procedure [4] that uses a heuristic search to find high quality potential rules in a data driven manner. In addition, rules can be created by copying the action and context of one of the prototypes in the global rule set; this provides a strong search bias towards rules that have been found to be useful for other tasks. Also, any existing rule can be removed.
- Add/Remove Literal. Any literal may be added or removed in the context of an existing rule.
- Split Rule. Any existing rule can be split on an atomic formula ℓ that does not currently occur in its context: the rule is replaced by two rules, one with ℓ in its context and one with $\neg\ell$.

Each of these operators also determines the outcomes for rules that it adds or modifies. This is done with a subsidiary greedy search, where the operators include adding or removing an outcome, adding or removing a literal in an outcome, splitting an outcome on a literal, and merging outcomes. The interesting part here is outcome addition: the added outcome may be derived by concatenating the changes seen in a training example (following [3]), or it may be any outcome from the corresponding prototype rule.

4.4 Learning the Rule Set Prototype

The second optimization involves finding the highest scoring rule set prototype G given rule set structures $(R_1^{\mathrm{S}}, ..., R_K^{\mathrm{S}})$. Again, we adopt an approach based on greedy search through the space of possible rule sets. This search has exactly the same initialization and uses all of the same search operators as the local rule set search. However, the *AddRule* operator tries to add rules that are present in the local rule sets, without directly referencing the training sets. Also, the Dirichlet parameters for the outcomes in each candidate prototype rule are estimated using a gradient ascent technique [11].

5 Experiments

We evaluate our learning algorithm on synthetic data from two families of related tasks, both variants of the classic blocks world (for additional experiments, see [5]). We restrict ourselves to learning the effects of a single action, $pickup(X, Y)$. Since the action is always observed, one could learn a rule set for multiple actions by learning a rule set for each action separately.

5.1 Methodology

For each run of our experiments, we begin by generating K "source task" rule sets from a prior distribution (implemented by a special-purpose program for each family of tasks). Then, we generate a training set for each source task. Each state transition in the training set is constructed by choosing the initial state randomly, and then sampling the next state according to the task-specific rule set. Note that the state transitions are sampled independently of each other; they do not form a trajectory. Once we have these K source-task training sets, we run our learning algorithm on them to find the best rule set prototype G^*.

Next, we generate a "target task" rule set R_{K+1} using the same distribution used to generate the source task rule sets. We also generate a target-task training set in the same way. Then we learn a rule set $\widehat{R}_{K+1}$ for the target task using the algorithm from Sec. 4.3, with G^* as the fixed rule set prototype.

Finally, we generate a test set of 1000 initial states. For each initial state s, we compute the *variational distance* between the next-state distributions defined by the true rule set R_{K+1} and the learned rule set $\widehat{R}_{K+1}$. This is defined in our

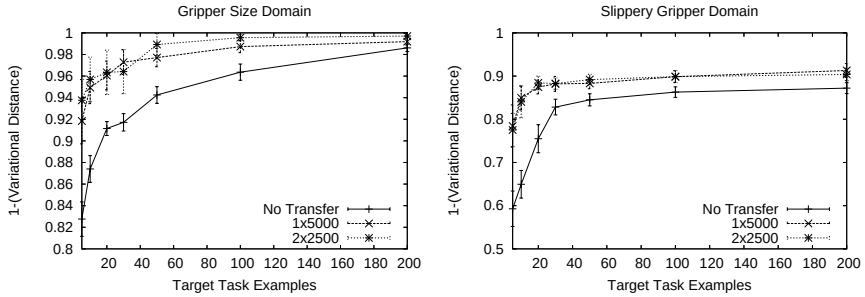
case as follows, with a equal to *pickup*(A, B) and s' ranging over possible next states:

$$\sum_{s'} \left| p(s'|s, a, R_{K+1}) - p(s'|s, a, \widehat{R}_{K+1}) \right|$$

To obtain a measure of accuracy, we use one minus the average variational distance over the whole test set.

5.2 Results

Our first experiment investigates transfer learning in a domain where the rule sets are very simple — just single rules — but the rule contexts vary across tasks. We use a family of tasks where the robot is equipped with grippers of varying sizes: the robot can only pick up blocks that are the same size as its gripper. Thus, each task can be described by a single rule saying that if block X has the proper size (which varies from task to task), then *pickup*(X, Y) succeeds with some significant probability. The domain also includes distracter predicates for block color and texture. Fig. 3(a) shows the transfer learning curves for this domain: the transfer learners are consistently able to learn the dynamics of the domain with fewer examples than the non-transfer learner.



(a) (b)

Fig. 3. Accuracy with an empty rule set prototype (labeled "No Transfer") and with transfer learning from K source tasks with N examples each (labeled KxN). Each experiment was repeated 20 times; these graphs show the average results with 95% confidence bars.

To see how transfer learning works for more complex rule sets, our next experiment uses a "slippery gripper" domain adapted from [1]. The correct model for this domain has four fairly complex rules, describing cases where the gripper is wet or not wet (which influences the success probability for *pickup*) and the block is being picked up from the table or from another block (in the latter case, the rule must include an additional outcome for the block falling on the table). The various tasks are all modeled by rules with the same structure, but include

relatively large variation in outcome probabilities. Fig. 3(b) shows that again, transfer significantly reduces the number of examples required to achieve high accuracy. Experiments on a more difficult set of tasks yield similar results [5].

6 Conclusion

In this work, we developed a transfer learning approach for relational probabilistic world dynamics. We presented a hierarchical Bayesian model and an algorithm for learning a generic rule set prior which, at least in our initial experiments, holds significant promise for generalizing across different tasks. This learning problem is particularly difficult due to the need to learn relational structure along with probabilities simultaneously for a large number of tasks. The current approach addresses many of the fundamental challenges for this task and provides a strong example that can be extended to work in more complex domains and with a wide range of representation languages.

References

1. Kushmerick, N., Hanks, S., Weld, D.S.: An algorithm for probabilistic planning. Artificial Intelligence **76** (1995) 239–286
2. Blum, A.L., Langford, J.C.: Probabilistic planning in the Graphplan framework. In: Proc. 5th European Conference on Planning. (1999)
3. Pasula, H.M., Zettlemoyer, L.S., Kaelbling, L.P.: Learning probabilistic relational planning rules. In: Proc. 14th International Conference on Automated Planning and Scheduling. (2004)
4. Zettlemoyer, L.S., Pasula, H.M., Kaelbling, L.P.: Learning planning rules in noisy stochastic worlds. In: Proc. 20th AAAI National Conference on Artificial Intelligence. (2005)
5. Desphande, A., Milch, B., Zettlemoyer, L.S., Kaelbling, L.P.: Learning probabilistic relational dynamics for multiple tasks. In: Proc. 23rd Conference on Uncertainty in Artificial Intelligence. (2007)
6. Deshpande, A.: Learning probabilistic relational dynamics for multiple tasks. Master's thesis, Dept. of Electrical Engineering and Computer Science, Massachusets Institute of Technology (2007)
7. Lindley, D.V.: The estimation of many parameters. In Godambe, V.P., Sprott, D.A., eds.: Foundations of Statistical Inference. Holt, Rinehart and Winston, Toronto (1971)
8. Yu, K., Tresp, V., Schwaighofer, A.: Learning Gaussian processes from multiple tasks. In: Proc. 22nd International Conference on Machine Learning. (2005)
9. Marx, Z., Rosenstein, M.T., Kaelbling, L.P., Dietterich, T.G.: Transfer learning with an ensemble of background tasks. In: NIPS Workshop on Inductive Transfer. (2005)
10. Zhang, J., Ghahramani, Z., Yang, Y.: Learning multiple related tasks using latent independent component analysis. In: Advances in Neural Information Processing Systems 18. MIT Press (2006)
11. Minka, T.P.: Estimating a Dirichlet distribution. Available at http://research.microsoft.com/∼minka/papers/dirichlet (2003)