Making Money Using Math

**MODERN
APPLICATIONS ARE
INCREASINGLY
USING
PROBABILISTIC
MACHINE-LEARNED
MODELS**

ERIK MEIJER

*If Google were
created from
scratch today,
much of it would
be learned, not
coded.*

*—Jeff Dean, Google
Senior Fellow,
Systems and
Infrastructure Group*

Machine learning, or ML, is all the rage today, and there are good reasons for that. Models created by machine-learning algorithms for problems such as spam filtering, speech and image recognition, language translation, and text understanding have many advantages over code written by human developers. Machine learning, however, is not as magical as it sounds at first. In fact, it is rather analogous to how human developers create code using test-driven development.⁴ Given a training set of input-output pairs $\{(a,b) \mid a \in A, b \in B\}$, guess a function $f \in A \rightarrow B$ that passes all the given tests but also generalizes to unspecified input values.

A big difference between human-written code and learned models is that the latter are usually not represented by text and hence are not understandable by human developers or manipulable by existing tools. The consequence is that none of the traditional software

engineering techniques for conventional programs, such as code reviews, source control, and debugging, are applicable anymore. Since incomprehensibility is not unique to learned code, these aspects are not of concern here.

A more interesting divergence between machines and humans is that machines are less arrogant than humans, and they acknowledge uncertainty in their code by returning a *probability distribution* or *confidence interval* of possible answers $f \in A \rightarrow \mathbb{P}(B)$ instead of claiming to know the precise result for every input. For example, a learned image-recognition function by a major cloud provider will predict with 95 percent certainty that I have hair, but is less confident about whether or not I am professional (figure 1).

The implication of incorporating learned models in human-written code is that you cannot get around the fact that the building blocks from which humans compose applications are fundamentally probabilistic. This is a challenge for mainstream programming languages, which all assume that computations are precise and deterministic. Fortunately, the 18th-century Presbyterian minister Thomas Bayes anticipated the need for dealing with uncertainty and formulated Bayes's rule⁶:

$$\mathbb{P}(A|B) * \mathbb{P}(B) = \mathbb{P}(A \& B) = \mathbb{P}(B|A) * \mathbb{P}(A)$$

As it turns out, Bayes's rule is exactly what the doctor ordered when it comes to bridging the gap between ML and contemporary programming languages.

Many of the mathematical explanations of Bayes's rule are deeply confusing for the working computer scientist, but, remarkably, when interpreted from a functional

1

FIGURE 1: **IMAGE RECOGNITION RESULTS**

programming point of view, *Bayes's rule is a theorem about composability and invertibility of monadic functions*. Let's break down Bayes's rule piece by piece and rebuild it slowly based on developer intuition.

PROBABILITY DISTRIBUTIONS

First let's explore what probability distributions $\mathbb{P}(A)$ are. The Wikipedia definition, "*a probability distribution is a mathematical description of a random phenomenon in terms of the probabilities of events*," is rather confusing from a developer perspective. If you click around for a bit, however, it turns out that a discrete distribution is just a generic list of

pairs of values and probabilities $\mathbb{P}(A)=[A \mapsto \mathbb{R}]$ such that the probabilities add up to 1. This is the *Bayesian representation* for distributions. Isomorphically, you can use the *frequentist representation* of distributions as infinite lists of type `dist ∈ [A]` such that as `n` gets larger, sampling from the collection and counting the frequencies of each element

```
from a in dist.Take(n) group by a into g select
g.Key ↦ g.Sum()/n
```

approximates the Bayesian representation of the distribution. When converting from the Bayesian to the frequentist implementation, the probabilities do not have to add up to 1, and the sampling process will ensure that the ratios are properly normalized.

Like true mathematicians, we will silently switch between these two representations of distributions whenever convenient. Unlike mathematicians, however, to keep things simple we will not consider continuous distributions. We want our distribution to hold generically for any type `A`, and most of the types we deal with in code are discrete and not “measurable” or real number-like.

Because the values we care about are usually not even comparable, we will also avoid cumulative distributions. One reason that mathematicians like standard continuous distributions—such as Gaussian, beta, binomial, and uniform—is because of their nice algebraic properties, called *conjugate priors*.² For example, a uniform prior combined with a binomial likelihood results in a beta posterior. This makes 18th- and 19th-century probabilistic computations using pencil and paper feasible, but that is

not necessary now that there are powerful computers that can run millions of simulations per second.

In programming examples, distributions typically come from outside data as discrete frequentist collections of data with an unknown distribution, or they are defined explicitly as a Bayesian representation by enumerating a list of value/probability pairs. For example, here is the distribution of weight of adults in the United States, according to the CDC [Centers for Disease Control]:

$\text{CDC} \in \mathbb{P}(\text{Weight})$

$\text{CDC} = [\text{obese} \rightarrow 0.4, \text{skinny} \rightarrow 0.6]$

Efficiently sampling from composed distributions is, indeed, rocket science. Just like database query optimizers, advanced sampling methods leverage properties of the leaf distributions and the structure of the query⁹ or program³ that computes the distribution. It leverages deep and complex mathematical techniques such as importance sampling, Metropolis-Hastings, Markov Chain Monte Carlo, and Gibbs sampling that are far outside the scope of this article but that are important for making real-world computations over probability distributions feasible. As Bayesian analysis consultant John D. Cook remarked “... Bayesian statistics goes back over 200 years, but it didn’t take off until the 1980s because that’s when people discovered practical numerical methods for computing with it...”

To illustrate the sophistication involved in efficiently sampling known discrete distributions, imagine converting the example distribution `CDC` into a frequentist representation.⁸

2

FIGURE 2: FREQUENTIST REPRESENTATION

Perhaps the most obvious method stacks the columns for *skinny* and *obese* on top of each other and draws one random number—say, p —between 0 and 1 and then checks if $p \leq 0.4$ yields *obese*, and otherwise yields *skinny*. In general, this search is linear in the number values in the distribution, but using tricks like binary search tree can speed things up. Mathematicians call this the *inverse transfer* method.

Another way is first to select a random integer—say, *weight*—to select between *obese* and *skinny*, and then choose a random double between 0 and 1—say, p —and if $CDC[weight] \leq p$, then yield *weight*, as shown in figure 2. Mathematicians call this algorithm rejection sampling, and as the histogram shows, half of the attempts to sample a

value from the distribution will fail (the pink part). This can be improved by picking a tighter envelope distribution, like that in the second histogram, but that still rejects 2 out of 12 samples.

The last method pads the lower probabilities by borrowing from the higher probabilities. Amazingly, it is always possible to do this in a way such that every column represents the probabilities for, at most, two values, so we need only one comparison to pick the right value. This comparison can be implemented using a second index table, and hence mathematicians call this sampling algorithm the *alias method*.

CONDITIONAL PROBABILITY DISTRIBUTIONS

Now that we have explained probability distributions $\mathbb{P}(A)$, let's examine conditional probability distributions $\mathbb{P}(B|A)$, which, according to Wikipedia, are "a measure of the probability of an event given that (by assumption, presumption, assertion, or evidence) another event has occurred." To developer ears that sounds exactly like a function $A \rightarrow \mathbb{P}(B)$ that returns a distribution, just like a learned model. The remainder of this article uses the notations $\mathbb{P}(B|A)$ and $A \rightarrow \mathbb{P}(B)$ interchangeably.

Going back to the example, we have the following model $\text{Doctor} \in \mathbb{P}(\text{Food}|\text{Weight})$ of food preference, given weight, that we could have obtained by asking patients what kind of food they consume:

$\text{Doctor} \in \mathbb{P}(\text{Food}|\text{Weight}) = \text{Weight} \rightarrow \mathbb{P}(\text{Food})$
 $\text{Doctor}(\text{obese}) = [\text{burger} \mapsto 0.9, \text{celery} \mapsto 0.1]$
 $\text{Doctor}(\text{skinny}) = [\text{burger} \mapsto 0.3, \text{celery} \mapsto 0.7]$

As argued in the introduction, these probabilistic functions, such as $\mathbb{P}(\text{Object}|\text{Image})$, $\mathbb{P}(\text{Text}|\text{Audio})$, $\mathbb{P}(\text{Spam}|\text{Text})$, etc., are increasingly the result of training some ML algorithm or neural net, instead of being coded by expensive and flaky human developers.

Now that you know that conditional probabilities are probabilistic functions, things are starting to get interesting, since this means that multiplication (*) used in Bayes's rule is an operator that applies a probabilistic function to a probability distribution as a parameter—that is, it has the following type signature:

$$\mathbb{P}(B|A) * \mathbb{P}(A) \in \mathbb{P}(A \& B)$$

Using the Bayesian representation of distributions, you can implement a probabilistic function application `likelihood*prior` where `likelihood` $\in \mathbb{P}(B|A)$ and `prior` $\in \mathbb{P}(A)$, using the following correlated query:

```
likelihood*prior =
  from a↦p in prior
  from b↦q in likelihood(a)
  select (a,b)↦p*q
```

Applying this definition to compute the result of `Doctor*CDC`, we obtain the table shown in figure 3 for the joint probability distribution $\mathbb{P}(\text{Food} \& \text{Weight})$.

Because the distributions for $\mathbb{P}(\text{Weight})$ and $\mathbb{P}(\text{Food})$ appear in the margins of this table, mathematicians call them *marginal probabilities*, and similarly the process of summing up the columns/rows is called

3

marginalization. When computing a joint distribution using $(*)$, mathematicians often use the name *likelihood* for the function and *prior* for the argument.

The beauty of the frequentist representation is that there is no need for multiplying probabilities. Sampling ensures that the underlying ratio of occurrence of values in the result will automatically reflect the proper product of values from the prior and likelihood. For example, if we implement the prior `CDC` by an infinite collection with odds `obese:skinny = 4:6`, and the result of `Doctor(skinny)` by an infinite collection with odds `burger:celery = 3:7`, and, respectively, that of `Doctor(obese)` by a collection with odds `burger:celery = 9:1`, then sampling from the infinite collection `Doctor*CDC`, which results from applying the prior to the likelihood, will have a ratio `(obese:burger):(obese,celery):(skinny,burger):(skinny,celery) = 36:4:18:24`.

The keen reader will note that $(*)$ is a slight variation of the well-known monadic bind operator, which, depending on your favorite programming language, is known under the names `(>>=)`, `SelectMany`, or `flatMap`. Indeed, *probability distributions form a monad*. Mathematicians

FIGURE 3: JOINT PROBABILITY DISTRIBUTION

P(FOOD & WEIGHT)	BURGER	CELERY	P(WEIGHT)
obese	$0.4 * 0.9 = 0.36$	$0.4 * 0.1 = 0.04$	$0.36 + 0.04 = 0.4$
skinny	$0.6 * 0.3 = 0.18$	$0.6 * 0.7 = 0.42$	$0.18 + 0.42 = 0.6$
P(food)	$0.36 + 0.18 = 0.54$	$0.04 + 0.42 = 0.46$	

call it the Giry monad, but Reverend Bayes beat everyone to it by nearly two centuries.

Note that as formulated, Bayes's rule has a type error that went unnoticed for centuries. The left-hand side returns a distribution of pairs $\mathbb{P}(A \& B)$, while the right-hand side returns a distribution of pairs $\mathbb{P}(B \& A)$. Not a big deal for mathematicians since $\&$ is commutative. For brevity we'll be sloppy about this as well. Since we often want to convert from $\mathbb{P}(A \& B)$ to $\mathbb{P}(A)$ or $\mathbb{P}(B)$ by dropping one side of the pair, we prefer the `C#` variant of `SelectMany` that takes a combiner function `A  $\oplus$  B  $\rightarrow$  C` to post-process the pair of samples from the prior and likelihood:

```
likelihood*prior =  
    from a→p in prior  
    from b→q in likelihood(a)  
    select a⊕b→p*q
```

Now that we know that `(*)` is monadic bind, we can start using syntactic sugar such as LINQ queries or `for/` monad comprehensions. All that is really saying is that it is safe to drop the explicit tracking of probabilities from any query written over distributions (i.e., the code on the left in figure 4 is simply sugar for the code on the right, which itself can be alternatively implemented with the frequentist approach using sampling).

Another way of saying this is that we can use query comprehensions as a DSL (domain-specific language) for specifying probabilistic functions. This opens the road to explore other standard query operators besides

application that can work over distributions and that can be added to our repertoire. The first one that comes to mind is filtering, or *projection* as the mathematicians prefer to call it.

Given a predicate $(A \rightarrow \mathbb{B})$, we can drop all values in a distribution for which the predicate does not hold using the division operator $(\div)$:

$\mathbb{P}(A) \div (A \rightarrow \mathbb{B}) \in \mathbb{P}(A)$
prior $\div$ condition = from a in prior where
condition(a) select a

The traditional division of a distribution $\mathbb{P}(A \& B)$ by distribution $\mathbb{P}(B)$ can be defined similarly as

joint $\div$ evidence =
 $\lambda b. \text{from } (a, b) \text{ in joint from } b' \text{ in evidence}$
where $b = b'$ select (a,b)

We can show that $(f * d) \div d = f$. Applying the latter version to Bayes's rule results in the following equivalence:

4

FIGURE 4: SYNTACTIC SUGAR

from a in prior	from $a \rightarrow p$ in prior
from b in likelihood [a]	from $b \rightarrow q$ in likelihood [a]
select $a \oplus b$	select $a \oplus b \rightarrow p^* q$

$$\mathbb{P}(A|B) = \mathbb{P}(B|A) * \mathbb{P}(A) \div \mathbb{P}(B)$$

In practice, it is most convenient to use query comprehensions directly instead of operators, and write code like this:

```
posterior ∈  $\mathbb{P}(C|B)=B \rightarrow \mathbb{P}(C)$ 
posterior(b) =
  from a in prior
  from b' in likelihood(a) where b = b'
  select a⊕b
```

Whichever way you spin it, this is incredibly cool! Bayes's rule shows how to invert a probabilistic function of type $\mathbb{P}(B|A) = A \rightarrow \mathbb{P}(B)$ into a probabilistic function of type $\mathbb{P}(A|B) = B \rightarrow \mathbb{P}(A)$ using conditioning.

When function inversion is applied to the running example, the probabilistic function $\text{PredictWeightFromFood} \in \mathbb{P}(\text{Weight}|\text{Food})$ can be defined as follows:

```
PredictWeightFromFood ∈  $\text{Food} \rightarrow \mathbb{P}(\text{Weight})$ 
PredictWeightFromFood(food) = (Doctor * CDC) ÷
  (_ = food)
```

Removing all syntactic sugar and using the value/ probability pairs implementation amounts to the following probabilistic function:

```

PredictWeightFromFood ∈ Food →  $\mathbb{P}(\text{Weight})$ 
PredictWeightFromFood(burger)
    = [obese ↦ 36, skinny ↦ 18]
PredictWeightFromFood(celery)
    = [obese ↦ 4, skinny ↦ 42]

```

In practice, most monads have an unsafe run function of type $\mathbb{P}(A) \rightarrow M(A)$ that teleports you out of the monad into some concrete container M . Mathematicians call this the *forgetful functor*. For distributions $\text{dist} \in \mathbb{P}(A)$, a common way to exit the monad is by picking the value $a \in A$ with the highest probability in dist . Mathematicians use the higher-order function `arg max` for this, and call it MLE (maximum likelihood estimator) or MAP (maximum *a posteriori*). In practice it is often more convenient to return the pair $a \mapsto p$ from dist with the highest probability.

A simple way to find the value with the maximal likelihood from a frequentist representation of a distribution is to blow up the source distribution $\mathbb{P}(A)$ into a distribution of distributions $\mathbb{P}(\mathbb{P}(A))$, where the outer distribution is an infinite frequentist list of inner Bayesian distributions $[A \rightarrow \mathbb{R}]$, computed by grouping and summing, that over time will converge to the true underlying distribution. Then you can select the n th inner distribution and take its maximum value.

```

WeightFromFood ∈ Food → [A → ℝ]
WeightFromFood food =
    PredictWeightFromFood(food).Run().
    ElementAt(1000)

```

Using the query-style DSL for composing and conditioning probabilistic functions is great, but it falls short of being a real programming language with arbitrary control flow, loops, try/catch blocks, recursion, etc. Since distributions are a variant of the continuation monad, it is possible to integrate probabilistic computations into a regular imperative language similar to the `async await` syntax that is now available in many programming languages. An example of an imperative *probabilistic programming language* is WebPPL (<http://webppl.org>), which embeds the distribution monad into regular JavaScript. In WebPPL, the running example looks as follows:

```
var cdc = function() {  
  return Categorical({ ps: [4, 6],  
vs: [ "obese", "skinny"] })  
}  
  
var doctor = function(weight) {  
  if("obese" == weight)  
    return Categorical({ ps: [9, 1],  
vs: [ "burger", "celery"] })  
  if("skinny" == weight)  
    return Categorical({ ps: [3, 7],  
vs: [ "burger", "celery"] })  
}  
  
var predict = function(food) {  
  var weight = sample(cdc())  
  var food_ = sample(doctor(weight))  
  condition(food == food_)  
  return weight  
}
```

The assignment + sample statement

```
var a = sample(prior)
... rest of the program...
is exactly like the query fragment
```

```
from a in prior
... rest of the query ...
```

and randomly picks a value $a \in A$ from a distribution $\text{prior} \in \mathbb{P}(A)$. The `condition(p)` statement corresponds to a `where` clause in a query.

To “run” this program, we pass the `predict` function into the WebPPL inference engine as follows:

```
Infer({method: enumerate, samples: 10000},
function(){return predict("burger")})
```

This samples from the distribution described by the program using the `Infer` function with the specified sampling method (which includes `enumerate`, `rejection`, and `MCMC`) that reifies the resulting distribution into a Bayesian representation.

APPLICATIONS OF PROBABILISTIC PROGRAMMING

Suppose ordinary developers had access to a probabilistic programming language. What scenarios would this open up?

If we take a step back and look at a typical web or mobile application, it implements the standard reinforcement

5

learning design pattern shown in figure 5. We have to predict an action to send to the user, based on the user's state and the dollar value extracted from the user, such that the sum of the rewards over time is maximized.

For games such as AlphaGo,¹⁰ the agent code is often a neural network, but if we abstract the pattern to apply to applications as a whole, it is likely a combination of ML learned models and regular imperative code. This hybrid situation is true even today, where things such as ad placement and search-result ranking are probabilistic but opaquely embedded into imperative code. Probabilistic programming and machine learning will allow developers to create applications that are highly specialized for each user.

One of the attractions of IDEs (integrated development environments) is autocomplete, where the IDE predicts what a user is going to type, based on what has been typed thus far. In most IDEs, autocomplete is driven by static type information. For example, if the user types `pp1`, the JetBrains Rider IDE shows all the properties of the `string` type as potential completions, as shown in figure 6.

Note that the completion list is shown in deterministic alphabetical order, rather than being probabilistically

FIGURE 5: **STANDARD REINFORCEMENT LEARNING DESIGN PATTERN**

ranked using some learned model based on which methods on `string` are the most likely in the given context. Hence, the IDE should implement autocomplete using a probabilistic function `autoComplete` $\in \mathbb{P}([\text{Completion}]|\text{Context})$ that returns a distribution of possible completions based on the current user context.⁷ Another recent application of ML and probabilistic programming in the compiler space is to infer pretty-print rules by learning from patterns in a large corpus of code `prettyPrint` $\in \mathbb{P}(\text{String}|\text{AST})$.⁵

For an example application of exposing the representation of distributions, let's revisit the feedback loop between user and application. Assume in this case that we want to determine the optimal title for this article that would maximize click-through on the *ACM Queue* website. That is, should we use “Probabilistic Programming for Dummies” instead of the current title “Making Money Using Math”?

In this case, we create the model shown in figure 7, the set of all users as a conditional distribution of a user clicking on the article given the title:

FIGURE 6: **AUTOCOMPLETE EXAMPLE**

6

$\text{user} \in \mathbb{P}(\text{Click}|\text{Title})$

Note we do not want to make any *a priori* assumptions about the underlying distributions other than the frequentist stream of clicks received, given the frequentist stream of titles served to the users.

The agent in this case wants to find out over time which possible title for a story will generate the most clicks from the users, and hence we will model the agent by the higher-order function that takes the users and from that creates a distribution of titles:

$\text{agent} \in (\text{Title} \rightarrow \mathbb{P}(\text{Click})) \rightarrow \mathbb{P}(\text{Title})$

Mathematicians call the implementation of `user` a *Bayesian bandit*,¹¹ and they leverage the fact that Bernoulli and beta distributions are conjugate priors.¹² They call the variant of the run function we will be using *Thompson sampling*.¹

When viewed from a computer scientist's point of view, the operational solution is relatively straightforward. We convert the user behavior that returns a distribution of clicks $\text{user} \in \text{Title} \rightarrow \mathbb{P}(\text{Click})$ into a function $\text{Title} \rightarrow$

FIGURE 7: SET OF ALL USERS AS A CONDITIONAL DISTRIBUTION

$\mathbb{P}(\text{Title} \& \text{Click} \mapsto \mathbb{R})$ that returns a distribution of pairs of titles and clicks using `run` as explained earlier (this corresponds to the beta distribution part of the algorithm. We do not track the “uncertainty” about $\mathbb{P}(\text{Click})$, but we can easily compute that together with the click probability if that is useful). A small tweak is needed in that we are interested only in clicks that are `true`, and not in those that are `false` (this is the Bernoulli part of the algorithm).

This allows us to observe how the probability that the user will click on each title evolves over time as we see more clicks from the users. Whenever we need to produce a new title, we use the `Title` for which the most recent `Title & Click  $\mapsto \mathbb{R}$` has the highest probability (this is the Thompson sampling part of the algorithm). In other words, the Bayesian bandit is essentially a merge sort over the reified underlying probability distributions of the clicks from the users.

The computational model underneath modern applications such as self-driving cars, speech and image recognition, personalized recommendations, etc. is changing from classical deterministic computations to probabilistic machine-learned models. Currently, building such applications requires deep knowledge and expertise of the underlying details of the ML algorithms using custom tools.

CONCLUSION

Probabilistic programming aims to democratize the application of machine learning by allowing regular programmers to incorporate ML in general-purpose

Related articles

➡ Information Extraction: Distilling
Structured Data from Unstructured Text
Andrew McCallum
<http://queue.acm.org/detail.cfm?id=1105679>

➡ Accountability in Algorithmic
Decision-making
Nicholas Diakopoulos
<http://queue.acm.org/detail.cfm?id=2886105>

➡ Learning from THE WEB
Adam Bosworth
<http://queue.acm.org/detail.cfm?id=1103833>

programming languages without friction. As illustrated in this article, from a semantics point of view, a probabilistic language simply adds the probability monad to the set of ambient side effects and leverages Bayes's rule to compose and condition probability distributions. Efficiently implementing probabilistic languages and providing the proper software engineering tooling, however, will keep compiler and programming-language experts busy for a long time.

References

1. Agrawal, S., Goyal, N. 2012. Analysis of Thompson sampling for the multi-armed bandit problem. *Journal of Machine Learning Research: Workshop and Conference Proceedings 23*; <http://jmlr.org/proceedings/papers/v23/agrawal12/agrawal12.pdf>.
2. Fink, D. 1997. A compendium of conjugate priors. Montana State University; <http://www.johndcook.com/CompendiumOfConjugatePriors.pdf>.
3. Goodman, N. D. 2013. The principles and practice of probabilistic programming. Proceedings of the 40th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages; <http://dl.acm.org/citation.cfm?id=2429117>.

4. Norvig, P. 2015. Machine learning for programming. InfoQueue; <https://www.infoq.com/presentations/machine-learning-general-programming>.
5. Parr, T., Vinju, J. 2016. Towards a universal code formatter through machine learning. Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering; <http://dl.acm.org/citation.cfm?id=2997383>.
6. Paulos, J. A. 2011. The mathematics of changing your mind. *New York Times Sunday Book Review*; http://www.nytimes.com/2011/08/07/books/review/the-theory-that-would-not-die-by-sharon-bertsch-mcgrayne-book-review.html?_r=0.
7. Raychev, V., Bielik, P., Vechev, M. 2016. Probabilistic model for code with decision trees. Proceedings of the ACM SIGPLAN International Conference on Object-oriented Programming, Systems, Languages, and Applications; <http://dl.acm.org/citation.cfm?doid=2983990.2984041>.
8. Schwarz, K. 2011. Darts, dice, and coins: sampling from a discrete distribution; <http://www.keithschwarz.com/darts-dice-coins/>.
9. Ścibior, A., Ghahramani, Z., Gordon, A. D. 2015. Practical probabilistic programming with monads. Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell; <http://dl.acm.org/citation.cfm?id=2804317>.
10. Silver, D., et al. 2016. Mastering the game of Go with deep neural networks and tree search; <https://blog.acolyer.org/2016/09/20/mastering-the-game-of-go-with-deep-neural-networks-and-tree-search/>.

11. Stucchio, C. 2013. Bayesian bandits—optimizing click-throughs with statistics; https://www.chrisstucchio.com/blog/2013/bayesian_bandit.html.
12. Wikipedia. Conjugate prior: discrete distributions; https://en.wikipedia.org/wiki/Conjugate_prior#Discrete_distributions.

Erik Meijer *has been working on “democratizing the cloud” for the past 15 years. He is perhaps best known for his work on, among others, the Haskell, C#, Visual Basic, and Dart programming languages and his contributions to LINQ and the Reactive Framework (Rx).*

Copyright © 2017 held by owner/author. Publication rights licensed to ACM.

SHAPE THE FUTURE OF COMPUTING!

Join ACM today at acm.org/join

**BE CREATIVE. STAY CONNECTED.
KEEP INVENTING.**

Association for
Computing Machinery

Advancing Computing as a Science & Profession