Microsoft Research

An Introduction to Infer.NET

A look at probabilistic programming and Microsoft Infer.NET

Version 1.0 – October 11, 2011

Abstract

This document provides a brief high-level introduction to probabilistic programming and Microsoft® Infer.NET.

For more detailed information and a walkthrough of sample applications, see "Infer.NET 101," listed in Resources at the end of this paper.

In this paper

Note:

To provide feedback, send e-mail to infersup@microsoft.com.

Version 1.0 – October 11, 2011

Content

Overview

Computers are rigorously logical but the real world is not, a simple fact that can pose some real challenges for programmers. For example, suppose you must represent what a user scribbles on a touch screen as a word. People usually aren't very careful or consistent about their handwriting, so does that scribble correspond to "hill", or is it "bull" or maybe even "hello"? The user knows what they wrote, but from the application's perspective, the value of the correct word is fundamentally uncertain. It's not completely uncertain though; "hull" is more likely than "gall" and you can completely rule out words like "train" or "helicopter."

How do you represent such uncertainty in a program, though? Conventional variables such as **bool** or **int** must have well-defined values. A scribble needs a variable—call it *someScribble*—that represents any of several possible words; you aren't sure which. However, that variable must represent uncertainty in a way that incorporates your understanding of the probability that each of the possible words might be the right one.

Probabilistic programming is designed to handle such uncertainty. It is based on *random variables*, which are extensions of standard types that can represent uncertain values. Each random variable represents a set or range of possible values, and has an associated *distribution* that assigns a probability to each possible value. The distribution quantitatively represents your understanding of the variable's possible values, and allows you to use statistical analysis to understand the variable's behaviour.

So *someScribble* is now represented by a random variable. How do you obtain *someScribble*'s distribution—the probabilities for each of its possible words? From the user's perspective, there is a cause-effect relationship between a word and the associated scribble. With probabilistic programming, you can construct a probabilistic model that defines how users transform words to scribbles. This model recognizes, among other things, that the same word can lead to different scribbles and that different words can lead to similar scribbles, and defines the associated probabilities.

How do you reason backwards from a particular scribble to the set of possible words and their probabilities? Probabilistic programming is based on a statistical methodology known as Bayesian inference. It allows you to reason backwards from an observation—the scribble—to its origin—the possible words and their probabilities—based on a probabilistic model.

A model usually has a set of adjustable parameters that govern its behaviour. How do you determine the correct parameter settings? You could just assign parameter values based on your general understanding of handwriting, which might work reasonably well. However, everybody writes a bit differently, so the possible words associated with *someScribble* and their probabilities will vary somewhat from user to user. To fine-tune the parameters for an individual user's particular writing style, a probabilistic program can treat the model parameters themselves as random variables and learn the real parameter values based on user interaction.

The results look OK, but how do you know that you aren't missing something. Would a more sophisticated model with more variables work even better? If you add enough variables to a model, you can fit almost anything. However, you generally reach a point of diminishing returns; at some point additional complexity starts reducing the model's quality. You need an Occam's razor to find a balance between accuracy and complexity. That "razor" is an integral part of Bayesian inference, which includes a robust way to assess model quality called *evidence* that you can use to pick the best model.

This might sound good in principle, but it's all hand waving so far. How do you actually go about implementing models and inferring probabilities? There are usually multiple ways to address probabilistic problems. However, probabilistic programming provides a natural integrated way to address this class of problems. You can implement probabilistic programs from scratch, but it can be challenging. The Microsoft® Infer.NET framework greatly simplifies the process, by providing:

- A modelling API, which simplifies the mechanics of constructing probabilistic models.

  Even complex models can often be expressed with only a few lines of code.
- An inference engine, which combines that model with your observations, and handles the complex mathematics of inferring probabilities for the specified variable's possible values.

This document provides a brief high-level introduction to probabilistic programming and Infer.NET. If you want to go further, a companion document, "Infer.NET 101," takes you through the basics of Infer.NET programming, based on a set of simple applications.

What Probabilistic Programming Can Do

This section illustrates the value of probabilistic programming by briefly describing three examples of working applications. These examples don't cover the complete range of possibilities, but they should give you a better sense of how probabilistic programming can help you implement new and better applications.

Select Conference Papers

There are usually more papers submitted to conferences than can possibly be accepted for presentation. Conference organizing committees therefore typically have each paper reviewed by several scientists, and use those reviews to select the top papers. However, there are usually too many papers for a single group of reviewers to review every one, so conferences typically have a relatively large group of reviewers, each of whom reviews a subset of the submitted papers.

When the reviews come back, the committee is faced with a problem of how to calibrate review scores. Given reviews for two papers—each of which was reviewed by a different group of scientists who do not necessarily have the same standards or expertise—how do you compare the scores to determine which paper is better? In addition, how do you do so efficiently, so the organizing committee can complete its work in a reasonable amount of time?

Researchers from Microsoft Research, Cambridge developed a Bayesian model to calibrate reviews for the SIGKDD'09 conference's research track. The model separates the individual reviewer's standards and expertise from the papers' inherent quality by using the fact that each paper has several reviewers, each of whom reviews several different papers. For more details, see "Novel Tools to Streamline the Conference Review Process" in "Resources" at the end of this document.

Rank Players in Multi-Player Games

Multi-player games such as those hosted by the Xbox LIVE ® video game service are more interesting if all the players in a particular game session have comparable skill. However, achieving this goal poses some challenges:

- A new player's skill level is unknown.
- A player's skill usually improves with time.
- A player's performance varies from game to game, even against the same opponents.

A good ranking system needs to be able to quickly learn a new player's skill level, and then adjust that ranking as the player continues to play and improve.

The TrueSkill® matchmaking system ranks Xbox LIVE gamers by starting with a standard distribution for new players, and then updating it as the player wins or loses games. TrueSkill can converge on a stable ranking in as few as three games, depending on the number of players. TrueSkill also associates an uncertainty with each level, which provides a more robust way to create compatible groups of players than simply using a skill value.

For more information, see "TrueSkill Ranking System" in "Resources and References." For a discussion of how to implement such an application with Infer.NET, see "How to represent large irregular graphs," in "Resources."

Predict Click-Through Rates

Sponsored search is a key revenue generator for online search engines such as Bing®. The search engines use a keyword auction to allocate space. The auction is based on a pay-per-click model, where advertisers pay only if users select their item from sponsored search results and *click-through* to the corresponding Web page.

To optimize user experience, search engine revenue, and advertiser revenue, the search engine needs to display the results that the user is most likely to click. To select and rank an optimal list of advertisers, the search engine must have an efficient and accurate way to predict the click-through rate for each advertiser.

Researchers from Microsoft Research, Cambridge developed a Bayesian online learning algorithm—called *adPredictor*—to optimize sponsored search results for the Bing search engine. The algorithm is currently running in the Bing production environment, which must handle on the order of 10^{10}-10^{11} advertising impressions per year. For details, see "Web-Scale Bayesian Click-Through Rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine" in "Resources."

How Probabilistic Programming Works

The following "Whodunit" scenario provides a convenient way to illustrate probabilistic programming's basic concepts and define some essential terminology and concepts.

Whodunit

You return to your country house after an evening of whist with your neighbour, only to discover that your houseguest has been foully murdered. You must discover the culprit. Initially, you know that:

- There are two possible culprits: the butler and the cook.
- There are three possible murder weapons: a butcher knife, a pistol, and a fireplace poker.

Initially, the culprit and murder weapon are unknown. However, probabilistic reasoning and observations can help identify the likely culprit, and in the process provide an introduction to the basics of probabilistic programming.

Suspects and Murder Weapons: Random Variables

You need a variable to represent the culprit. One suspect is guilty and the other isn't, so the obvious choice is a variable that can take either of two possible values. A **bool** type can represent two possible values. However, a **bool** type can be only **true** or **false**, and you can't be certain about the variable's actual value until the culprit confesses. You can, however, estimate the probability that each suspect is the culprit.

What you need is a random variable, which essentially extends standard *domain types* such as **bool** or **double** to handle both deterministic and uncertain values.

- A random variable has a set or range of possible values, which are drawn from the domain type.

  For example, the possible values of a **bool** random variable are **true** and **false**. The possible values of a **double** random variable are real numbers over a continuous range such as [0, 1] or [-∞, ∞].

- Each random variable has an associated probability distribution which specifies the probability of each possible value.

  For example, a **bool** random variable could have a 70% probability of being **true**, and a 30% probability of being **false**.

The Whodunit scenario requires two random variables:

- *theCulprit*: A random variable with two possible values: *theButler* and *theCook*.
- *theMurderWeapon*: A random variable with three possible values: *thePistol*, *theKnife*, and *thePoker*.

The Likely Culprit: Probability Distributions

Even though you can't definitively say who committed the murder at this point, you can estimate the probability that each suspect is guilty. Every random variable is associated with a function known as a *probability distribution*—usually shortened to just "distribution"—that assigns a probability to each of the variable's possible values.

The distributions associated with *theCulprit* and *theMurderWeapon* are known as Discrete distributions, which assign probabilities to an enumerable set of possible values. Because the actual value must be one of the possible values, the probabilities must sum to 100%.

Getting Started: The Prior Distribution

Probabilistic programming starts by defining an initial distribution for each random variable, which is called a *prior distribution* or sometimes *prior belief*. It is commonly shortened to just *prior*. A prior defines your understanding of a variable before you have made any observations. For *theCulprit*:

- The butler is an upstanding fellow who has served the family faithfully for years.
- The cook was hired recently, and there are rumours of an unsavoury past.

From this information, you estimate the *theCulprit* prior to be: *theButler* = 20% and *theCook* = 80%.

Investigate: Observations and the Posterior Distribution

The prior is a useful starting point, but you can improve your understanding if you *observe* the actual value of one or more of the random variables in your model. At that point, the variable is no longer uncertain, which allows you to make a better estimate of the other variables' distributions.

For example, suppose that you know that the butler is more likely to have used the pistol, and the cook more likely to have used the knife or poker. After you receive the coroner's report, you can *infer* a new *theCulprit* distribution, which incorporates the observation of *theMurderWeapon* into the *theCulprit* prior and improves your estimate of the probable culprit. If the murder weapon is the pistol, *theButler* increases and *theCook* decreases.

An observation doesn't supplant the prior—that information is still valid—but you can use the additional information to revise your prior belief so that it accurately reflects all available data. The new distribution is known as a *posterior distribution*, or just "posterior."

In fact, the distinction between prior and posterior can sometimes be somewhat blurred. A more general and useful way to look at priors and posteriors is:

- A prior represents your understanding of the system before you make a particular set of observations.
- The corresponding posterior represents your understanding of the system after you have made the observations.

Suppose you come up with an additional observation: a new coroner's report gives an estimate of when the murder was committed. Before you receive this second report, your understanding of the situation is represented by the posterior belief you had after reading the first coroner's report. That posterior is therefore the logical choice for the new prior, which you can then use with the observed murder time to infer a second posterior. This type of inference is incremental in nature and is referred to as online learning. Alternatively you go back to the original prior belief, and reconsider all the evidence from scratch.

The more evidence you have, the less important your prior belief is. Conclusive evidence should override any prior belief, but if you don't have much evidence to work with, prior belief counts for much more.

Quantify the Problem: Conditional, Joint, and Marginal Distributions

The preceding discussion makes some general arguments about how to infer the most likely culprit, but is a bit vague about actual numbers. This section quantifies the discussion.

Conditional Distributions

To infer a posterior, you must first construct a mathematical model for the murder scenario. You have already specified a prior for *theCulprit*. You also know that:

- The butler keeps an old Webley from his Army days in a locked drawer but the cook does not own a pistol.

  The butler is much more likely to have used the pistol.
- The cook has an ample supply of sharp butcher knives—and has forbidden the butler to set foot in the kitchen.

  The butler is less likely to have used the butcher knife.
- The butler is much older than the cook, and getting a bit frail.

  The butler is less likely to have used a physically demanding weapon like the poker.

The simplest way to start constructing a model is to estimate the probability that the suspects used each of the possible murder weapons. This is known as a *conditional distribution*, the distribution for *theMurderWeapon*, conditioned by a particular value for *theCulprit*.

To simplify the discussion, we'll simply specify the two distributions, as shown in Figure 1.

	Pistol	**Knife**	**Poker**	
Cook	5%	65%	30%	**= 100%**
Butler	80%	10%	10%	**= 100%**

Figure 1. Whodunit conditional distributions

Each suspect must use one of the possible weapons, so each conditional distribution sums to 100%.

Joint Distributions

You can use *theCulprit* prior and the conditional distributions from the previous section to construct a model for Whodunit, which is known as a *joint distribution*. A

joint distribution represents the probabilities of all possible combinations of the random variable's possible values—the cook with the knife, the butler with the poker, and so on. It contains your complete understanding of the murder scene before you make any observations.

Figure 2 shows the Whodunit joint distribution.

	Pistol	**Knife**	**Poker**
Cook	4%	52%	24%
Butler	16%	2%	2%

Figure 2. Whodunit joint distribution

Each row contains the conditional probabilities from the previous section multiplied by the corresponding probabilities from the *theCulprit* prior, 80% for the *theCook* and 20% for *theButler*. Because the combination of culprit and weapon must be one of these six possibilities, they sum to 100%.

You can actually create joint distributions in a variety of ways. All of them produce the same numbers. The approach used in this document is basically cause-effect—the culprit selects the weapon and uses it commit the murder.

- *theCulprit* is the variable that we are interested in—and that we want to infer a posterior for—so we define a prior for that variable.
- *theMurderWeapon* is the variable that we can observe, so we define conditional distributions for that variable.

Cause-effect is usually the simplest way to construct a probabilistic model even if you want to reason in the reverse direction, such as inferring the culprit based on knowledge of the murder weapon.

Marginal Distributions

You can use the joint distribution to ask a variety of questions. Suppose we wanted to know the probability that the pistol is the murder weapon. You can compute that from the joint distribution by summing the probability that the cook used the pistol and the probability that the butler used the pistol. You can do the same computation for the knife and the poker. The distribution that remains after you "sum out" all but one variable in the joint distribution is the remaining variable's *marginal distribution*—or more commonly just *marginal*.

Figure 3 shows the marginal for *theMurderWeapon*. Prior to receiving the coroner's report, the most likely murder weapon appears to be the knife or the poker.

	Pistol	Knife	Poker
Cook	4%	52%	24%
Butler	16%	2%	2%
	= 20%	**= 54%**	**= 26%**

Figure 3. Marginal distribution for *theMurderWeapon*

You can compute the marginal for *theCulprit* in the same way, which simply gives you back the prior probabilities that were specified earlier.

	Pistol	Knife	Poker	
Cook	4%	52%	24%	**= 80%**
Butler	16%	2%	2%	**= 20%**

Figure 4. Marginal for *theCulprit*

So far, these marginals basically tell you what you already know. They are more interesting after you make an observation and add some new information to the mix.

Whodunit: Infer a Posterior

When the coroner's report arrives, you can use the observation of the murder weapon to infer a posterior for *theCulprit*, which should contain an improved estimate of the probable culprit.

The posterior is a conditional marginal, the marginal for *theCulprit*, conditioned by the observation that the murder weapon is the pistol. In this simple case, you can obtain the posterior from the joint distribution table.

The coroner's report means that you can eliminate the knife and the poker from the grid. The remaining values for the pistol indicate the probability that each suspect is guilty. However, they are not a proper distribution; the numbers don't add up to 1.0. To finish the computation, divide each value by the marginal for *thePistol* (0.20). This renormalizes the values and produces the posterior distribution to the right of the table, as shown in Figure 5.

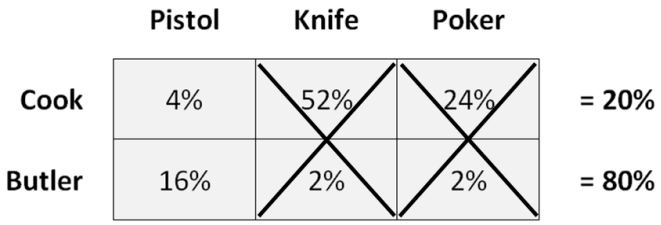

Figure 5. Posterior for *theCulprit*

The result isn't completely certain, but it looks bad for the butler!

In practice, models are usually much more complicated than the one used in this example, which in turn makes computing posteriors much more difficult. Applying probabilistic programming to "real world" scenarios requires a more sophisticated approach to both model construction and inference.

What's Next: A More Sophisticated Approach

The joint distribution in the previous section is a very simple model, with only two variables and a small set of possible values. While the observation of the murder weapon strongly suggests that the butler is the likely culprit, there's still a possibility that the cook is guilty. A more sophisticated model that could handle a wider range of observations might provide more certainty. However, computing posteriors for more sophisticated models is correspondingly more difficult.

More Sophisticated Models

You could perhaps handle the additional variables by expanding the joint distribution table in Figure 2. However, for more than two variables—or for variables with many possible values—tables rapidly become unmanageable. In addition, tables are useful only for discrete distributions. If you want to define a distribution for the time of the murder, it must represent a continuous range of values, which can't be represented by a table at all.

A more flexible and powerful approach is to create a conceptual model for the joint distribution in the form of a graph that represents the relationships between the system's random variables. Figure 6 shows two examples, where (a) is the graph for Whodunit, and (b) extends that graph to handle additional random variables.

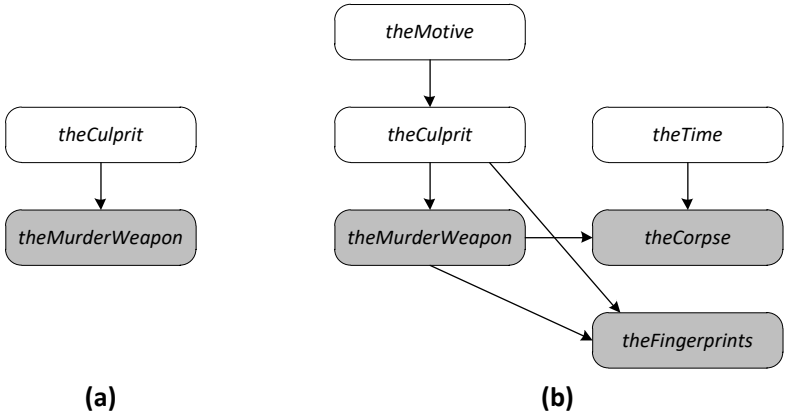

Figure 6. Graphical models

The model represents the relationships between random variables, as follows:

- Each box represents a random variable.
- The arrows indicate cause-effect relationships between random variables.
- The shaded boxes indicate the observable random variables.
- The unshaded boxes indicate unobservable random variables that we would like to infer.

For model (a), the model is: the culprit chooses a murder weapon. You observe the murder weapon, and use the model and observation to infer the likely culprit. Even though the model represents cause-effect—which is known as a generative model—you can use it to reason in either direction. If you know the culprit, for example, you can use that observation to infer the most likely murder weapon.

Model (b) is a more sophisticated model that incorporates additional random variables—the time of the murder, the state of the corpse, and so on—several of which are observable. You can use this model combined with observations of several random variables to compute posteriors, and perhaps either vindicate the butler or produce such overwhelming evidence that he is compelled to confess.

More Sophisticated Inference

With Whodunit, you could use simple arithmetic to infer a posterior from the table in Figure 2. That approach becomes more difficult as you add variables—especially if they have large numbers of possible values—and doesn't work at all for variables that represent a continuous range of possible values. More realistic models require a more sophisticated way to infer posteriors.

The conceptual models in Figure 6 are actually a graphical representation of the mathematical expression of the joint distribution. Probabilistic programs can use this expression and the mathematics of Bayesian inference to infer posteriors for arbitrarily complex graphs, including graphs that have variables with continuous distributions. You can even use the mathematical representation to directly define models that cannot be represented graphically.

How to Pick the Best Model

Model (a) provided an estimate of the probable culprit, but probably not good enough to convict. A more complex and sophisticated model might be more convincing. However, adding more variables doesn't necessarily produce a better model. There is usually a point of diminishing returns, beyond which additional complexity either adds nothing to the model, or actually makes it worse.

For example, when you fit a polynomial to a set of data points, you can always get an exact fit by adding enough elements to the polynomial. However, a polynomial that exactly fits every data point typically swings wildly in between each point—a phenomenon known as overfitting—which might be accurate in some sense, but isn't very useful. A polynomial with fewer elements can often fit the data almost as well, and provide a much more useful and realistic model.

What you want is a happy medium: a model that fits the data reasonably well without being overly complex. In short, you need to apply Occam's razor—the best model is the simplest one that adequately fits the data—to the possible models.

For example, consider the two models in Figure 7.

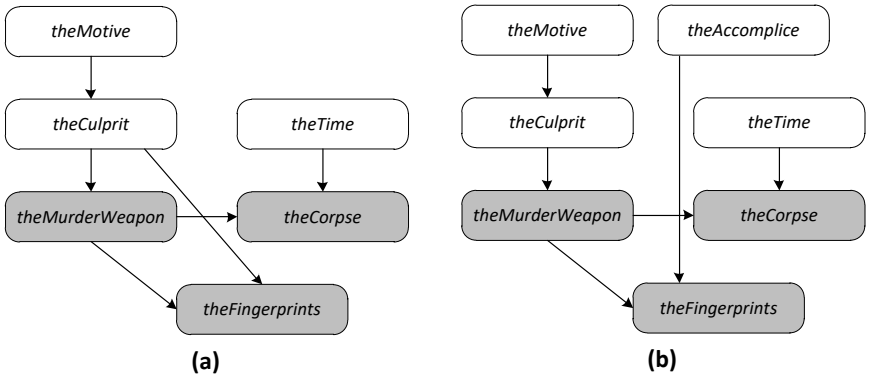

Figure 7. Model comparison

The two models account for the presence of fingerprints in different ways:

- With model (a), the culprit leaves fingerprints at the crime scene.
- With model (b), the culprit was careful about leaving fingerprints, but had an accomplice who was not.

You usually can't be certain which model is the optimal one. However, with probabilistic programming, you can treat evidence as a random variable and infer that variable's distribution. The distribution gives you the probability that each model is optimal, and you can use that information to pick the best model. For example, if the evidence for model (b) has a probability of only 15%, an accomplice probably adds unnecessary complexity to the model. You can therefore stick with the simpler explanation: the butler did it.

Probabilistic Programming with Infer.NET

Probabilistic programming is a general concept, and can be implemented in a variety of ways. What's the advantage of using Infer.NET? In short, Infer.NET provides a straightforward way to represent graphical models in code, and includes an inference engine that handles the complex mathematics of inferring posteriors.

This section describes Infer.NET's features, and how they can help you quickly and easily implement robust probabilistic programs.

Powerful and Flexible Model Construction

Creating a good conceptual model can be difficult, and is outside the Infer.NET scope. However, the Infer.NET API modelling API makes converting a conceptual model into code a simple and straightforward task. The API can be used to implement a wide range of models—including standard models such as Bayes point machine, latent Dirichlet allocation, factor analysis, and principal component analysis—often with only a few lines of code.

Scalable and Composable Models

The Infer.NET modelling API is composable, so that you can implement complex conceptual models from simple building blocks. However, you don't have to implement the entire model at once. For example, you can start with a simplified conceptual model, which captures the basic features. When you have worked the kinks out, you can scale up the model and the data set—in multiple stages if needed—until you have a fully-implemented model that can process real data sets.

Infer.NET models can also be scaled up computationally. You can start with a small data set and scale it up to handle much larger amounts of data, including using parallelized computation.

Built-in Inference Engine

Computing posteriors is usually quite difficult, and requires a deep understanding of Bayesian inference and numerical analysis. Infer.NET includes an inference engine that handles this task for you. With Infer.NET, your application constructs a model, observes one or more variables, and then queries the inference engine for posteriors. The query requires only a single line of code. The inference engine does all the numerical heavy lifting—using any of several supported algorithms—and returns the requested posterior.

Separation of Model from Inference

Probabilistic programs are often designed around a particular problem and implemented monolithically, with no clear distinction between the model and the inference algorithm. This approach has some practical drawbacks, including:

- It is limited to relatively simple models.
- It is difficult to change the model.
- It is easy to introduce inconsistencies in the model.
- It limits you to a particular inference algorithm.

Infer.NET maintains a clear distinction between model and inference.

- The model encodes core prior knowledge.

  An Infer.NET model—even a complex one—is typically confined to a single relatively small block of code. The model is often encapsulated in a separate class, so that you can use the same model for different queries. A separate model is straightforward to understand and modify, and is much more resistant to inconsistencies. Any that do creep in will be caught by the inference engine.

- The inference engine handles the computations.

  You can change the model without touching the inference engine, and you can change the inference algorithm without touching the model.

Quantifies Uncertainty

Bayesian inference doesn't just give you a best-fitting result; it also quantifies the result's uncertainty, which is very useful in interpreting the results. For example, a Bayesian handwriting recognition model doesn't just give you the best-fitting word; it provides a list of possible words and their associated probabilities.

Support for a Variety of Learning Models

Bayesian inference—with its basic model of prior plus observations yield a posterior, which becomes the prior for the next observations—is naturally suited to online learning, and implementing online learning with Infer.NET is simple and straightforward.

Infer.NET also simplifies the implementation of more sophisticated learning techniques, such as hierarchical models. Suppose, for example, that you have a model for a user interaction such as speech recognition. You want to provide new users with a reasonably functional "out-of-the-box" model, but then personalize that model for the individual user based on their interactions. You can use Infer.NET to train the out-of-box model, based on observations of a reasonably large population. You can then use the out-of-box model as the initial prior for a new user, and train the model further to personalize it for that particular user.

Built-in Model Selection Criterion

The ability to balance model complexity against fit is an inherent part of Bayesian inference, which supports a model-selection criterion called *evidence*. With Infer.NET, evidence-based model comparison is simple and straightforward to implement. You can also use evidence to implement *hyperparameter learning*, where a hyperparameter represents a quantity that cannot be learned directly from the model.

Easy to use Different Data Sources

With Infer.NET, you can easily use different data sources for the observed values of different variables. For example, you could combine data from click logs with direct queries to users to infer the basis for user preferences.

Resources

This section contains links to additional information about Infer.NET, or related topics.

Infer.NET

Infer.NET
http://infernet.azurewebsites.net/

Infer.NET 101
http://infernet.azurewebsites.net/InferNet101.pdf

Infer.NET: Building Software with Intelligence (PDC presentation)
http://microsoftpdc.com/Sessions/VTL03

Infer.NET User Guide
http://infernet.azurewebsites.net/docs/default.aspx

General References

Bayesian inference
http://en.wikipedia.org/wiki/Bayesian_inference

Information Theory, Inference, and Learning Algorithms
http://www.inference.phy.cam.ac.uk/mackay/itila/book.html

Novel Tools To Streamline the Conference Review Process: Experiences from SIGKDD'09
http://research.microsoft.com/pubs/122784/ReviewerCalibration.pdf

Pattern Recognition and Machine Learning
http://research.microsoft.com/PRML/

TrueSkill Ranking System
http://research.microsoft.com/trueskill/

Web-Scale Bayesian Click-Through Rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine
http://research.microsoft.com/apps/pubs/default.aspx?id=122779