Will Monroe
CS 109

Lecture Notes #16
July 31, 2017

Advanced Expectation

Based on a handout by Chris Piech

Conditional Expectation

Let X and Y be joint random variables. Recall that the conditional probability mass function (if they are discrete) and probability density function (if they are continuous) are respectively:

$$p_{X|Y}(x|y) = \frac{p_{X,Y}(x,y)}{p_Y(y)}$$
$$f_{X|Y}(x|y) = \frac{f_{X,Y}(x,y)}{f_Y(y)}$$

We define the conditional expectation of X given $Y = y$ to be:

$$E[X \mid Y = y] = \sum_x x p_{X|Y}(x|y)$$
$$E[X \mid Y = y] = \int_{-\infty}^{\infty} dx\, x f_{X|Y}(x|y)$$

Where the first equation applies if X and Y are discrete and the second applies if they are continuous.

Properties of Conditional Expectation

Here are some helpful, familiar properties of conditional expectation:

$$E[g(X) \mid Y = y] = \sum_x g(x) p_{X|Y}(x|y) \qquad \text{if X and Y are discrete}$$
$$E[g(X) \mid Y = y] = \int_{-\infty}^{\infty} dx\, g(x) f_{X|Y}(x|y) \qquad \text{if X and Y are continuous}$$
$$E[\sum_{i=1}^{n} X_i \mid Y = y] = \sum_{i=1}^{n} E[X_i \mid Y = y]$$
$$E[E[X \mid Y]] = E[X]$$

The last property may look a bit weird. Because Y is a random variable, $E[X \mid Y]$ is a random variable too (it depends on Y). The outer expectation is over values of Y.

Example 1

You roll two 6-sided dice D_1 and D_2. Let $X = D_1 + D_2$ and let $Y =$ the value of D_2. What is $E[X|Y = 6]$?

$$E[X|Y = 6] = \sum_x x P(X = x \mid Y = 6)$$
$$= \left(\frac{1}{6}\right)(7 + 8 + 9 + 10 + 11 + 12) = \frac{57}{6} = 9.5$$

Which makes intuitive sense since $6 + E[\text{value of } D_1] = 6 + 3.5 = 9.5$.

Example 2

Consider the following code with random numbers:

```
int recurse() {
   int x = randomInt(1, 3);  // Equally likely values
   if (x == 1) return 3;
   else if (x == 2) return (5 + recurse());
   else return (7 + recurse());
}
```

Let Y = value returned by `recurse`. What is $E[Y]$? In other words, what is the expected return value? (In fact, the function has been constructed such that the return value roughly corresponds to the running time as well.)

We can use the nested expectation property:

$$E[Y] = E[E[Y \mid X]] = E[Y|X = 1]P(X = 1) + E[Y|X = 2]P(X = 2) + E[Y|X = 3]P(X = 3)$$

We'll need to calculate each of the conditional expectations:

$$\begin{aligned} E[Y|X = 1] &= 3 \\ E[Y|X = 2] &= E[5 + Y] = 5 + E[Y] \\ E[Y|X = 3] &= E[7 + Y] = 7 + E[Y] \end{aligned}$$

Now we can plug those values into the equation. Note that the probability of X taking on 1, 2, or 3 is 1/3:

$$\begin{aligned} E[Y] &= E[Y|X = 1]P(X = 1) + E[Y|X = 2]P(X = 2) + E[Y|X = 3]P(X = 3) \\ &= 3(1/3) + (5 + E[Y])(1/3) + (7 + E[Y])(1/3) \\ &= 5 + (2/3)E[Y] \\ (1/3)E[Y] &= 5 \\ E[Y] &= 15 \end{aligned}$$

Fun with Approximation

The examples below use some advanced approximation methods to solve some cool algorithmic problems.

Example 3: Hiring Software Engineers

You are interviewing n software engineer candidates and will hire only 1 candidate. All orderings of candidates are equally likely. Right after each interview, you must decide to hire or not hire. You cannot go back on a decision. At any point in time you can know the relative ranking of the candidates you have already interviewed.

The strategy that we propose is that we interview the first k candidates and reject them all. Then you hire the next candidate that is better than all of the first k candidates. What is the probability that the best of all the n candidates is hired for a particular choice of k? Let's denote that result $P_k(\text{Best})$. Let X be the position in the ordering of the best candidate:

$$\begin{aligned} P_k(\text{Best}) &= \sum_{i=1}^{n} P_k(\text{Best}|X=i)P(X=i) \\ &= \frac{1}{n}\sum_{i=1}^{n} P_k(\text{Best}|X=i) && \text{since each position is equally likely} \end{aligned}$$

What is $P_k(\text{Best}|X=i)$? If $i \leq k$, then the probability is 0, because the best candidate will be rejected without consideration. Otherwise we will chose the best candidate, who is in position i, only if the best of the first $i-1$ candidates is among the first k interviewed. If the best among the first $i-1$ is not among the first k, that candidate will be chosen over the true best. Since all orderings are equally likely, the probability that the best among the $i-1$ candidates is in the first k is:

$$\frac{k}{i-1} \qquad \text{if } i > k$$

Now we can plug this back into our original equation:

$$\begin{aligned} P_k(\text{Best}) &= \frac{1}{n}\sum_{i=1}^{n} P_k(\text{Best}|X=i) \\ &= \frac{1}{n}\sum_{i=k+1}^{n} \frac{k}{i-1} && \text{since we know } P_k(\text{Best}|X=i) \\ &\approx \frac{1}{n}\int_{i=k+1}^{n} \frac{k}{i-1}\,di && \text{by Riemann sum approximation} \\ &= \frac{k}{n}\ln(i-1)\Big|_{k+1}^{n} = \frac{k}{n}\ln\frac{n-1}{k} \approx \frac{k}{n}\ln\frac{n}{k} \end{aligned}$$

If we think of $P_k(\text{Best}) = \frac{k}{n}\ln\frac{n}{k}$ as a function of k, we can find the value of k that optimizes it by taking its derivative and setting it equal to 0. It turns out that the optimal value of k is n/e, where $e \approx 2.718$ is Euler's number.

Quicksort Analysis

In CS 106B/X, they told you that the Quicksort sorting algorithm was $O(n^2)$ in the worst case, but on average it's $O(n \log n)$, and it's fast in practice. Where does that $O(n \log n)$ claim come from? At this point we don't have to take anyone's word for it: we can prove it ourselves.

As a reminder, the high-level algorithm for Quicksort has four steps:

1. Pick an element to serve as the "pivot".
2. "Partition" the array: move all elements less than the pivot before the pivot, and move all elements greater than or equal to the pivot after it.
3. Sort the left partition (with a recursive call).
4. Sort the right partition (with a recursive call).

In code, it looks like this:

```
void quicksort(int arr[], int n)
{
   if (n < 2) return;

   int boundary = partition(arr, n);

   // Sort subarray up to pivot
   quicksort(arr, boundary);

   // Sort subarray after pivot to end
   quicksort(arr + boundary + 1, n - boundary - 1);
}
```

So simple! Of course, that simplicity is obtained mostly by abstraction: most of the work is done by the second line, the function call to `partition`. Here's the code for `partition`:

```
int partition(int arr[], int n)
{
   int lh = 1, rh = n - 1;

   int pivot = arr[0];
   while (true) {
     while (lh < rh && arr[rh] >= pivot) rh--;
     while (lh < rh && arr[lh] < pivot) lh++;
     if (lh == rh) break;
     swap(arr[lh], arr[rh]);
   }
   if (arr[lh] >= pivot) return 0;
   swap(arr[0], arr[lh]);
   return lh;
}
```

There is lot of looping to keep track of here, but knowing the intuition helps: we choose the first element as the pivot, and we keep two pointers, the "left hand" and the "right hand", which start at the second element and the end (respectively) and move towards each other. As they move inwards, each "hand" stops when it reaches an element that is on the wrong side of the pivot. Once both hands have stopped, we swap the two elements they are pointing at to allow them to continue. When they meet, everything to the left of the hands (except the pivot) is less than the pivot, and everything to the right is greater than or equal. Finally, we swap the pivot into its place between the two partitions (where the hands are) and return.

Probability of the Worst Case

The fact that all of the swapping and comparing is done in `partition` tells us that we should focus on the inner loops of `partition` to determine the running time of the algorithm. In particular, we can measure how long the algorithm takes by counting the number of comparisons that are made to pivots.

An important fact to note here is that once two elements are separated by the partitioning step, they can never be compared again. That means that the worst case will be when we never separate elements in `partition`—every pivot chosen is either the smallest or largest element remaining. In this case, every element gets to be the pivot, and we compare it to every other element that hasn't been the pivot yet: a total of $\binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$ comparisons. What is the probability of this worst case?

This should remind you of the degenerate binary tree example. The size of the sample space is the number of orderings of picking pivots: $n!$. For the event space, at each step, we have 2 "worst" choices: the minimum and maximum of the remaining elements. The total number of steps is the number of times we still have at least two elements remaining, which is $n - 1$. So assuming every ordering of picking pivots is equally likely, the probability of the worst case of Quicksort is $\frac{2^{n-1}}{n!}$. Since $n!$ grows much faster than 2^{n-1}, this probability shrinks very rapidly!

Average Case Analysis

On the other hand, this doesn't guarantee us better than $O(n^2)$ average running time. For that, we'll need to compute the *expectation* of the number of comparisons. Whenever you see a problem that requires computing the expectation of a number of events that happen, you should think about rewriting it as a sum of indicator variables.

Call the total number of comparisons X. Let's define $Y_1, \ldots Y_n$ to be the elements of the array in sorted order, such that Y_1 is the smallest element in the array, Y_2 is the second-smallest, and so on. Let $I_{a,b}$ be an indicator variable for the event that Y_a and Y_b are ever compared. We don't want to double-count comparisons, so we will enforce that b is always greater than a:

$$E[X] = E\left[\sum_{a=1}^{n-1}\sum_{b=a+1}^{n} I_{ab}\right] = \sum_{a=1}^{n-1}\sum_{b=a+1}^{n} E[I_{ab}] = \sum_{a=1}^{n-1}\sum_{b=a+1}^{n} P(Y_a \text{ and } Y_b \text{ ever compared})$$

What is the probability that Y_a and Y_b are ever compared? There are three cases to consider:

- Either Y_a or Y_b is chosen as the pivot. Then the pivot is compared to everything, so Y_a and Y_b *will* be compared.
- A pivot is chosen that is between Y_a and Y_b. Then Y_a and Y_b are separated by the partition around that pivot, so they *will not* compared.
- A pivot is chosen that is less than Y_a or greater than Y_b. In this case we've only delayed the decision: Y_a and Y_b aren't compared now, but they are still in the same partition to be compared in a future recursive call.

So in the end, the question of whether Y_a and Y_b are ever compared reduces to the question of which happens first: Y_a or Y_b becoming the pivot, or one of the values between them? Again assuming all choices of pivot are equally likely, the probability of Y_a or Y_b being the pivot before any of the $b - a - 1$ elements between them is $\frac{2}{b-a+1}$.

What's left is a lot of math and a healthy dose of approximations. Let's finish this off:

$$E[X] = \sum_{a=1}^{n-1} \sum_{b=a+1}^{n} \frac{2}{b-a+1}$$

As in the hiring example, we can approximate the sum with a Riemann integral:

$$\begin{aligned}
\sum_{b=a+1}^{n} \frac{2}{b-a+1} &\approx \int_{b=a+1}^{n} db\, \frac{2}{b-a+1} \\
&= [2\ln(b-a+1)]_{b=a+1}^{n} \\
&= 2\ln(n-a+1) - 2\ln 2 \\
&\approx 2\ln(n-a+1) && \text{for large } n
\end{aligned}$$

Plugging this into the outer sum:

$$\begin{aligned}
\sum_{a=1}^{n-1} \sum_{b=a+1}^{n} \frac{2}{b-a+1} &\approx \sum_{a=1}^{n-1} 2\ln(n-a+1) \\
&\approx \int_{a=1}^{n-1} da\, 2\ln(n-a+1) && \text{yet another Riemann approximation} \\
&= -2\int_{y=n}^{2} dy\ \ln y && \text{letting } y = n-a+1 \\
&= -2[y\ln y - y]_{y=n}^{2} && \text{integration by parts}^1 \\
&= -2[(2\ln 2 - 2) - (n\ln n - n)] \\
&= O(n\ln n)
\end{aligned}$$

Victory! You can rest easy now, knowing that your Quicksort routine is as fast (on average) as you always knew it was.

[1] Using $u = \ln y$ and $dv = dy$. Of course, it may just be easier to memorize $\int dy\ \ln y = y\ln y - y + C$.