Лекція 7.

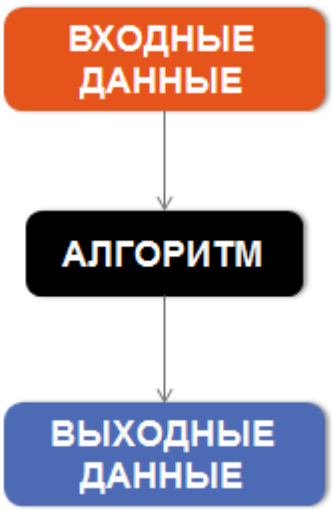
ЙМОВІРНІСНЕ ПРОГРАМУВАННЯ

Вероятностное программирование можно определять как ***компактный,композиционный*** способ представления ***порождающих вероятностных моделей*** и проведения ***статистического вывода*** в них с учетом данных с помощью обобщенных алгоритмов.

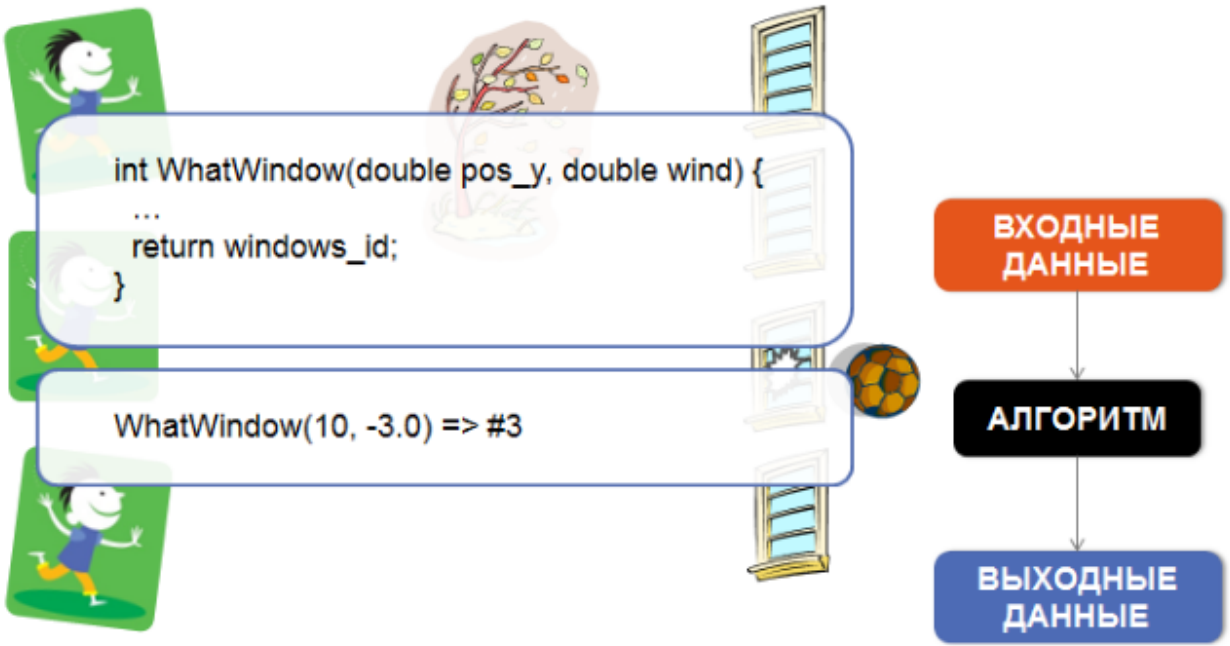
«Обычное» программирование

Для знакомства с вероятностным программирование давайте сначала поговорим об «обычном» программировании. В «обычном» программировании основой является алгоритм, обычно детерминированный, который позволяет нам из входных данных получить выходные по четко установленным правилам.

«ОБЫЧНОЕ» ПРОГРАММИРОВАНИЕ



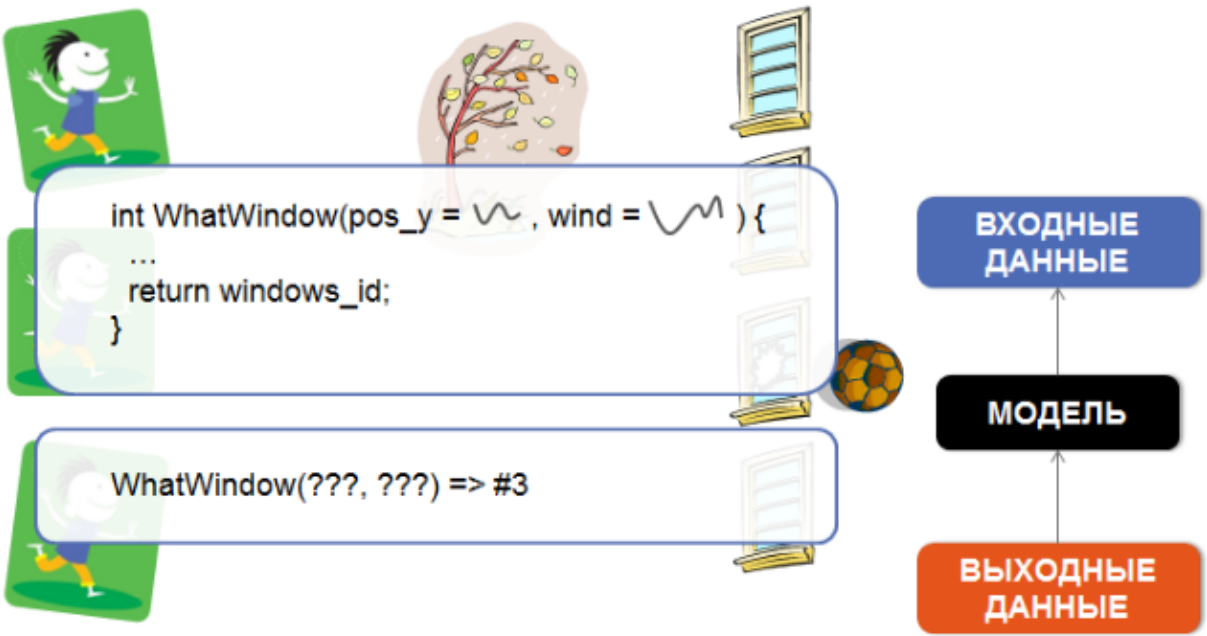
Например, если у нас есть мальчик Вася, и мы знаем где он находится, куда он бросает мяч и каковы внешние условия (например, сила ветра), мы узнаем, какое окно он, к сожалению, разобьет в здании школы. Для этого достаточно симулировать простые законы школьной физики, которые легко можно записать в виде алгоритма.


int WhatWindow(double pos_y, double wind) {
...
return windows_id;
}
WhatWindow(10, -3.0) => #3
ВХОДНЫЕ ДАННЫЕ
АЛГОРИТМ
ВЫХОДНЫЕ ДАННЫЕ

А теперь вероятностное программирование

Однако часто мы знаем только результат, исход, и мы заинтересованы в том, чтобы узнать то, какие неизвестные значения привели именно к этому результату? Чтобы ответить на этот вопрос с помощью теории математического моделирования создается вероятностная модель, часть параметров которой не определены точно.

Например, в случае с мальчиком Васей, зная то, какое окно он разбил, и имея априорные знания о том, около какого окна он и его друзья обычно играют в футбол, и зная прогноз погоды на этот день, мы хотим узнать апостериорные распределение местоположения мальчика Васи: откуда же он бросал мяч?


int WhatWindow(pos_y = , wind =) {
...
return windows_id;
}
WhatWindow(???, ???) => #3
ВХОДНЫЕ ДАННЫЕ
МОДЕЛЬ
ВЫХОДНЫЕ ДАННЫЕ

Итак, зная выходные данные, мы заинтересованы в том, чтобы узнать наиболее вероятные значения скрытых, неизвестных параметров.

В рамках машинного обучения рассматриваются в том числе *порождающие*вероятностные модели. В рамках порождающих вероятностных моделей модель описывается как алгоритм, но вместо точных однозначных значений скрытых параметров и некоторых входных параметров мы используем вероятностные распределениях на них.

Существует более 15 языков вероятностного программирования.

В этой лекции приведен пример кода на вероятностных языках Venture/Anglican, который имеют очень схожий синтаксис и которые берут свое начало от вероятностного языка Church.

Church в свою очередь основан на языке «обычного» программирования Lisp и Scheme.

ІСНУЮЧІ СИСТЕМИ ЙМОВІРНІСНОГО ПРОГРАМУВАННЯ

Ниже представлен список вероятностных систем программирования, включая языки, реализации / компиляторs, а также программных библиотек для построения вероятностных моделей и наборов инструментальных средств для построения вероятностных алгоритмов логического вывода.

- Alchemy is программный пакет обеспечивает ряд алгоритмов для статистического реляционного обучения и вероятностного логического вывода, основанный на Марковской логике представлении.
- Anglican is портативный, полный по Тюрингу вероятностный язык программирования, который включает в себя части MCMC (Markov chain Monte Carlo) вывода.

- BLOG, or Bayesian logic, is a probabilistic programming language with elements of first-order logic, as well as an MCMC-based inference algorithm. BLOG makes it relatively easy to represent uncertainty about the number of underlying objects explaining observed data.

- BUGS is a language for specifying finite graphical models and accompanying software for performing B(ayesian) I(nference) U(sing) G(ibbs) S(ampling), although modern implementations (such as WinBUGS, JAGS, and OpenBUGS) are based on Metropolis-Hastings. BiiPS is an implementation based on interacting particle systems methods like Sequential Monte Carlo.

- Church is a universal probabilistic programming language, extending Scheme with probabilistic semantics, and is well suited for describing infinite-dimensional stochastic processes and other recursively-defined generative processes (Goodman, Mansinghka, Roy, Bonawitz and Tenenbaum, 2008). The active implementation of Church is webchurch. Older implementations include MIT-Church, Cosh, Bher, and JSChurch. See also Venture below.

- Dimple is a software tool that performs inference and learning on probabilistic graphical models via belief propagation algorithms or sampling based algorithms.

- FACTORIE is a Scala library for creating relational factor graphs, estimating parameters and performing inference.

- Figaro is a Scala library for constructing probabilistic models that also provides a number of built-in reasoning algorithms that can be applied automatically to any constructed models.

- HANSEI is a domain-specific language embedded in OCaml, which allows one to express discrete-distribution models with potentially infinite support, perform exact inference as well as importance sampling-based inference, and model inference over inference.

- Hakaru is a simply-typed probabilistic programming language which offers composable and pluggable inference engines as well as computer-algebra guided program optimization.

- Hierarchical Bayesian Compiler (HBC) is a language for expressing and compiler for implementing hierarchical Bayesian models, with a focus on large-dimension discrete models and support for a number of non-parametric process priors.

- LibBi is a library for Bayesian inference, based largely on the sequential Monte Carlo framework.

- Markov the Beast is a software package for statistical relational learning and structured prediction based on Markov logic.

- PRISM is a general programming language intended for symbolic-statistical modeling, and the PRISM programming system is a tool that can be used to learn the parameters of a PRISM program from data, e.g., by expectation-maximization.

- Infer.NET is a software library developed by Microsoft for expressing graphical models and implementing Bayesian inference using a variety of algorithms.

- PRAiSE is a system performing probabilistic inference without grounding random variables or sampling, but instead performing it exactly and directly over an expressive higher-level language involving difference arithmetic over integers, linear real arithmetic, equality over categorical types, with relational random variables and algebraic data types coming next.

- Probabilistic-C is a C-language probabilistic programming system that, using standard compilation tools, automatically produces a compiled parallel inference executable from C-language generative model code.

- ProbLog is a probabilistic extension of Prolog based on Sato's distribution semantics. While ProbLog1 focuses on calculating the success probability of a query, ProbLog2 can calculate both conditional probabilities and MPE states.

- Probabilistic Soft Logic is a machine learning framework for developing probabilistic models. Models are defined using a straightforward logical syntax and solved via convex optimization.

- PyMC is a python module that implements a suite of MCMC algorithms as python classes, and is extremely flexible and applicable to a large suite of problems. PyMC includes methods for summarizing output, plotting, goodness-of-fit and convergence diagnostics.

- PyMLNs is a toolbox and software library for learning and inference in Markov logic networks.

- R2 is a probabilistic programming system that employs powerful techniques from programming language design, program analysis and verification for scalable and efficient inference.

- Stan exposes a language for defining probability density functions for probabilistic models. Stan includes a compiler, which produces C++ code that performs Bayesian inference via a method similar to Hamiltonian Monte Carlo sampling.

- Tuffy is a highly scalable inference engine for Markov logic networks using a database backend.

- Venture is an interactive, Turing-complete, higher-order probabilistic programming platform that aims to be sufficiently expressive, extensible and efficient for general-purpose use. Its virtual machine supports multiple scalable, reprogrammable inference strategies, plus two front-end languages: VenChurch and VentureScript.

- WebPPL A PPL based on and implemented in Javascript. Include MH, SMC, variational, and other experimental inference engines.

Пример Байесовской линейной регрессии

Рассмотрим задание простой вероятностной модели Байесовской линейной регрессии на языке вероятностного программирования Venture/Anglican в виде вероятностной программы:

```
01: [ASSUME t1 (normal 0 1)]
02: [ASSUME t2 (normal 0 1)]
03: [ASSUME noise 0.01]
04: [ASSUME noisy_x (lambda (time) (normal (+ t1 (* t2 time)) noise))]
05: [OBSERVE (noisy_x 1.0) 10.3]
06: [OBSERVE (noisy_x 2.0) 11.1]
07: [OBSERVE (noisy_x 3.0) 11.9]
08: [PREDICT t1]
09: [PREDICT t2]
10: [PREDICT (noisy_x 4.0)]
```

Скрытые искомые параметры — значения коэффициентов **t1** и **t2** линейной функции **x = t1 + t2 * time**. У нас есть априорные предположения о данных коэффициентах, а именно мы предполагаем, что они распределены по закону нормального распределения **Normal(0, 1)** со средним 0 и стандартным отклонением 1. Таким образом, мы определили в первых двух строках вероятностной программы априорную вероятность на скрытые переменные, **P(T)**. Инструкцию **[ASSUME name expression]**можно рассматривать как определение случайной величины с именем **name**, принимающей значение вычисляемого выражение (программного кода) **expression**, которое содержит в себе неопределенность.

Вероятностные языки программирования (имеются в виду конкретно Church, Venture, Anglican), как и Lisp/Scheme, являются функциональными языками программирования, и используют польскую нотацию при записи выражений для вычисления. Это означает, что в выражении вызова функции сначала располагается оператор, а уже только потом аргументы: **(+ 1 2)**, и вызов функции обрамляется круглыми скобками. На других языках программирования, таких как C++ или Python, это будет эквивалентно коду **1 + 2**.

В вероятностных языках программирования выражение вызова функции принято разделять на три разных вида:

- Вызов детерминированных процедур **(primitive-procedure arg1... argN)**, которые при одних и тех же аргументах всегда возвращают одно и то же значение. К таким процедурам, например, относятся арифметические операции.

- Вызов вероятностных (стохастических) процедур **(stochastic-procedure arg1… argN)**, которые при каждом вызове генерируют случайным образом элемент из соответствующего распределения. Такой вызов определяет новую *случайную величину*. Например, вызов вероятностной процедуры **(normal 1 10)**определяет случайную величину, распределенную по закону нормального распределения **Normal(1, sqrt(10))**, и результатом выполнения каждый раз будет какое-то вещественное число.
- Вызов составных процедур **(compound-procedure arg1… argN)**, где**compound-procedure** — введенная пользователем процедура с помощью специального выражения **lambda**: **(lambda (arg1… argN) body)**, где **body** — тело процедуры, состоящее из выражений. В общем случае составная процедура является стохастической (недетерминированной) составной процедурой, так как ее тело может содержать вызовы вероятностных процедур.

Вернемся к исходному коду на языке программирования Venture/Anglican. После первых двух строк мы хотим задать условную вероятность **P(X | T)**, то есть условную вероятность наблюдаемых переменных **x1**, **x2**, **x3** при заданных значениях скрытых переменных **t1**, **t2** и параметра **time**.

Перед вводом непосредственно самих наблюдений с помощью выражения **[OBSERVE ...]** мы определяем общий закон для наблюдаемых переменных **xi** в рамках нашей модели, а именно мы предполагаем, что данные наблюдаемые случайные величины при заданных **t1**, **t2** и заданном уровне шума **noise** распределены по закону нормального распределения **Normal(t1 + t2 * time, sqrt(noise))** со средним **t1 + t2 * time** и стандартным отклонением **noise**. Данная условная вероятность определена на строках 3 и 4 данной вероятностной программы. **noisy_x** определена как

функция, принимающая параметр **time** и возвращающая случайное значение, определенное с помощью вычисления выражение **(normal (+ t1 (* t2 time)) noise)** и обусловленное значениями случайных величин **t1** и **t2** и переменной **noise**. Отметим, что выражение **(normal (+ t1 (* t2 time)) noise)** содержит в себе неопределенность, поэтому каждый раз при его вычислении мы будем получать в общем случае разное значение.

На строках 5—7 мы непосредственно вводим известные значения **x1 = 10.3**, **x2 = 11.1**, **x3 = 11.9**. Инструкция вида **[OBSERVE expression value]** фиксирует наблюдение о том, что случайная величина, принимающая значение согласно выполнению выражения **expression**, приняла значение **value**.

Повторим на данном этапе всё, что мы сделали. На строках 1—

4 с помощью инструкций вида **[ASSUME ...]** мы задали непосредственно саму вероятностную модель: **P(T)** и **P(X | T)**. На строках 5—7 мы непосредственно задали известные нам значения наблюдаемых случайных величин **X** с помощью инструкций вида **[OBSERVE ...]**.

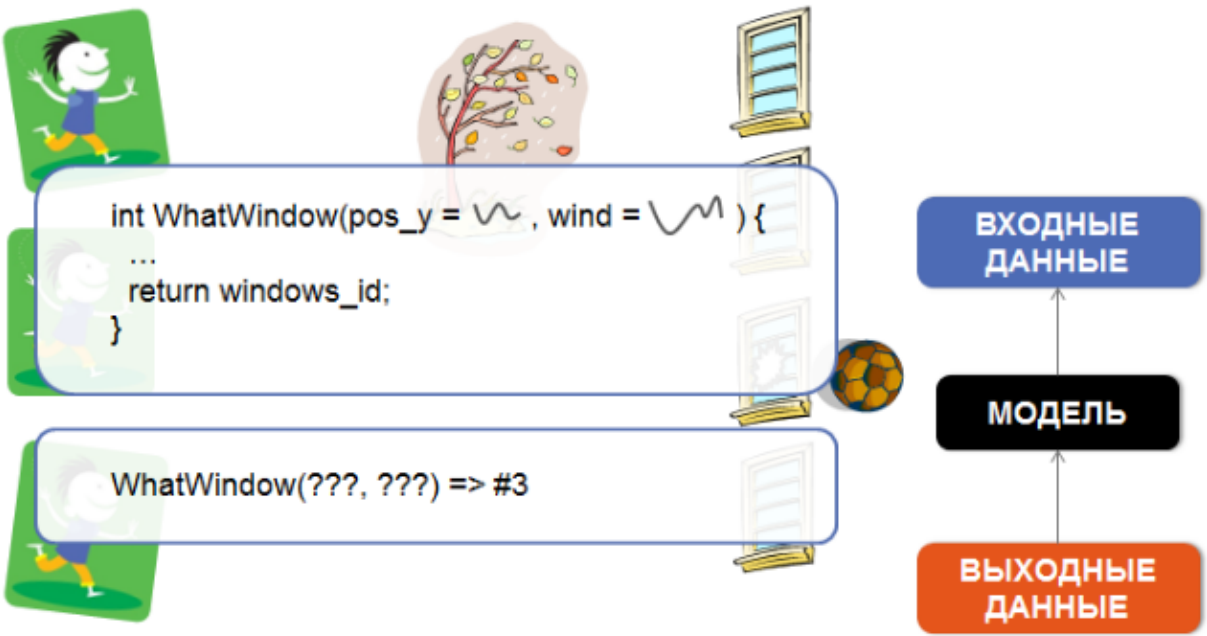
На строках 8—9 мы запрашиваем у системы вероятностного программирования апостериорное распределение **P(T | X)** скрытых случайных величин **t1** и **t2**. Как уже было сказано, при большом объеме данных и достаточно сложных моделях получить точное аналитическое представление невозможно, поэтому инструкции вида**[PREDICT ...]** генерируют выборку значений случайных величин из апостериорного распределения **P(T | X)** или его приближения. Инструкция вида **[PREDICT expression]** в общем случае генерирует один элемент выборки из значений случайной величины, принимающей значение согласно

выполнению выражения**expression**. Если перед инструкциями вида **[PREDICT ...]** расположены инструкции вида **[OBSERVE ...]**, то выборка будет из апостериорного распределения (говоря точнее, конечно, из приближения апостериорного распределения), обусловленного перечисленными ранее введенными наблюдениями.

Отметим, что в завершении мы можем также предсказать значение функции **x(time)** в другой точке, например, при **time = 4.0**. Под предсказанием в данном случае понимается генерация выборки из апостериорного распределения новой случайной величины при значениях скрытых случайных величин **t1**, **t2** и параметре **time = 4.0**.

Для генерации выборки из апостериорного распределения **P(T | X)** в языке программирования Venture в качестве основного используется алгоритм Метрополиса-Гастингса, который

относится к методам Монте-Карло по схеме Марковских цепей. По*добобщенным* выводом в данном случае понимается то, что алгоритм может быть применен к любым вероятностным программам, написанным на данном вероятностном языке программирования.


int WhatWindow(pos_y = , wind =) {
...
return windows_id;
}
WhatWindow(???, ???) => #3
ВХОДНЫЕ ДАННЫЕ
МОДЕЛЬ
ВЫХОДНЫЕ ДАННЫЕ

Уже более полутора лет назад прошла новость о том, что «DARPA намерено совершить революцию в машинном обучении». Конечно, DARPA всего лишь выделила деньги на исследовательскую программу, связанную с вероятностным программированием. Само же вероятностное программирование существует и развивается без DARPA достаточно давно, причем исследования ведутся, как в ведущих университетах, таких как MIT, так и в крупных корпорациях, таких как Microsoft. И вовсе не зря DARPA, Microsoft, MIT и т.д. обращают пристальное внимание на эту область, ведь она по-настоящему перспективна для машинного обучения, а, может, и для искусственного интеллекта в целом. Говорят, что вероятностное программирование для машинного обучения будет играть ту же роль, что и высокоуровневые языки для обычного программирования. Мы бы привели другую параллель – с ролью Пролога, которую он сыграл для старого доброго ИИ.

Вот только в Рунете по данной теме до сих пор можно найти лишь единичные ссылки, и то в основном содержащие лишь описания общих принципов. Возможно, это связано с тем, что потенциал вероятностного программирования еще только начал раскрываться и оно не стало основным трендом. Однако на что же способны или будут способны вероятностные языки?

Можно выделить два основных класса вероятностных языков программирования – это языки, допускающие задание генеративных моделей только в форме Байесовских сетей (или других графических вероятностных моделей), или Тьюринг-полные языки.

Типичным представителем первых является Infer.NET, разрабатываемый в Microsoft. В нем благодаря использованию в качестве генеративных моделей Байесовких сетей оказывается возможным применять известные для них эффективные методы вывода. Естественно, использование хорошо известного класса моделей с известными методами вывода не приводит к возможности решения каких-то принципиально новых задач (и даже такие генеративные модели, как сети глубокого обучения на основе ограниченных машинах Больцмана оказываются не представимы в таких языках), но дает вполне практичный инструмент.

Как говорят разработчики, с использованием этого инструмента можно за пару часов реализовать нетривиальную вероятностную модель, такую как полная Байесовская версия анализа главных компонент, которая будет занимать всего пару десятков строк кода и для которой отдельная реализация эффективной процедуры вывода на обычном языке потребовала бы заметно большего объема знаний и нескольких недель работы. Таким образом, за счет вероятностного программирования использование графических моделей становится гораздо более простым и доступным.

Гораздо большим потенциалом, однако, обладают Тьюринг-полные вероятностные языки. Они позволяют выйти за рамки того класса задач, которые существующие методы машинного обучения уже умеют решать. Естественно, в таких языках возникает проблема эффективности вывода, которая пока далека от решения, что приводит к плохой масштабируемости на задачи реального мира. Однако это направление активно развивается, и существует ряд работ, показывающих как в вероятностных языках общего назначения достичь эффективного вывода для интересных практических задач. Можно надеяться, что в ближайшем будущем эти решения станут доступными для использования в конкретных языках.

Кроме того, Тьюринг-полные вероятностные языки уже сейчас оказываются весьма полезными в исследованиях, связанных с когнитивным моделированием и общим искусственным интеллектом. По этим причинам мы и рассмотрим основные принципы вероятностного программирования именно на примере Тьюринг-полных языков, из которых мы выбрали Чёрч (Church), являющийся расширением языка Лисп (конкретнее, его диалекта – Scheme). Удобство этого языка (по крайней мере, в целях начального знакомства с ним) заключается в существовании для него web-реализации (web-church), с которой можно экспериментировать без установки дополнительного программного обеспечения.

Программа на вероятностном языке может, на первый взгляд, ничем не отличаться от программы на обычном языке. Именно так сделано в Чёрче. Как и в обычном Лиспе, в этом языке могут быть определены переменные, функции, выполнены детерминированные вычисления. Например, следующая программа задает функцию от одного аргумента, вычисляющую факториал по рекурсивной формуле n!=n*(n–1)!, и вызывает эту функцию для n=10

```
(define (f n)
  (if (= n 0) 1 (* n (f (- n 1)))))
(f 10)
```

Также в этом языке могут быть обращения к (псевдо)случайным функциям. Например, при выполнении вызова (flip 0.3) с вероятностью 0.3 будет возвращено значение #t, а с вероятностью 0.7 – #f. Такая функция элементарно реализуется и в Лиспе как

```
(define (flip p) (< (random) p))
```

Чёрч, как и другие вероятностные языки, включает много встроенных функций, которые возвращают случайные значения в соответствии с тем или иным распределением. Например, (gaussian x0 s) возвращает вещественную случайную величину, распределенную по гауссиане с заданными параметрами. В качестве других реализованных распределений вероятностей обычно присутствуют равномерное, мультиномиальное, Дирихле, бета, гамма. Все эти распределения не так сложно реализовать вручную в обычном языке, и здесь пока нет принципиального отличия между Чёрчем и Лиспом.

Однако помимо обычной семантики программа на Чёрче обладает вероятностной семантикой, в рамках которой полагается, что программа, содержащая вызовы случайных функций, не просто при своем запуске порождает какие-то конкретные значения случайных величин, но задает распределение вероятностей над ними. Так, (gaussian x0 s) – это не просто функция, возвращающая некоторое конкретное значение случайной величины, распределенной по гауссиане, но именно само Гауссово распределение.

Но как получать эти распределения вероятностей, задаваемые программой? Представим, например, программу

```
(if (flip 0.4) (flip 0.1) (flip 0.6))
```

То есть с вероятностью 0.4 значение этого выражения – это P(#t)=0.1 и P(#f)=0.9, а с вероятностью 0.6 – P(#t)=0.6 и P(#f)=0.4. Откуда возьмется итоговое распределение, задаваемое этим выражением: P(#t)=0.4 и P(#f)=0.6? Эта вероятностная семантика зачастую реализуется через процесс сэмплирования: мы можем просто много раз запустить программу и построить выборку результатов ее выполнения. Такую процедуру, конечно, также несложно реализовать на обычном языке (и, действительно, еще Симула-67 таким способом регулярно использовалась для моделирования стохастических процессов).

Однако современные вероятностные языки идут дальше и добавляют в процесс сэмплирования условие, накладываемое на результаты выполнения программы. Эта идея ведет к простейшему сэмплированию с отказами, которая в Чёрче реализуется функцией rejection-query. Эта функция на вход принимает вероятностную программу (как совокупность define), предпоследнее выражение в которой вычисляет возвращаемое значение, а последнее выражение – это условие (предикат), который в процессе выполнения должен оказаться истинным.

Рассмотрим программу

```
(rejection-query
  (define A (flip 0.4))
  (define B (flip 0.6))
  B
  (or A B))
```

rejection-query выполняет поданную ей программу до тех пор, пока не будет выполнено последнее условие – здесь (or A B) – и возвращает (один раз) значение предпоследнего выражения – здесь B. Чтобы получить выборку значений, можно воспользоваться функцией repeat. Также Чёрч имеет встроенные функции для построения гистограмм.

Рассмотрим немного расширенную программу:

```
(define (get-sample) (rejection-query
  (define A (flip 0.4))
  (define B (flip 0.6))
  B
  (or A B)))
(hist (repeat 1000 get-sample))
```

При запуске мы получим следующий результат: #f — 21%, #t — 79% (цифры от запуска к запуску могут немного меняться). Этот результат означает, что значение B равно #t с вероятностью чуть меньше 0.8. Откуда взялась эта вероятность, если в программе B – это бинарная случайная величина, для которой P(#t)=0.6? Очевидно, дело в наложении условия: (or A B).

В процессе сэмплирования мы принимаем только такие значения B, что верно или A, или само B. Фактически, мы считаем апостериорную вероятность P(B|A+B). Можно было бы воспользоваться правилом Байеса для того, чтобы вычислить эту вероятность вручную:

$$P(B|A+B) = P(A+B|B)P(B)/P(A+B) =$$
$$=(P(A|B)+P(B|B)-P(A|B)P(B|B))P(B)/(P(A)+P(B)-P(A)P(B))=$$
$$=(P(A)+1-P(A))P(B)/(P(A)+P(B)-P(A)P(B))=0.6/(0.4+0.6-0.4*0.6)=0.789.$$

Однако уже для такой элементарной программы ручное применение правила Байеса требует некоторого времени, а для нетривиальных программ аналитически вычислить значения может и вовсе не получиться.

Итак, сэмплирование позволяет нам вычислять апостериорные вероятности интересующих нас случайных величин при наложении тех или иных условий. Оно заменяет правило Байеса, широко используемое в машинном обучении для выбора моделей или выполнения предсказаний. При этом запись программы на вероятностном языке для многих людей может оказаться гораздо понятнее, чем применение правила Байеса. Конечно, само режекторное сэмплирование весьма просто можно реализовать на обычном языке программирования, но вероятностные языки этим не ограничиваются.

В Чёрче, в частности, реализована другая функция для сэмплирования – enumeration-query. Запустим программу

```
(enumeration-query
  (define A (flip 0.4))
  (define B (flip 0.6))
  B
  (or A B))
```

На выходе мы получим: ((#t #f) (0.7894736842105263 0.2105263157894737)).

Здесь выведены точные значения (конечно, со скидкой на конечную разрядную сетку) вероятностей P(B|A+B). enumeration-query уже не просто запускает много раз программу, но анализирует пути ее выполнения и перебирает все возможные значения случайных переменных с учетом их вероятностей. Конечно, такое «сэмплирование» будет работать, только когда множество возможных комбинаций значений случайных переменных не слишком велико.

Есть в Чёрче и более продвинутая замена режекторному сэмплированию на основе MCMC (Monte Carlo Markov Chains), а именно Metropolis Hastings алгоритм, откуда и название у процедуры – mh-query. Эта процедура запроса сразу формирует заданное число сэмплов (а также получает на вход один дополнительный параметр – лаг). Эта процедура также нетривиальна в реализации, так что использование готового вероятностного языка (а не собственная реализация простых процедур сэмплирования на обычном языке) приобретает смысл.

Однако главное, что дает вероятностное программирование, – это стиль мышления.

От азов к применению

Разные разработчики находят разные применения вероятностному программированию. Многие применяют его непосредственно для решения задач машинного обучения. Авторы же языка Чёрч, Noah D. Goodman and Joshua B. Tenenbaum, в своей web-книге «Probabilistic Models of Cognition» показывают применение вероятностного программирования для когнитивного моделирования.

Также известно, как решение задач планирования удобно представлять в терминах вывода в вероятностных языках. Оно также оказывается применимым для представления знаний и вывода над ними, а также для задач машинного восприятия (в том числе, распознавания изображений). Все эти приложения пока более или менее разрозненные, но наличие общего фреймворка для всех них свидетельствует о том, что вероятностное программирование может стать «теорией великого объединения» для ИИ. Посмотрим на простейшие примеры возможного использования.

Одним из наиболее классических примеров применения экспертных систем является медицинская диагностика. В частности, система MYCIN была построена на системе правил вида:

Rule 52:
If
THE SITE OF THE CULTURE IS BLOOD

1. THE GRAM OF THE ORGANISM I S NEG
2. THE MORPHOLOGY OF THE ORGANISM IS ROD
3. THE BURN OF THE PATIENT IS SERIOUS

Then there is weakly suggestive evidence (0.4) that
THE IDENTITY OF THE ORGANISM IS PSEUDOMONAS

Очевидно, правила такого вида хорошо описываются на языке типа Чёрч. При этом нет необходимости еще и реализовывать процедуру вывода – достаточно просто записать систему правил. Приведем пример из упомянутой книги «Probabilistic Models of Cognition»:

```
(define samples
  (mh-query 1000 100
    (define lung-cancer (flip 0.01))
    (define TB (flip 0.005))
    (define cold (flip 0.2))
    (define stomach-flu (flip 0.1))
    (define other (flip 0.1))
    (define cough (or (and cold (flip 0.5)) (and lung-cancer
(flip 0.3)) (and TB (flip 0.7)) (and other (flip 0.01))))
    (define fever (or (and cold (flip 0.3)) (and stomach-flu
(flip 0.5)) (and TB (flip 0.2)) (and other (flip 0.01))))
    (define chest-pain (or (and lung-cancer (flip 0.4)) (and TB
(flip 0.5)) (and other( flip 0.01))))
    (define shortness-of-breath (or (and lung-cancer (flip 0.4))
(and TB (flip 0.5)) (and other (flip 0.01))))
    (list lung-cancer TB)
    (and cough fever chest-pain shortness-of-breath)))
(hist samples "Joint inferences for lung cancer and TB")
```

В этой программе определяются априорные вероятности появления у больного рака легких, туберкулеза, простуды и т.д. Далее определяются вероятности наблюдения кашля, жара, боли в груди и стесненного дыхания при тех или иных заболеваниях. Возвращаемая величина – это пара булевых значений, есть ли у пациента рак и/или туберкулез. И, наконец, условие – это совокупность наблюдаемых симптомов (то есть сэмплирование производится при условии, что значение всех переменных – cough fever chest-pain shortness-of-breath – #t).

Результат выполнения программы будет иметь следующий вид: (#f #f) — 4%, (#f #t) — 58%, (#t #f) — 37%, (#t #t) — 1%.

Несложно сделать, чтобы samples была функцией, в которую подается перечень симптомов, который далее в mh-query используется для сэмплирования, что даст возможность ставить диагнозы разным пациентам. Конечно, этот пример сильно упрощенный, но видно, что в стиле вероятностного программирования вполне можно представлять знания и делать вывод над ними.

Естественно, можно решать и задачи машинного обучения. Их отличие будет лишь в том, что неизвестными величинами будут параметры самой модели, а в качестве условия для сэмплирования будет выступать генерация этой моделью обучающей выборки. К примеру, в представленной выше программе мы бы числа в строках вида (define lung-cancer (flip 0.01)) могли бы заменить на переменные, которые сами бы задавались как случайные, например (define p-lung-cancer (uniform 0 1)), а далее для каждого пациента из обучающей выборки значение lung-cancer уже определялось бы с вероятностью p-lung-cancer.

Рассмотрим эту возможность на простом примере оценивания параметров многочлена по набору точек. В следующей программе функция calc-poly вычисляет значение многочлена с параметрами ws в точке x. Функция generate применяет calc-poly к каждому значению из заданного списка xs и возвращает список соответствующих ординат. Процедура noisy-equals? «приближенно» сравнивает два заданных значения (если эти значения равны, то функция возвращает #t с вероятностью 1; если же они не равны, то чем больше они отличаются, тем с меньшей вероятностью она вернет #t).

```
(define (calc-poly x ws)
  (if (null? ws) 0
      (+ (car ws) (* x (calc-poly x (cdr ws))))))

(define (generate xs ws)
  (map (lambda (x) (calc-poly x ws)) xs))

(define (noisy-equals? x y)
  (flip (exp (* -3 (expt (- x y) 2)))))

(define (samples xs ys)
  (mh-query 1 100
            (define n-coef 4)
            (define ws (repeat n-coef (lambda ()
(gaussian 0 3))))
            ws
            (all (map noisy-equals? (generate xs ws)
ys))))
(samples '(0 1 2 3 4) '(0.01 1.95 6.03 12.01 20.00))
```

Внутри вызова mh-query параметр n-coef определяет число коэффициентов в многочлене (то есть его степень плюс один); ws – это список, состоящий из случайных величин, сгенерированных в соответствии с нормальным распределением. Возвращаемое значение – список параметров многочлена. Условие для сэмплирования – «приближенное» равенство всех заданных значений ys всем ординатам, порожденным многочленом при данных ws. Здесь мы запрашиваем всего одну реализацию, которая проходит по условию (поскольку строить гистограмму для вектора параметров не очень удобно). Результатом этого запроса может быть, к примеру, список (2.69 1.36 0.53 -0.10), задающий многочлен 2.69+1.36x+0.53x^2–0.10x^3.

Вообще, вывод на моделях с вещественными параметрами – не самая сильная сторона языка Чёрч (но не стоит это считать глобальным недостатком вероятностного программирования вообще). Тем не менее, на этом примере mh-query кое-как работает. Чтобы в этом убедиться, вместо определения значений параметров в запросе можно попросить возвращать предсказание в некоторой точке. Перепишем последний фрагмент кода так:

```
(define (samples xs ys)
  (mh-query 100 100
            (define n-coef 4)
            (define ws (repeat n-coef (lambda ()
(gaussian 0 3))))
            (calc-poly 5 ws)
            (all (map noisy-equals? (generate xs ws)
ys))))
(hist (samples '(0 1 2 3 4) '(0.01 1.95 6.03 12.01
20.00)))
```

То есть мы запрашиваем наиболее вероятное (при имеющихся данных) значение в точке x=5. При разных запусках максимум гистограммы, к сожалению, будет приходиться на несколько различающиеся значения (метод MCMC, теоретически, гарантирует схождение к истинному распределению, но лишь в пределе), но обычно эти значения будут достаточно вразумительными. Стоит заметить, что здесь мы «бесплатно» (заменой одной строчки) получили полное байесовское предсказание: вместо выбора лучшей модели и предсказания лишь по ней, мы получили апостериорное распределение значений в точке x=5, усредненное сразу по множеству моделей с учетом их собственных вероятностей.

Но и это еще не все. Опять же, заменой одной строчки – (define n-coef 4) -> (define n-coef (random-integer 5)) мы можем сделать автоматический выбор между моделями с разным числом параметров. Причем сэмплирование величины n-coef показывает (хотя и не очень стабильно), что наиболее вероятным значением является n-coef=3 (то есть парабола, которая и заложена в заданный набор точек). При такой модификации более стабильным становится и предсказание. Иными словами, не возникает и эффекта переобучения!

Почему же не выбираются многочлены более высокой степени, ведь они могут точнее проходить к заданным точкам? Дело в том, что при сэмплировании «угадать» подходящие значения параметров многочлена меньшей степени проще, чем многочлена более высокой степени, поэтому вероятность породить такие параметры, которые пройдут проверку, для многочлена второй степени выше, чем для третьей. В то же время, многочлен первой степени будет давать большие отклонения, для которых вероятность срабатывания noisy-equals? будет сильно понижаться.

Посмотрим еще на одно применение, которое в рамках вероятностного программирования может показаться неожиданным. Это решение «дедуктивных» задач. Возьмем приведенную в начале функцию вычисления факториала, но вместо вызова ее с фиксированным значением будем считать, что аргумент – случайная переменная, но на само значение факториала наложено ограничение:

```
(define (f n)
  (if (= n 0) 1 (* n (f (- n 1)))))
(enumeration-query
 (define n (random-integer 20))
 n
 (equal? (f n) 120))
```

В качестве ответа мы увидим n=5 с вероятностью 1. Если же мы вместо 120 зададим 100, то программа не зациклится (в отличие от случая использования rejection-query или mh-query, что можно считать их недостатком), а просто вернет пустое множество. Можно поставить в качестве условия и не строгое равенство, а какое-то другое ограничение.

Таким же образом можно решать и более сложные задачи. Допустим, мы хотим решить задачу о сумме подмножеств: в ней надо из заданного множества чисел найти такое подмножество, сумма в котором равна заданному числу (обычно в качестве этого числа берется 0 и требуется, чтобы подмножество было не пустым; но чтобы избавиться от проверки на нетривиальность решения, мы возьмем ненулевую сумму). Казалось бы, причем тут вероятностное программирование? Но случайные величины – это просто неизвестные величины (для которых заданы априорные вероятности). В любых задачах нам нужно найти что-то неизвестное, в том числе и в задаче о сумме подмножеств.

Посмотрим на следующую элементарную программу (ее можно было бы даже еще упростить, записав summ через fold).

```
(define (solution xs v)
  (rejection-query
   (define ws (repeat (length xs) flip))
   (define (summ xs ws)
     (if (null? xs) 0
         (+ (if (car ws) (car xs) 0) (summ (cdr xs)
(cdr ws)))))
   ws
   (equal? (summ xs ws) v)))
(solution '(-1 3 7 5 -9 -1) 1)
```

Здесь ws – список случайных булевых значений. Процедура summ вычисляет сумму элементов списка xs, для которых соответствующие элементы списка ws истинны. Далее запрашивается значения ws, для которых выполняется условие равенства полученной суммы заданному числу v.

Запустив эту программу, можно получить такой результат: (#f #t #t #f #t #f), который, конечно, является правильным (так как 3+7-9=1).

Естественно, Чёрч не делает чуда и при повышении размерности этой задачи он с ней не справится. Однако не может не удивлять то, что столь разные задачи ИИ могут быть хотя бы поставлены (и отчасти решены) с использованием одного и того же языка. Ну, а проблема эффективного вывода как была, так и остается. В вероятностных языках она хотя бы выделяется в чистом виде.