Python\_Prob\_Stat\_Machine\_Learning\_Unpingco\_\_2E\_C02

2.1 Introduction

This chapter takes a geometric view of probability theory and relates it to familiar concepts in linear algebra and geometry. This approach connects your natural geometric intuition to the key abstractions in probability that can help guide your reasoning. This is particularly important in probability because it is easy to be misled. We need a bit of rigor and some intuition to guide us.

In grade school, you were introduced to the natural numbers (i.e., 1,2,3,..) and you learned how to manipulate them by operations like addition, subtraction, and multiplication. Later, you were introduced to positive and negative numbers and were again taught how to manipulate them. Ultimately, you were introduced to the calculus of the real line, and learned how to differentiate, take limits, and so on. This progression provided more abstractions, but also widened the ﬁeld of problems you could successfully tackle. The same is true of probability. One way to think about probability is as a new number concept that allows you to tackle problems that have a special kind of uncertainty built into them. Thus, the key idea is that there is some number, say x, with a traveling companion, say, f (x), and this companion represents the uncertainties about the value of x as if looking at the number x through a frosted window. The degree of opacity of the window is represented by f (x). If we want to manipulate x, then we have to ﬁgure out what to do with f (x). For example if we want y = 2x, then we have to understand how f (x) generates f (y).

Where is the random part? To conceptualize this, we need still another analogy: think about a beehive with the swarm around it representing f (x), and the hive itself, which you can barely see through the swarm, as x. The random piece is you don’t know which bee in particular is going to sting you! Once this happens the uncertainty evaporates. Up until that happens, all we have is a concept of a swarm (i.e., density of bees) which represents a potentiality of which bee will ultimately sting. In summary, one way to think about probability is as a way of carrying through mathematical reasoning (e.g., adding, subtracting, taking limits) with a notion of potentiality that is so-transformed by these operations.

**2.1.1 Understanding Probability Density**

In order to understand the heart of modern probability, which is built on the Lebesgue theoryofintegration,weneedtoextendtheconceptofintegrationfrombasiccalculus. To begin, let us consider the following piecewise function

$$

f(x) = \left\{\begin{matrix}

1 & \text{ if } 0 < x \leq 1 \\

2 & \text{ if } 1 < x \leq 2 \\

0 & \text{ otherwise }\\

\end{matrix}\right.

$$

as shown in Fig. 2.1. In calculus, you learned Riemann integration, which you can apply here as

$$

\int\_0^2 f(x) dx = 1 + 2 = 3

$$

which has the usual interpretation as the area of the two rectangles that make up f (x). So far, so good.

With Lesbesgue integration, the idea is very similar except that we focus on the y-axis instead of moving along the x-axis. The question is given f (x) = 1, what

Fig. 2.1 Simple piecewise constant function

is the set of x values for which this is true? For our example, this is true whenever x ∈ (0, 1 ] . So now we have a correspondence between the values of the function (namely, 1 and 2) and the sets of x values for which this is true, namely, { (0, 1 ]} and { (1, 2 ]} , respectively. To compute the integral, we simply take the function values (i.e., 1,2) and some way of measuring the size of the corresponding interval (i.e., μ ) as in the following:

$$

\int\_0^2 f dμ = 1 μ ( { (0, 1 ]} ) + 2 μ ( { (1, 2 ]} )

$$

We have suppressed some of the notation above to emphasize generality. Note that we obtain the same value of the integral as in the Riemann case when μ ((0, 1 ] ) = μ ((1, 2 ] ) = 1. By introducing the μ function as a way of measuring the intervals above, we have introduced another degree of freedom in our integration. This accommodates many weird functions that are not tractable using the usual Riemann theory, but we refer you to a proper introduction to Lesbesgue integration for further study [1]. Nonetheless, the key step in the above discussion is the introduction of the μ function, which we will encounter again as the so-called probability density function.

**2.1.2 Random Variables**

Most introductions to probability jump straight into random variables and then explain how to compute complicated integrals. The problem with this approach is that it skips over some of the important subtleties that we will now consider. Unfortunately, the term random variable is not very descriptive. A better term is measurable function. To understand why this is a better term, we have to dive into the formal constructions of probability by way of a simple example.

Consider tossing a fair six-sided die. There are only six outcomes possible,

Ω = { 1, 2, 3, 4, 5, 6 }

As we know, if the die is fair, then the probability of each outcome is 1/6. To say this formally, the measure of each set (i.e., { 1 } , { 2 } , . . . , { 6 } ) is μ ( { 1 } ) = μ ( { 2 } ) . . . = μ ( { 6 } ) = 1/6. In this case, the μ function we discussed earlier is the usual probability mass function, denoted by P. The measurable function maps a set into a number on the real line. For example, { 1 } ↦ → 1 is one such function.

Now, here’s where things get interesting. Suppose you were asked to construct a fair coin from the fair die. In other words, we want to throw the die and then record the outcomes as if we had just tossed a fair coin. How could we do this? One way would be to deﬁne a measurable function that says if the die comes up 3 or less, then we declare heads and otherwise declare tails. This has some strong intuition behind it, but let’s articulate it in terms of formal theory. This strategy creates two different non-overlapping sets { 1, 2, 3 } and { 4, 5, 6 } . Each set has the same probability measure,

P( { 1, 2, 3 } ) = 1/2

P( { 4, 5, 6 } ) = 1/2

And the problem is solved. Everytime the die comes up { 1, 2, 3 } , we record heads and record tails otherwise.

Is this the only way to construct a fair coin experiment from a fair die? Alternatively, we can deﬁne the sets as { 1 } , { 2 } , { 3, 4, 5, 6 } . If we deﬁne the corresponding measure for each set as the following

P( { 1 } ) = 1/2

P( { 2 } ) = 1/2

P( { 3, 4, 5, 6 } ) = 0

then, we have another solution to the fair coin problem. To implement this, all we do is ignore every time the die shows 3,4,5,6 and throw again. This is wasteful, but it solves the problem. Nonetheless, we hope you can see how the interlocking pieces of the theory provide a framework for carrying the notion of uncertainty/potentiality from one problem to the next (e.g., from the fair die to the fair coin).

Let’s consider a slightly more interesting problem where we toss two dice. We assume that each throw is independent, meaning that the outcome of one does not inﬂuence the other. What are the sets in this case? They are all pairs of possible outcomes from two throws as shown below,

Ω = { (1, 1), (1, 2), . . . , (5, 6), (6, 6) }

What are the measures of each of these sets? By virtue of the independence claim, the measure of each is the product of the respective measures of each element. For instance,

$$

P((1, 2)) = P( { 1 } )P( { 2 } ) = \frac{1}{6^2}

$$

With all that established, we can ask the following question: what is the probability that the sum of the dice equals seven? As before, the ﬁrst thing to do is characterize the measurable function for this as X : (a, b) ↦ → (a + b). Next, we associate all of the (a, b) pairs with their sum. We can create a Python dictionary for this as shown,

**>>> d={(i,j):i+j for i in range(1,7) for j in range(1,7)}**

The next step is to collect all of the (a, b) pairs that sum to each of the possible values from two to twelve.

>>> from collections import defaultdict

>>> dinv = defaultdict(list)

>>> for i,j in d.items(): ... dinv[j].append(i) ...

**Programming Tip**

The defaultdict object from the built-in collections module creates dictionaries with default values when it encounters a new key. Otherwise, we would have had to create default values manually for a regular dictionary.

For example, dinv[7] contains the following list of pairs that sum to seven,

**>>> dinv[7]**

[(1, 6), (2, 5), (3, 4), (4, 3), (5, 2), (6, 1)]

The next step is to compute the probability measured for each of these items. Using the independence assumption, this means we have to compute the sum of the products of the individual item probabilities in dinv. Because we know that each outcome is equally likely, the probability of every term in the sum equals 1/36. Thus, all we have to do is count the number of items in the corresponding list for each key in dinv and divide by 36. For example, dinv[11] contains [(5, 6), (6, 5)]. The probability of 5+6=6+5=11 is the probability of this set which is composed of the sum of the probabilities of the individual elements (5,6),(6,5). In this case, we have P(11) = P( { (5, 6) } ) + P( { (6, 5) } ) = 1/36 + 1/36 = 2/36. Repeating this procedure for all the elements, we derive the probability mass function as shown below,

**>>> X={i:len(j)/36. for i,j in dinv.items()}**

**>>> print(X)**

{2: 0.027777777777777776,

3: 0.05555555555555555, 4: 0.08333333333333333, 5: 0.1111111111111111, 6: 0.1388888888888889, 7: 0.16666666666666666, 8: 0.1388888888888889, 9: 0.1111111111111111, 10: 0.08333333333333333, 11: 0.05555555555555555, 12: 0.027777777777777776} 44

**Programming Tip**

In the preceding code note that 36. is written with the trailing decimal mark. This is a good habit to get into because the default division operation changed between Python 2.x and Python 3.x. In Python 2.x division is integer division by default, and it is ﬂoating-point division in Python 3.x.

The above example exposes the elements of probability theory that are in play for this simple problem while deliberately suppressing some of the gory technical details. With this framework, we can ask other questions like what is the probability that half the product of three dice will exceed the their sum? We can solve this using the same method as in the following. First, let’s create the ﬁrst mapping,

**>>> d={(i,j,k):((i\*j\*k)/2>i+j+k) for i in range(1,7)**

**... ...**

**for j in range(1,7) for k in range(1,7)}**

The keys of this dictionary are the triples and the values are the logical values of whether or not half the product of three dice exceeds their sum. Now, we do the inverse mapping to collect the corresponding lists,

**>>> dinv = defaultdict(list)**

**>>> for i,j in d.items(): ... dinv[j].append(i) ...**

Note that dinv contains only two keys, True and False. Again, because the dice are independent, the probability of any triple is 1/6 3 . Finally, we collect this for each outcome as in the following,

**>>> X={i:len(j)/6.0\*\*3 for i,j in dinv.items()}**

**>>> print(X)**

{False: 0.37037037037037035, True: 0.6296296296296297}

Thus, the probability of half the product of three dice exceeding their sum is 136/6.0\*\*3) = 0.63. The set that is induced by the random variable has only two elements in it, True and False, with P(True) = 136/216 and P(False) = 1 − 136/216.

As a ﬁnal example to exercise another layer of generality, let is consider the ﬁrst problem with the two dice where we want the probability of a seven, but this time one of the dice is no longer fair. The distribution for the unfair die is the following:

P( { 1 } ) = P( { 2 } ) = P( { 3 } ) = 9

P( { 4 } ) = P( { 5 } ) = P( { 6 } ) = 9

From our earlier work, we know the elements corresponding to the sum of seven are the following: 2.1 Introduction

(1, 6), (2, 5), (3, 4), (4, 3), (5, 2), (6, 1) }

Because we still have the independence assumption, all we need to change is the probability computation of each of elements. For example, given that the ﬁrst die is the unfair one, we have

P((1, 6)) = P(1)P(6) = 9 6

and likewise for (2, 5) we have the following:

1 1 × P((2, 5)) = P(2)P(5) = 9 6

and so forth. Summing all of these gives the following:

1 1 1 1 1 1 2 1 2 1 2 1 1 × × × × × × P X (7) = = + + + + + 9 6 9 6 9 6 9 6 9 6 9 6 6

Let’s try computing this using Pandas instead of Python dictionaries. First, we construct a DataFrame object with an index of tuples consisting of all pairs of possible dice outcomes.

>>> from pandas import DataFrame

>>> d=DataFrame(index=[(i,j) for i in range(1,7) for j in range(1,7)], ... columns=[’sm’,’d1’,’d2’,’pd1’,’pd2’,’p’])

Now, we can populate the columns that we set up above where the outcome of the ﬁrst die is the d1 column and the outcome of the second die is d2,

**>>> d.d1=[i[0] for i in d.index]**

**>>> d.d2=[i[1] for i in d.index]**

Next, we compute the sum of the dices in the sm column,

**>>> d.sm=list(map(sum,d.index))**

With that established, the DataFrame now looks like the following:

**>>> d.head(5)** # show first five lines sm d1 d2 pd1 pd2 p (1, 1) 2 1 1 NaN NaN NaN (1, 2) 3 1 2 NaN NaN NaN (1, 3) 4 1 3 NaN NaN NaN (1, 4) 5 1 4 NaN NaN NaN (1, 5) 6 1 5 NaN NaN NaN

Next, we ﬁll out the probabilities for each face of the unfair die (d1) and the fair die (d2),

2

Probability

>>> d.loc[d.d1<=3,’pd1’]=1/9.

>>> d.loc[d.d1 > 3,’pd1’]=2/9.

>>> d.pd2=1/6.

>>> d.head(10) sm d1 d2 pd1 (1, 1) 2 1 1 0.111111 (1, 2) 3 1 2 0.111111 (1, 3) 4 1 3 0.111111 (1, 4) 5 1 4 0.111111 (1, 5) 6 1 5 0.111111 (1, 6) 7 1 6 0.111111 (2, 1) 3 2 1 0.111111 (2, 2) 4 2 2 0.111111 (2, 3) 5 2 3 0.111111 (2, 4) 6 2 4 0.111111

pd2 p

0.166667 NaN

0.166667 NaN

0.166667 NaN

0.166667 NaN

0.166667 NaN

0.166667 NaN

0.166667 NaN

0.166667 NaN

0.166667 NaN

0.166667 NaN

Finally, we can compute the joint probabilities for the sum of the shown faces as the following:

>>> d.p = d.pd1 \* d.pd2

>>> d.head(5) sm d1 d2 pd1 (1, 1) 2 1 1 0.111111 (1, 2) 3 1 2 0.111111 (1, 3) 4 1 3 0.111111 (1, 4) 5 1 4 0.111111 (1, 5) 6 1 5 0.111111

pd2 p

0.166667 0.0185185

0.166667 0.0185185

0.166667 0.0185185

0.166667 0.0185185

0.166667 0.0185185

With all that established, we can compute the density of all the dice outcomes by using groupby as in the following,

>>> d.groupby(’sm’)[’p’].sum() sm 2 0.018519 3 0.037037 4 0.055556 5 0.092593 6 0.129630 7 0.166667 8 0.148148 9 0.129630 10 0.111111 11 0.074074 12 0.037037 Name: p, dtype: float64

These examples have shown how the theory of probability breaks down sets and measurements of those sets and how these can be combined to develop the probability mass functions for new random variables. 2.1 Introduction

**2.1.3 Continuous Random Variables**

The same ideas work with continuous variables but managing the sets becomes trickierbecausetherealline,unlikediscretesets,hasmanylimitingpropertiesalready built into it that have to be handled carefully. Nonetheless, let’s start with an example that should illustrate the analogous ideas. Suppose a random variable X is uniformly distributed on the unit interval. What is the probability that the variable takes on values less than 1/2?

In order to build intuition onto the discrete case, let’s go back to our dice-throwing experiment with the fair dice. The sum of the values of the dice is a measurable function,

Y : { 1, 2, . . . , 6 } 2 → ↦ { 2, 3, . . . , 12 }

That is, Y is a mapping of the cartesian product of sets to a discrete set of outcomes. In order to compute probabilities of the set of outcomes, we need to derive the probability measure for Y, P Y , from the corresponding probability measures for each die. Our previous discussion went through the mechanics of that. This means that

P Y : { 2, 3, . . . , 12 } ↦ → [ 0, 1 ]

Note there is a separation between the function deﬁnition and where the target items of the function are measured in probability. More bluntly,

Y : A ↦ → B

with,

P Y : B ↦ → [ 0, 1 ]

Thus, to compute P Y , which is derived from other random variables, we have to express the equivalence classes in B in terms of their progenitor A sets.

The situation for continuous variables follows the same pattern, but with many more deep technicalities that we are going to skip. For the continuous case, the random variable is now,

X : R ↦ → R

with corresponding probability measure,

P X : R ↦ → [ 0, 1 ]

But where are the corresponding sets here? Technically, these are the Borel sets, but we can just think of them as intervals. Returning to our question, what is the probability that a uniformly distributed random variable on the unit interval takes values less than 1/2? Rephrasing this question according to the framework, we have the following:

X : [ 0, 1 ] ↦ → [ 0, 1 ]

with corresponding,

P X : [ 0, 1 ] ↦ → [ 0, 1 ]

To answer the question, by the deﬁnition of the uniform random variable on the unit interval, we compute the following integral,

1/2 P X ( [ 0, 1/2 ] ) = P X (0 < X < 1/2) = dx = 1/2 ∫ 0

where the above integral’s dx sweeps through intervals of the B-type. The measure of any dx interval (i.e., A-type set) is equal to dx, by deﬁnition of the uniform random variable. To get all the moving parts into one notationally rich integral, we can also write this as,

1/2 P X (0 < X < 1/2) = dP X (dx) = 1/2 ∫ 0

Now, let’s consider a slightly more complicated and interesting example. As before, suppose we have a uniform random variable, X and let us introduce another random variable deﬁned,

Y = 2X

Now, what is the probability that 0 < Y < 2 1 ? To express this in our framework, we write,

Y : [ 0, 1 ] ↦ → [ 0, 2 ]

with corresponding,

P Y : [ 0, 2 ] ↦ → [ 0, 1 ]

To answer the question, we need to measure the set [0,1/2], with the probability measure for Y, P Y ( [ 0, 1/2 ] ). How can we do this? Because Y is derived from the X random variable, as with the fair-dice throwing experiment, we have to create a set of equivalences in the target space (i.e., B-type sets) that reﬂect back on the input space (i.e., A-type sets). That is, what is the interval [0,1/2] equivalent to in terms of the X random variable? Because, functionally, Y = 2X, then the B-type interval [0,1/2] corresponds to the A-type interval [0,1/4]. From the probability measure of X, we compute this with the integral,

1/4 P Y ( [ 0, 1/2 ] ) = P X ( [ 0, 1/4 ] ) = dx = 1/4 ∫ 0

Now, let’s up the ante and consider the following random variable,

Y =X2 2.1 Introduction

49

where now X is still uniformly distributed, but now over the interval [ −1/2, 1/2 ] . We can express this in our framework as,

Y : [ −1/2, 1/2 ] ↦ → [ 0, 1/4 ]

with corresponding,

P Y : [ 0, 1/4 ] ↦ → [ 0, 1 ]

What is the P Y (Y < 1/8)? In other words, what is the measure of the set B Y = [ 0, 1/8 ] ? As before, because X is derived from our uniformly distributed random variable, we have to reﬂect the B Y set onto sets of the A-type. The thing to recognize is that because X 2 is symmetric about zero, all B Y sets reﬂect back into two sets. This means that for any set B Y , we have the correspondence B Y = A X + ∪ A X − . So, we have,

1 1 1 B Y = − <X <0 0 < Y < = 0 < X < { 8 } { √ 8 } ∪ { √ 8 }

From this perspective, we have the following solution,

P Y (B Y ) = P(A X + )/2 + P(A X − )/2

where the 2 1 comes from normalizing the P Y to one. Also,

1 A X + = 0 < X < { √ 8 }

1 AX − = <X <0 { √ 8 }

Therefore,

1 1 P Y (B Y ) = + 2 √ 8 2 √ 8

because P(A X + ) = P(A X − ) = 1/ √ 8. Let’s see if this comes out using the usual transformation of variables method from calculus. Using this method, the density f Y (y) = f X ( √ y)/(2 √ y) = 2 1 y . Then, we obtain, √

1 8 1 1 dy = ∫ 0 2 √ y √ 8

which is what we got using the sets method. Note that you would favor the calculus method in practice, but it is important to understand the deeper mechanics, because sometimes the usual calculus method fails, as the next problem shows.

**2.1.4 Transformation of Variables Beyond Calculus**

Suppose X and Y are uniformly distributed in the unit interval and we deﬁne Z as

X Z= Y−X

What is the f Z (z)? If you try this using the usual calculus method, you will fail (try it!). The problem is one of the technical prerequisites for the calculus method is not in force.

The key observation is that Z ∈ / (−1, 0 ] . If this were possible, the X and Y would have different signs, which cannot happen, given that X and Y are uniformly distributed over (0, 1 ] . Now, let’s consider when Z > 0. In this case, Y > X because Z cannot be positive otherwise. For the density function, we are interested in the set { 0 < Z < z } . We want to compute

P(Z < z) = B 1 d XdY ∫ ∫

with,

B 1 = { 0 < Z < z }

Now, we have to translate that interval into an interval relevant to X and Y. For 0 < Z, we have Y > X. For Z < z, we have Y > X(1/z + 1). Putting this together gives

A 1 = { max(X, X(1/z + 1)) < Y < 1 }

Integrating this over Y as follows,

1 X z X Xz { max(X, X(1/z + 1)) < Y < 1 } dY = where z > ∫ 0 z 1 − X

and integrating this one more time over X gives

z 1+z −X + z − Xz z dX = where z > 0 z 2(z + 1) ∫ 0

Note that this is the computation for the probability itself, not the probability density function. To get that, all we have to do is differentiate the last expression to obtain

1 f Z (z) = where z > 0 (z + 1)2

Now we need to compute this density using the same process for when z < −1. We want the interval Z < z for when z < −1. For a ﬁxed z, this is equivalent to X(1 + 1/z) < Y. Because z is negative, this also means that Y < X. Under these terms, we have the following integral,

1 X { X(1/z + 1) < Y < X } dY = − where z < −1 ∫ 0 z

and integrating this one more time over X gives the following

1 where z < −1 2z

To get the density for z < −1, we differentiate this with respect to z to obtain the following,

1 f Z (z) = where z < −1 2z2

Putting this all together, we obtain,

⎪ ⎧ (z + 1 1) 2 if z > 0 f Z (z) = 1 2 if z < −1 ⎪ ⎨ 2z ⎩ 0 otherwise

We will leave it as an exercise to show that this integrates out to one.

**2.1.5 Independent Random Variables**

Independence is a standard assumption. Mathematically, the necessary and sufﬁcient condition for independence between two random variables X and Y is the following:

P(X, Y) = P(X)P(Y)

Two random variables X and Y are uncorrelated if,

E(X − X)E(Y − Y) = 0

where X = E(X) Note that uncorrelated random variables are sometimes called orthogonal random variables. Uncorrelatedness is a weaker property than independence, however. For example, consider the discrete random variables X and Y uniformly distributed over the set { 1, 2, 3 } where

⎧ ⎪ 1 if ω = 1 X = 0 if ω = 2 ⎪ ⎨ ⎩ −1 if ω = 3 52

and also,

⎪ ⎧ 0 if ω = 1 Y = 1 if ω = 2 ⎪ ⎨ ⎩ 0 if ω = 3

Thus, E(X) = 0 and E(XY) = 0, so X and Y are uncorrelated. However, we have

1 P(X = 1, Y = 1) = 0 = ̸ P(X = 1)P(Y = 1) = 9

So, these two random variables are not independent. Thus, uncorrelatedness does not imply independence, generally, but there is the important case of Gaussian random variables for which it does. To see this, consider the probability density function for two zero-mean, unit-variance Gaussian random variables X and Y,

x 2 −2 ρ xy+y 2

2 ( ρ 2 −1 ) e f X,Y (x, y) = 2π √ 1 − ρ2

where ρ := E(XY) is the correlation coefﬁcient. In the uncorrelated case where ρ = 0, the probability density function factors into the following,

e− 1 2 ( x 2 + y 2 ) e− x 2 2 e− y 2 2 f X,Y (x, y) = = = f X (x) f Y (y) 2π √ 2π √ 2π

which means that X and Y are independent.

Independence and conditional independence are closely related, as in the following:

P(X, Y | Z) = P(X | Z)P(Y | Z)

which says that X and Y and independent conditioned on Z. Conditioning independent random variables can break their independence. For example, consider two independent Bernoulli-distributed random variables, X 1 , X 2 ∈ { 0, 1 } . We deﬁne Z = X 1 + X 2 . Note that Z ∈ { 0, 1, 2 } . In the case where Z = 1, we have,

P(X 1 | Z = 1) > 0

P(X 2 | Z = 1) > 0

Even though X 1 , X 2 are independent, after conditioning on Z, we have the following,

P(X 1 = 1, X 2 = 1 | Z = 1) = 0 = ̸ P(X 1 = 1 | Z = 1)P(X 2 = 1 | Z = 1)

Thus, conditioning on Z breaks the independence of X 1 , X 2 . This also works in the opposite direction—conditioning can make dependent random variables independent. Deﬁne Z n = ∑ i n X i with X i independent, integer-valued random variables. The Z n variables are dependent because they stack the same telescoping set of Xi variables. Consider the following,

P(Z1 = i, Z 2 = k, Z 3 = j) P(Z 1 = i, Z 3 = j | Z 2 = k) = P(Z 2 = k)

(2.1.5.1)

P(X1 = i)P(X 2 = k − i)P(X 3 = j − k) = (2.1.5.2) P(Z 2 = k)

where the factorization comes from the independence of the X i variables. Using the deﬁnition of conditional probability,

P(Z1 = i, Z 2 = k) P(Z 1 = i | Z 2 ) = P(Z 2 = k)

We can continue to expand Eq. 2.1.5.1,

P(X3 = j − k)P(Z 2 = k) P(Z 1 = i, Z 3 = j | Z 2 = k) = P(Z 1 = i | Z 2 ) P(Z 2 = k)

= P(Z 1 = i | Z 2 )P(Z 3 = j | Z 2 )

where P(X 3 = j − k)P(Z 2 = k) = P(Z 3 = j, Z 2 ). Thus, we see that dependence between random variables can be broken by conditioning to create conditionally independent random variables. As we have just witnessed, understanding how conditioning inﬂuences independence is important and is the main topic of study in Probabilistic Graphical Models, a ﬁeld with many algorithms and concepts to extract these notions of conditional independence from graph-based representations of random variables.

**2.1.6 Classic Broken Rod Example**

Let’s do one last example to exercise ﬂuency in our methods by considering the following classic problem: given a rod of unit-length, broken independently and randomly at two places, what is the probability that you can assemble the three remaining pieces into a triangle? The ﬁrst task is to ﬁnd a representation of a triangle as an easy-to-apply constraint. What we want is something like the following:

P( triangle exists ) =

1 ∫ 0 ∫ 0

1

{

triangle exists } d XdY

where X andY areindependentanduniformlydistributedintheunit-interval.Heron’s formula for the area of the triangle,

area = √ (s − a)(s − b)(s − c)s

where s = (a + b + c)/2 is what we need. The idea is that this yields a valid area only when each of the terms under the square root is greater than or equal to zero. Thus, suppose that we have

a=X

b=Y−X

c = 1−Y

assuming that Y > X. Thus, the criterion for a valid triangle boils down to

{ (s > a) ∧ (s > b) ∧ (s > c) ∧ (X < Y) }

After a bit of manipulation, this consolidates into:

1 1 1 < Y < 1 (2Y 1) < X < { 2 ∧ 2 2 }

which we integrate out by d X ﬁrst to obtain

1 1 1 1 1 P( triangle exists ) = ∧ < Y < 1 (2Y 1) < X < d XdY ∫ 0 ∫ 0 { 2 2 2 }

1 P( triangle exists ) = (1 − Y)dY ∫ 1

2

and then by dY to obtain ﬁnally,

1 P( triangle exists ) = 8

when Y > X. By symmetry, we get the same result for X > Y. Thus, the ﬁnal result is the following:

1 1 1 P( triangle exists ) = = + 8 8 4

We can quickly check using this result using Python for the case Y > X using the following code:

>>> import numpy as np

>>> x,y = np.random.rand(2,1000) # uniform rv

>>> a,b,c = x,(y-x),1-y # 3 sides

>>> s = (a+b+c)/2

>>> np.mean((s>a) & (s>b) & (s>c) & (y>x)) # approx 1/8=0.125

0.137 2.1 Introduction

55

**Programming Tip**

The chained logical & symbols above tell Numpy that the logical operation should be considered element-wise.

**2.2 Projection Methods**

The concept of projection is key to developing an intuition about conditional probability. We already have a natural intuition of projection from looking at the shadows of objects on a sunny day. As we will see, this simple idea consolidates many abstract ideas in optimization and mathematics. Consider Fig. 2.2 where we want to ﬁnd a point along the blue line (namely, x) that is closest to the black square (namely, y). In other words, we want to inﬂate the gray circle until it just touches the black line. Recall that the circle boundary is the set of points for which

(y − x) T (y − x) = y − x = √ ∥ ∥

for some value of . So we want a point x along the line that satisﬁes this for the smallest . Then, that point will be the closest point on the black line to the black square. It may be obvious from the diagram, but the closest point on the line occurs where the line segment from the black square to the black line is perpendicular to the line. At this point, the gray circle just touches the black line. This is illustrated below in Fig. 2.3.

Fig. 2.2 Given the point y (black square) we want to ﬁnd the x along the line that is closest to it. The gray circle is the locus of points within a ﬁxed distance from y 56

2

Probability

Fig. 2.3 The closest point on the line occurs when the line is tangent to the circle. When this happens, the black line and the line (minimum distance) are perpendicular

**Programming Tip**

Figure 2.2 uses the matplotlib.patches module. This module contains primitive shapes like circles, ellipses, and rectangles that can be assembled into complex graphics. After importing a particular shape, you can apply that shape to an existing axis using the add\_patch method. The patches themselves can by styled using the usual formatting keywords like color and alpha.

Now that we can see what’s going on, we can construct the the solution analytically. We can represent an arbitrary point along the black line as:

x = αv

where α ∈ R slides the point up and down the line with

v = [1, 1]T

Formally, v is the subspace onto which we want to project y. At the closest point, the vector between y and x (the error vector above) is perpendicular to the line. This means that

(y − x) T v = 0

and by substituting and working out the terms, we obtain

yT v α = ∥ v ∥2

The error is the distance between αv and y. This is a right triangle, and we can use the Pythagorean theorem to compute the squared length of this error as

v ∥2 ∥ yT 2 = ∥ (y − x) ∥ 2 = ∥ y ∥ 2 − α 2 ∥ v ∥ 2 = ∥ y ∥ 2 ∥ v ∥2

where ∥ v ∥ 2 = v T v. Note that since 2

≥

0, this also shows that

∥ y T v ∥ ≤ ∥ y ∥∥ v ∥

which is the famous and useful Cauchy–Schwarz inequality which we will exploit later. Finally, we can assemble all of this into the projection operator

1 P v = vvT ∥ v ∥2

With this operator, we can take any y and ﬁnd the closest point on v by doing

vT y Pv y=v ( ∥ v ∥ 2 )

where we recognize the term in parenthesis as the α we computed earlier. It’s called an operator because it takes a vector (y) and produces another vector (αv). Thus, projection uniﬁes geometry and optimization.

**2.2.1 Weighted Distance**

We can easily extend this projection operator to cases where the measure of distance between y and the subspace v is weighted. We can accommodate these weighted distances by re-writing the projection operator as

vT QT Pv =v v T Qv

(2.2.1.1)

where Q is positive deﬁnite matrix. In the previous case, we started with a point y and inﬂated a circle centered at y until it just touched the line deﬁned by v and this point was closest point on the line to y. The same thing happens in the general case with a weighted distance except now we inﬂate an ellipse, not a circle, until the ellipse touches the line.

Note that the error vector (y − αv) in Fig. 2.4 is still perpendicular to the line (subspace v), but in the space of the weighted distance. The difference between the ﬁrst projection (with the uniform circular distance) and the general case (with the elliptical weighted distance) is the inner product between the two cases. For example,

Fig. 2.4 In the weighted case, the closest point on the line is tangent to the ellipse and is still perpendicular in the sense of the weighted distance

in the ﬁrst case we have y T v and in the weighted case we have y T Q T v. To move from the uniform circular case to the weighted ellipsoidal case, all we had to do was change all of the vector inner products. Before we ﬁnish, we need a formal property of projections:

Pv Pv =Pv

known as the idempotent property which basically says that once we have projected onto a subspace, subsequent projections leave us in the same subspace. You can verify this by computing Eq. 2.2.1.1.

Thus, projection ties a minimization problem (closest point to a line) to an algebraic concept (inner product). It turns out that these same geometric ideas from linear algebra [2] can be translated to the conditional expectation. How this works is the subject of our next section.

**2.3 Conditional Expectation as Projection**

Now that we understand projection methods geometrically, we can apply them to conditional probability. This is the key concept that ties probability to geometry, optimization, and linear algebra.

Inner Product for Random Variables. From our previous work on projection for vectors in R n , we have a good geometric grasp on how projection is related to Minimum Mean Squared Error (MMSE). By one abstract step, we can carry all of our geometric interpretations to the space of random variables. For example, we previously noted that at the point of projection, we had the following orthogonal (i.e., perpendicular vectors) condition,

(y − v opt ) T v = 0

which by noting the inner product slightly more abstractly as 〈 x, y 〉 = x T y, we can express as

〈 y − v opt , v 〉 = 0

and by deﬁning the inner product for the random variables X and Y as

〈 X, Y 〉 = E(XY)

we have the same relationship:

〈 X − h opt (Y), Y 〉 = 0

which holds not for vectors in R n , but for random variables X and Y and functions of those random variables. Exactly why this is true is technical, but it turns out that one can build up the entire theory of probability this way [3], by using the expectation as an inner product.

Furthermore, by abstracting out the inner product concept, we have connected minimum-mean-squared-error (MMSE) optimization problems, geometry, and random variables. That’s a lot of mileage to get a out of an abstraction and it enables us to shift between these interpretations to address real problems. Soon, we’ll do this with some examples, but ﬁrst we collect the most important result that ﬂows naturally from this abstraction.

Conditional Expectation as Projection. The conditional expectation is the minimum mean squared error (MMSE) solution to the following problem 1 :

min (x − h(y)) 2 f X,Y (x, y)dxdy h ∫ R 2

with the minimizing hopt

(Y) as

h opt (Y) = E(X | Y)

which is another way of saying that among all possible functions h(Y), the one that minimizes the MSE is E(X | Y). From our previous discussion on projection, we noted that these MMSE solutions can be thought of as projections onto a subspace that characterizes Y. For example, we previously noted that at the point of projection, we have perpendicular terms,

〈 X − h opt (Y), Y 〉 = 0

(2.3.0.1)

1 See appendix for proof using the Cauchy–Schwarz inequality. 60

but since we know that the MMSE solution

h opt (Y) = E(X | Y)

we have by direct substitution,

E(X − E(X | Y), Y) = 0

(2.3.0.2)

That last step seems pretty innocuous, but it ties MMSE to conditional expectation to the inner project abstraction, and in so doing, reveals the conditional expectation to be a projection operator for random variables. Before we develop this further, let’s grab some quick dividends. From the previous equation, by linearity of the expectation, we obtain,

E(XY) = E(YE(X | Y))

which is the so-called tower property of the expectation. Note that we could have found this by using the formal deﬁnition of conditional expectation,

fX,Y (x, y) E(X | Y) = x dxdy 2 f Y (y) ∫ R

and brute-force direct integration,

fX,Y (x, y) E(YE(X | Y)) = y x f Y (y)dxdy ∫ R ∫ R f Y (y)

= xy f X,Y (x, y)dxdy ∫ R 2

= E(XY)

which is not very geometrically intuitive. This lack of geometric intuition makes it hard to apply these concepts and keep track of these relationships.

We can keep pursuing this analogy and obtain the length of the error term from the orthogonality property of the MMSE solution as,

〈 X − h opt (Y), X − h opt (Y) 〉 = 〈 X, X 〉 − 〈 h opt (Y), h opt (Y) 〉

and then by substituting all the notation we obtain

E(X − E(X | Y)) 2 = E(X) 2 − E(E(X | Y))2

which would be tough to compute by direct integration.

To formally establish that E(X | Y) is in fact a projection operator we need to show idempotency. Recall that idempotency means that once we project something onto a subspace, further projections do nothing. In the space of random variables, E(X | ·) is the idempotent projection as we can show by noting that

h opt = E(X | Y)

is purely a function of Y, so that

E(h opt (Y) | Y) = h opt (Y)

because Y is ﬁxed, this veriﬁes idempotency. Thus, conditional expectation is the corresponding projection operator for random variables. We can continue to carry over our geometric interpretations of projections for vectors (v) into random variables (X). With this important result, let’s consider some examples of conditional expectations obtained by using brute force to ﬁnd the optimal MMSE function hopt as well as by using our new perspective on conditional expectation.

Example. Suppose we have a random variable, X, then what constant is closest to X in the sense of the mean-squared-error (MSE)? In other words, which c ∈ R minimizes the following mean squared error:

MSE = E(X − c)2

we can work this out many ways. First, using calculus-based optimization,

E(X − c) 2 = E(c 2 − 2cX + X 2 ) = c 2 − 2cE(X) + E(X 2 )

and then take the ﬁrst derivative with respect to c and solve:

c opt = E(X)

Remember that X may potentially take on many values, but this says that the closest number to X in the MSE sense is E(X). This is intuitively pleasing. Coming at this same problem using our inner product, from Eq. 2.3.0.2 we know that at the point of projection

E((X − c opt )1) = 0

where the 1 represents the space of constants we are projecting onto. By linearity of the expectation, gives

c opt = E(X)

Using the projection approach, because E(X | Y) is the projection operator, with Y = Ω (the entire underlying probability space), we have, using the deﬁnition of conditional expectation:

E(X | Y = Ω) = E(X) 62

This is because of the subtle fact that a random variable over the entire Ω space can only be a constant. Thus, we just worked the same problem three ways (optimization, orthogonal inner products, projection).

Example. Let’s consider the following example with probability density f X,Y = x + y where (x, y) ∈ [ 0, 1 ] 2 and compute the conditional expectation straight from the deﬁnition:

1 (x, y) 1 3y 2 fX,Y x y + + E(X | Y) = x dx = x dx = ∫ 0 f Y (y) ∫ 0 y + 1/2 6y + 3

That was pretty easy because the density function was so simple. Now, let’s do it the hard way by going directly for the MMSE solution h(Y). Then,

1 1 MSE = min (x − h(y)) 2 f X,Y (x, y)dxdy h ∫ 0 ∫ 0

1 1 1 1 2 = min yh 2 (y) − yh(y) + (y) y h2 h(y) dy + + h ∫ 0 3 2 3 4

Now we have to ﬁnd a function h that is going to minimize this. Solving for a function, as opposed to solving for a number, is generally very, very hard, but because we are integrating over a ﬁnite interval, we can use the Euler–Lagrange method from variational calculus to take the derivative of the integrand with respect to the function h(y)andsetittozero.UsingEuler–Lagrangemethods,weobtainthefollowingresult,

2 = 0 2yh(y) − y + h(y) 3

Solving this gives

3y + 2 h opt (y) = 6y + 3

which is what we obtained before. Finally, we can solve this using our inner product in Eq. 2.3.0.1 as

E((X − h(Y))Y) = 0

Writing this out gives,

1

∫ 0 ∫ 0

1

∫ 0

1

(x − h(y))y(x + y)dxdy =

1 y(−3(2y 1)h(y) 3y 2)dy = 0 + + + 6

and the integrand must be zero,

2y + 3y 2 − 3yh(y) − 6y 2 h(y) = 0 2.3 Conditional Expectation as Projection

63

and solving this for h(y) gives the same solution:

3y + 2 h opt (y) = 6y + 3

Thus, doing it by the brute force integration from the deﬁnition, optimization, or inner product gives us the same answer; but, in general, no method is necessarily easiest because they both involve potentially difﬁcult or impossible integration, optimization, or functional equation solving. The point is that now that we have a deep toolbox, we can pick and choose which tools we want to apply for different problems.

Before we leave this example, let’s use Sympy to verify the length of the error function we found earlier for this example:

E(X − E(X | Y)) 2 = E(X) 2 − E(E(X | Y))2

that is based on the Pythagorean theorem. First, we need to compute the marginal densities,

>>> from sympy.abc import y,x

>>> from sympy import integrate, simplify # joint density

>>> fxy = x + y

>>> fy = integrate(fxy,(x,0,1)) # marginal density

>>> fx = integrate(fxy,(y,0,1)) # marginal density

Then, we need to write out the conditional expectation,

>>> EXY = (3\*y+2)/(6\*y+3) # conditional expectation

Next, we can compute the left side, E(X − E(X | Y)) 2 , as the following,

>>> # from the definition

>>> LHS=integrate((x-EXY)\*\*2\*fxy,(x,0,1),(y,0,1))

>>> LHS # left-hand-side

-log(3)/144 + 1/12

We can similarly compute the right side, E(X) 2 − E(E(X | Y)) 2 , as the following,

>>> # using Pythagorean theorem

>>> RHS=integrate((x)\*\*2\*fx,(x,0,1))-integrate((EXY)\*\*2\*fy,(y,0,1))

>>> RHS # right-hand-side

-log(3)/144 + 1/12

Finally, we can verify that the left and right sides match,

>>> print(simplify(LHS-RHS)==0) True

In this section, we have pulled together all the projection and least-squares optimization ideas from the previous sections to connect geometric notions of projection from vectors in R n to random variables. This resulted in the remarkable realization that the conditional expectation is in fact a projection operator for random variables.

Knowing this allows to approach difﬁcult problems in multiple ways, depending on which way is more intuitive or tractable in a particular situation. Indeed, ﬁnding the right problem to solve is the hardest part, so having many ways of looking at the same concepts is crucial.

For much more detailed development, the book by Mikosch [4] has some excellent sections covering much of this material with a similar geometric interpretation. Kobayashi et al. [5] does too. Nelson [3] also has a similar presentation based on hyper-real numbers.

**2.3.1 Appendix**

We want to prove that we the conditional expectation is the minimum mean squared error minimizer of the following:

J = min | X − h(Y) | 2 f X,Y (x, y)dxdy h ∫ R 2

We can expand this as follows,

J = min | X | 2 f X,Y (x, y)dxdy + | h(Y) | 2 f X,Y (x, y)dxdy h ∫ R 2 ∫ R 2

−

∫ R 2

2Xh(Y) f X,Y (x, y)dxdy

To minimize this, we have to maximize the following:

A = max Xh(Y) f X,Y (x, y)dxdy h ∫ R 2

Breaking up the integral using the deﬁnition of conditional expectation

A = max X f X | Y (x | y)dx h(Y) f Y (y)dy h ∫ R ( ∫ R )

(2.3.1.1)

= max E(X | Y)h(Y) f Y (Y)dy h ∫ R

(2.3.1.2)

From properties of the Cauchy–Schwarz inequality, we know that the maximum happens when h opt (Y) = E(X | Y), so we have found the optimal h(Y) function as:

h opt (Y) = E(X | Y)

which shows that the optimal function is the conditional expectation. 2.4 Conditional Expectation and Mean Squared Error

65

**2.4 Conditional Expectation and Mean Squared Error**

In this section, we work through a detailed example using conditional expectation and optimization methods. Suppose we have two fair six-sided dice (X and Y) and we want to measure the sum of the two variables as Z = X + Y. Further, let’s suppose that given Z, we want the best estimate of X in the mean-squared-sense. Thus, we want to minimize the following:

J(α) = ∑ (x − αz) 2 P(x, z)

where P is the probability mass function for this problem. The idea is that when we have solved this problem, we will have a function of Z that is going to be the minimum MSE estimate of X. We can substitute in for Z in J and get:

J(α) = ∑ (x − α(x + y)) 2 P(x, y)

Let’s work out the steps in Sympy in the following:

>>> import sympy as S

>>> from sympy.stats import density, E, Die

>>> x=Die(’D1’,6) # 1st six sided die

>>> y=Die(’D2’,6) # 2nd six sides die

>>> a=S.symbols(’a’) # sum of 1st and 2nd die

>>> z = x+y

>>> J = E((x-a\*(x+y))\*\*2) # expectation

>>> print(S.simplify(J)) 329\*a\*\*2/6 - 329\*a/6 + 91/6

With all that setup we can now use basic calculus to minimize the objective function J,

>>> sol,=S.solve(S.diff(J,a),a) # using calculus to minimize

>>> print(sol) # solution is 1/2 1/2

**Programming Tip**

Sympy has a stats module that can do some basic work with expressions involving probability densities and expectations. The above code uses its E function to compute the expectation.

This says that z/2 is the MSE estimate of X given Z which means geometrically (interpreting the MSE as a squared distance weighted by the probability mass function) that z/2 is as close to x as we are going to get for a given z.

Fig. 2.5 The values of Z are in yellow with the corresponding values for X and Y on the axes. The gray scale colors indicate the underlying joint probability density

Let’s look at the same problem using the conditional expectation operator E(· | z) and apply it to our deﬁnition of Z. Then

E(z | z) = E(x + y | z) = E(x | z) + E(y | z) = z

using the linearity of the expectation. Now, since by the symmetry of the problem (i.e., two identical die), we have

E(x | z) = E(y | z)

we can plug this in and solve

2E(x | z) = z

which once again gives,

z E(x | z) = 2

which is equal to the estimate we just found by minimizing the MSE. Let’s explore this further with Fig. 2.5. Figure 2.5 shows the values of Z in yellow with the corresponding values for X and Y on the axes. Suppose z = 2, then the closest X to this is X = 1, which is what E(x | z) = z/2 = 1 gives. What happens when Z = 7? In this case, this value is spread out diagonally along the X axis so if X = 1, then Z is 6 units away, if X = 2, then Z is 5 units away and so on.

Now, back to the original question, if we had Z = 7 and we wanted to get as close as we could to this using X, then why not choose X = 6 which is only one unit away from Z? The problem with doing that is X = 6 only occurs 1/6 of the time, so we are not likely to get it right the other 5/6 of the time. So, 1/6 of the time we are one unit away but 5/6 of the time we are much more than one unit away. This means that the MSE score is going to be worse. Since each value of X from 1 to 6 is equally likely, to play it safe, we choose 7/2 as the estimate, which is what the conditional expectation suggests.

We can check this claim with samples using Sympy below:

>>> import numpy as np

>>> from sympy import stats

>>> # Eq constrains Z

>>> samples\_z7 = lambda : stats.sample(x, S.Eq(z,7))

>>> #using 6 as an estimate

>>> mn= np.mean([(6-samples\_z7())\*\*2 for i in range(100)])

>>> #7/2 is the MSE estimate

>>> mn0= np.mean([(7/2.-samples\_z7())\*\*2 for i in range(100)])

>>> print(’MSE=%3.2f using 6 vs MSE=%3.2f using 7/2 ’ % (mn,mn0)) MSE=9.20 using 6 vs MSE=2.99 using 7/2

**Programming Tip**

The stats.sample(x, S.Eq(z,7))functioncallsamplesthe xvariable subject to a condition on the z variable. In other words, it generates random samples of x die, given that the sum of the outcomes of that die and the y die

add up to z==7.

Please run the above code repeatedly until you are convinced that the E(x | z) gives the lower MSE every time. To push this reasoning, let’s consider the case where the die is so biased so that the outcome of 6 is ten times more probable than any of the other outcomes. That is,

P(6) = 2/3

whereas P(1) = P(2) = . . . = P(5) = 1/15. We can explore this using Sympy as in the following:

>>> # here 6 is ten times more probable than any other outcome

>>> x=stats.FiniteRV(’D3’,{1:1/15., 2:1/15.,

... ...

3:1/15., 4:1/15., 5:1/15., 6:2/3.})

As before, we construct the sum of the two dice, and plot the corresponding probability mass function in Fig. 2.6. As compared with Fig. 2.5, the probability mass has been shifted away from the smaller numbers.

Let’s see what the conditional expectation says about how we can estimate X from Z.

>>> E(x, S.Eq(z,7)) # conditional expectation E(x|z=7) 5.00000000000000

Now that we have E(x | z = 7) = 5, we can generate samples as before and see if this gives the minimum MSE.

Probability

Fig. 2.6 The values of Z are in yellow with the corresponding values for X and Y on the axes

>>> samples\_z7 = lambda : stats.sample(x, S.Eq(z,7))

>>> #using 6 as an estimate

>>> mn= np.mean([(6-samples\_z7())\*\*2 for i in range(100)])

>>> #5 is the MSE estimate

>>> mn0= np.mean([(5-samples\_z7())\*\*2 for i in range(100)])

>>> print(’MSE=%3.2f using 6 vs MSE=%3.2f using 5 ’ % (mn,mn0)) MSE=3.19 using 6 vs MSE=2.86 using 5

Using a simple example, we have emphasized the connection between minimum mean squared error problems and conditional expectation. Hopefully, the last two ﬁgures helped expose the role of the probability density. Next, we’ll continue revealing the true power of the conditional expectation as we continue to develop corresponding geometric intuition.

**2.5 Worked Examples of Conditional Expectation and Mean Square Error Optimization**

Brzezniak [6] is a great book because it approaches conditional expectation through a sequence of exercises, which is what we are trying to do here. The main difference is that Brzezniak takes a more abstract measure-theoretic approach to the same problems. Note that you do need to grasp measure theory for advanced areas in probability, but for what we have covered so far, working the same problems in his text using our methods is illuminating. It always helps to have more than one way to solve any problem. I have numbered the examples corresponding to the book and tried to follow its notation.

**2.5.1 Example**

This is Example 2.1 from Brzezniak. Three coins, 10, 20 and 50p are tossed. The values of the coins that land heads up are totaled. What is the expected total given that two coins have landed heads up? In this case we have we want to compute E( ξ|η ) where

ξ := 10X 10 + 20X 20 + 50X50

where X i ∈ { 0, 1 } and where X 10 is the Bernoulli-distributed random variable corresponding to the 10p coin (and so on). Thus, ξ represents the total value of the heads-up coins. The η represents the condition that only two of the three coins are heads-up,

η := X 10 X 20 (1 − X 50 ) + (1 − X 10 )X 20 X 50 + X 10 (1 − X 20 )X50

and is a function that is non-zero only when two of the three coins lands heads-up. Each triple term catches each of these three possibilities. For example, the ﬁrst term equals one when the 10 and 20p are heads up and the 50p is heads down. The the remaining terms are zero.

To compute the conditional expectation, we want to ﬁnd a function h of η that minimizes the mean-squared-error (MSE),

1 MSE = ∑ 3 ( ξ − h( η ))2 2 X∈ { 0,1 } 3

where the sum is taken over all possible triples of outcomes for { X 10 , X 20 , X 50 } because each of the three coins has a 2 1 chance of coming up heads.

Now, the question boils down to how can we characterize the function h( η )? Note that η ↦ → { 0, 1 } so h takes on only two values. So, the orthogonal inner product condition is the following:

〈ξ − h( η ), η〉 = 0

But, because are only interested in η = 1, this simpliﬁes to

〈ξ − h(1), 1 〉 = 0

〈ξ , 1 〉 = 〈 h(1), 1 〉

This doesn’t look so hard to evaluate but we have to compute the integral over the set where η = 1. In other words, we need the set of triples { X 10 , X 20 , X 50 } where η = 1. That is, we can compute

ξ d X = h(1) d X ∫ {η =1 } ∫ {η =1 } 70

2

Probability

which is what Brzezniak does. Instead, we can deﬁne h( η ) = α η and then ﬁnd α. Re-writing the orthogonal condition gives

〈ξ − η , α η〉 = 0

〈ξ , η〉 = α 〈η , η〉

〈ξ , α = η〉 〈η , η〉

where

1 〈ξ , η〉 = ∑ ( ξη ) 23 X∈ { 0,1 } 3

Note that we can just sweep over all triples { X 10 , X 20 , X 50 } because the deﬁnition of h( η ) zeros out when η = 0 anyway. All we have to do is plug everything in and solve. This tedious job is perfect for Sympy.

>>> import sympy as S

>>> X10,X20,X50 = S.symbols(’X10,X20,X50’,real=True)

>>> xi = 10\*X10+20\*X20+50\*X50

>>> eta = X10\*X20\*(1-X50)+X10\*(1-X20)\*(X50)+(1-X10)\*X20\*(X50)

>>> num=S.summation(xi\*eta,(X10,0,1),(X20,0,1),(X50,0,1))

>>> den=S.summation(eta\*eta,(X10,0,1),(X20,0,1),(X50,0,1))

>>> alpha=num/den

>>> print(alpha) # alpha=160/3 160/3

This means that

160 E( ξ|η ) = 3 η

which we can check with a quick simulation

>>> import pandas as pd

>>> d = pd.DataFrame(columns=[’X10’,’X20’,’X50’])

>>> d.X10 = np.random.randint(0,2,1000)

>>> d.X10 = np.random.randint(0,2,1000)

>>> d.X20 = np.random.randint(0,2,1000)

>>> d.X50 = np.random.randint(0,2,1000)

**Programming Tip**

The code above creates an empty Pandas data frame with the named columns. The next four lines assigns values to each of the columns.

The code above simulates ﬂipping the three coins 1000 times. Each column of the dataframe is either 0 or 1 corresponding to heads-down or heads-up, respectively. The condition is that two of the three coins have landed heads-up. Next, we can group the columns according to their sums. Note that the sum can only be in { 0, 1, 2, 3 } corresponding to 0 heads-up, 1 heads-up, and so on.

>>> grp=d.groupby(d.eval(’X10+X20+X50’))

**Programming Tip**

The eval function of the Pandas data frame takes the named columns and evaluates the given formula. At the time of this writing, only simple formulas involving primitive operations are possible.

Next, we can get the 2 group, which corresponds to exactly two coins having landed heads-up, and then evaluate the sum of the values of the coins. Finally, we can take the mean of these sums.

>>> grp.get\_group(2).eval(’10\*X10+20\*X20+50\*X50’).mean() 52.60162601626016

The result is close to 160/3=53.33 which supports the analytic result. The following code shows that we can accomplish the same simulation using pure Numpy.

>>> import numpy as np

>>> from numpy import array

>>> x=np.random.randint(0,2,(3,1000))

>>> print(np.dot(x[:,x.sum(axis=0)==2].T,array([10,20,50])).mean())

52.860759493670884

In this case, we used the Numpy dot product to compute the value of the heads-up coins. The sum(axis=0)==2 part selects the columns that correspond to two heads-up coins.

Still another way to get at the same problem is to forego the random sampling part and just consider all possibilities exhaustively using the itertools module in Python’s standard library.

>>> import itertools as it

>>> list(it.product((0,1),(0,1),(0,1))) [(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]

Note that we need to call list above in order to trigger the iteration in it.product. This is because the itertools module is generator-based so does not actually do the iteration until it is iterated over (by list in this case). This shows all possible triples (X 10 , X 20 , X 50 ) where 0 and 1 indicate heads-down and heads-up, respectively. The next step is to ﬁlter out the cases that correspond to two heads-up coins.

>>> list(filter(lambda i:sum(i)==2,it.product((0,1),(0,1),(0,1)))) [(0, 1, 1), (1, 0, 1), (1, 1, 0)]

Next, we need to compute the sum of the coins and combine the prior code.

>>> list(map(lambda k:10\*k[0]+20\*k[1]+50\*k[2], ...

filter(lambda

i:sum(i)==2,

... it.product((0,1),(0,1),(0,1))))) [70, 60, 30]

The mean of the output is 53.33, which is yet another way to get the same result. For this example, we demonstrated the full spectrum of approaches made possible using Sympy, Numpy, and Pandas. It is always valuable to have multiple ways of approaching the same problem and cross-checking the result.

**2.5.2 Example**

This is Example 2.2 from Brzezniak. Three coins, 10, 20 and 50p are tossed as before. What is the conditional expectation of the total amount shown by the three coins given the total amount shown by the 10 and 20p coins only? For this problem,

ξ :=10X 10 + 20X 20 + 50X50

η :=30X 10 X 20 + 20(1 − X 10 )X 20 + 10X 10 (1 − X 20 )

which takes on four values η ↦ → { 0, 10, 20, 30 } and only considers the 10p and 20p coins. In contrast to the last problem, here we are interested in h( η ) for all of the values of η . Naturally, there are only four values for h( η ) corresponding to each of these four values. Let’s ﬁrst consider η = 10. The orthogonal condition is then

〈ξ − h(10), 10 〉 = 0

The domain for η = 10 is { X 10 = 1, X 20 = 0, X 50 } which we can integrate out of the expectation below,

E { X 10 =1,X 20 =0,X 50 } ( ξ − h(10))10 = 0

E { X 50 } (10 − h(10) + 50X 50 ) = 0

10 − h(10) + 25 = 0

which gives h(10) = 35. Repeating the same process for η ∈ { 20, 30 } gives h(20) = 45 and h(30) = 55, respectively. This is the approach Brzezniak takes. On the other hand, we can just look at afﬁne functions, h( η ) = a η + b and use brute-force calculus.

>>> from sympy.abc import a,b

>>> h = a\*eta + b

>>> eta = X10\*X20\*30 + X10\*(1-X20)\*(10)+ (1-X10)\*X20\*(20)

>>> MSE=S.summation((xi-h)\*\*2\*S.Rational(1,8),(X10,0,1),

... (X20,0,1), ... (X50,0,1))

>>> sol=S.solve([S.diff(MSE,a),S.diff(MSE,b)],(a,b))

>>> print(sol) {a: 64/3, b: 32}

**Programming Tip**

The Rational function from Sympy code expresses a rational number that Sympy is able to manipulate as such. This is different that specifying a fraction like 1/8., which Python would automatically compute as a ﬂoating point number (i.e., 0.125). The advantage of using Rational is that Sympy can later produce rational numbers as output, which are sometimes easier to make sense of.

This means that

E( ξ|η ) = 25 + η

(2.5.2.1)

since η takes on only four values, { 0, 10, 20, 30 } , we can write this out explicitly as

⎪ ⎪ ⎧ 25 for η = 0 ⎪ 35 for η = 10 E( ξ|η ) = ⎪ ⎪ ⎨ 45 for η = 20 ⎪ ⎩ 55 for η = 30

(2.5.2.2)

Alternatively, we can use orthogonal inner products to write out the following conditions for the postulated afﬁne function:

〈ξ − h( η ), η〉 = 0

(2.5.2.3)

〈ξ − h( η ), 1 〉 = 0

(2.5.2.4)

Writing these out and solving for a and b is tedious and a perfect job for Sympy. Starting with Eq. 2.5.2.3,

>>> expr=S.expand((xi-h)\*eta)

>>> print(expr)

30\*X10\*\*2\*X20\*X50\*a - 10\*X10\*\*2\*X20\*a - 10\*X10\*\*2\*X50\*a + 100\*X10\*\*2 + 60\*X10\*X20\*\*2\*X50\*a - 20\*X10\*X20\*\*2\*a - 30\*X10\*X20\*X50\*a + 400\*X10\*X20 + 500\*X10\*X50 - 10\*X10\*b - 20\*X20\*\*2\*X50\*a + 400\*X20\*\*2 + 1000\*X20\*X50 - 20\*X20\*b 74

2

Probability

and then because E(X i 2 ) = 1/2 = E(X i ), we make the following substitutions

>>> expr.xreplace({X10\*\*2:0.5, X20\*\*2:0.5,X10:0.5,X20:0.5,X50:0.5}) -7.5\*a - 15.0\*b + 725.0

We can do this for the other orthogonal inner product in Eq. 2.5.2.4 as follows,

**Programming Tip**

Because Sympy symbols are hashable, they can be used as keys in Python dictionaries as in the xreplace function above.

>>> S.expand((xi-h)\*1).xreplace({X10\*\*2:0.5,

...

...

...

...

-0.375\*a - b + 40.0

X20\*\*2:0.5, X10:0.5, X20:0.5, X50:0.5})

Then, combining this result with the previous one and solving for a and b gives,

>>> S.solve([-350.0\*a-15.0\*b+725.0,-15.0\*a-b+40.0]) {a: 1.00000000000000, b: 25.0000000000000}

which again gives us the ﬁnal solution,

E( ξ|η ) = 25 + η

The following is a quick simulation to demonstrate this. We can build on the Pandas dataframe we used for the last example and create a new column for the sum of the 10p and 20p coins, as shown below.

>>> d[’sm’] = d.eval(’X10\*10+X20\*20’)

We can group this by the values of this sum,

>>> d.groupby(’sm’).mean() X10 X20 X50 sm 0 0.0 0.0 0.502024 10 1.0 0.0 0.531646 20 0.0 1.0 0.457831 30 1.0 1.0 0.516854

But we want the expectation of the value of the coins

>>> d.groupby(’sm’).mean().eval(’10\*X10+20\*X20+50\*X50’) sm 0 25.101215 10 36.582278 20 42.891566 30 55.842697 dtype: float64

which is very close to our analytical result in Eq. 2.5.2.2. 2.5 Worked Examples of Conditional Expectation and Mean Square Error Optimization

75

**2.5.3 Example**

This is Example 2.3 paraphrased from Brzezniak. Given X uniformly distributed on [ 0, 1 ] , ﬁnd E( ξ|η ) where

ξ (x) = 2x2

⎪ ⎧ 1 if x ∈ [ 0, 1/3 ] η (x) = ⎨ ⎪ 2 if x ∈ (1/3, 2/3) ⎩ 0 if x ∈ (2/3, 1 ]

Note that this problem is different from the previous two because the sets that characterize η are intervals instead of discrete points. Nonetheless, we will eventually have three values for h( η ) because η ↦ → { 0, 1, 2 } . For η = 1, we have the orthogonal conditions,

〈ξ − h(1), 1 〉 = 0

which boils down to

E { x∈ [ 0,1/3 ]} ( ξ − h(1)) = 0

1 3 (2x 2 − h(1))dx = 0 ∫ 0

and then by solving this for h(1) gives h(1) = 2/24. This is the way Brzezniak works this problem. Alternatively, we can use h( η ) = a + b η + c η 2 and brute force calculus.

>>> x,c,b,a=S.symbols(’x,c,b,a’)

>>> xi = 2\*x\*\*2

>>> eta=S.Piecewise((1,S.And(S.Gt(x,0),

... ... ... ... ...

S.Lt(x,S.Rational(1,3)))), # 0 < x < 1/3 (2,S.And(S.Gt(x,S.Rational(1,3)), S.Lt(x,S.Rational(2,3)))), # 1/3 < x < 2/3, (0,S.And(S.Gt(x,S.Rational(2,3)), S.Lt(x,1)))) # 1/3 < x < 2/3

>>> h = a + b\*eta + c\*eta\*\*2

>>> J=S.integrate((xi-h)\*\*2,(x,0,1))

>>> sol=S.solve([S.diff(J,a), ... S.diff(J,b), ... S.diff(J,c), ... ], ... (a,b,c))

>>> print(sol)

{a: 38/27, b: -20/9, c: 8/9}

>>> print(S.piecewise\_fold(h.subs(sol))) Piecewise((2/27, (x > 0) & (x < 1/3)),

(14/27, (x > 1/3) & (x < 2/3)),

(38/27, (x > 2/3) & (x < 1))) 76

2

Probability

Thus, collecting this result gives:

38 20 8 E( ξ|η ) = 2 + 27 9 η 9 η

which can be re-written as a piecewise function of x,

⎪ ⎧ 27 2 for 0 < x < 3 1 E( ξ|η (x)) = 14 27 for 3 1 < x < 2 3 ⎪ ⎨ 38 2 ⎩ 27 for 3 < x < 1

(2.5.3.1)

Alternatively, we can use the orthogonal inner product conditions directly by choosing h( η ) = c + η b + η 2 a,

〈ξ − h( η ), 1 〉 = 0

〈ξ − h( η ), η〉 = 0

〈ξ − h( η ), η 2 〉 = 0

and then solving for a,b, and c.

>>> x,a,b,c,eta = S.symbols(’x,a,b,c,eta’,real=True)

>>> xi = 2\*x\*\*2

>>> eta=S.Piecewise((1,S.And(S.Gt(x,0),

... ... ... ... ...

S.Lt(x,S.Rational(1,3)))), # 0 < x < 1/3 (2,S.And(S.Gt(x,S.Rational(1,3)), S.Lt(x,S.Rational(2,3)))), # 1/3 < x < 2/3, (0,S.And(S.Gt(x,S.Rational(2,3)),

S.Lt(x,1))))

#

1/3

<

x

<

2/3

>>> h = c+b\*eta+a\*eta\*\*2

Then, the orthogonal conditions become,

>>> S.integrate((xi-h)\*1,(x,0,1))

-5\*a/3 - b - c + 2/3

>>> S.integrate((xi-h)\*eta,(x,0,1))

-3\*a - 5\*b/3 - c + 10/27

>>> S.integrate((xi-h)\*eta\*\*2,(x,0,1))

-17\*a/3 - 3\*b - 5\*c/3 + 58/81

Now, we just combine the three equations and solve for the parameters,

>>> eqs=[ -5\*a/3 - b - c + 2/3,

... -3\*a - 5\*b/3 - c + 10/27, ... -17\*a/3 - 3\*b - 5\*c/3 + 58/81]

>>> sol=S.solve(eqs)

>>> print(sol)

{a: 0.888888888888889, b: -2.22222222222222, c: 1.40740740740741} 2.5 Worked Examples of Conditional Expectation and Mean Square Error Optimization

77

We can assemble the ﬁnal result by substituting in the solution,

>>> print(S.piecewise\_fold(h.subs(sol))) Piecewise((0.074074074074074, (x > 0) & (x < 1/3)), (0.518518518518518, (x > 1/3) & (x < 2/3)), (1.40740740740741, (x > 2/3) & (x < 1)))

which is the same as our analytic result in Eq. 2.5.3.1, just in decimal format.

**Programming Tip**

The deﬁnition of Sympy’s piecewise function is verbose because of the way Python parses inequality statements. As of this writing, this has not been reconciled in Sympy, so we have to use the verbose declaration.

To reinforce our result, let’s do a quick simulation using Pandas.

>>> d = pd.DataFrame(columns=[’x’,’eta’,’xi’])

>>> d.x = np.random.rand(1000)

>>> d.xi = 2\*d.x\*\*2

>>> d.xi.head() 0 0.649201 1 1.213763 2 1.225751 3 0.005203 4 0.216274 Name: xi, dtype: float64

Now, we can use the pd.cut function to group the x values in the following,

>>> pd.cut(d.x,[0,1/3,2/3,1]).head() 0 (0.333, 0.667] 1 (0.667, 1.0] 2 (0.667, 1.0] 3 (0.0, 0.333] 4 (0.0, 0.333] Name: x, dtype: category Categories (3, interval[float64]): [(0.0, 0.333] < (0.333, 0.667] < (0.667, 1.0]]

Note that the head() call above is only to limit the printout shown. The categories listed are each of the intervals for eta that we speciﬁed using the [0,1/3,2/3,1] list. Now that we know how to use pd.cut, we can just compute the mean on each group as shown below,

>>> d.groupby(pd.cut(d.x,[0,1/3,2/3,1])).mean()[’xi’] x (0.0, 0.333] 0.073048 (0.333, 0.667] 0.524023 (0.667, 1.0] 1.397096 Name: xi, dtype: float64 78

2

Probability

whichisprettyclosetoouranalyticresultinEq.2.5.3.1.Alternatively,sympy.stats has some limited tools for the same calculation.

>>> from sympy.stats import E, Uniform

>>> x=Uniform(’x’,0,1)

>>> E(2\*x\*\*2,S.And(x < S.Rational(1,3), x > 0)) 2/27

>>> E(2\*x\*\*2,S.And(x < S.Rational(2,3), x > S.Rational(1,3))) 14/27

>>> E(2\*x\*\*2,S.And(x < 1, x > S.Rational(2,3))) 38/27

which again gives the same result still another way.

**2.5.4 Example**

This is Example 2.4 from Brzezniak. Find E( ξ|η ) for

ξ (x) = 2x2

2 if 0 ≤ x < 1 = 2 η 1 { x if 2 < x ≤ 1

Once again, X is uniformly distributed on the unit interval. Note that η is no longer discrete for every domain. For the domain 0 < x < 1/2, h(2) takes on only one value, say, h 0 . For this domain, the orthogonal condition becomes,

E {η =2 } (( ξ (x) − h 0 )2) = 0

which simpliﬁes to,

1/2 2x 2 − h 0 dx = 0 ∫ 0

1/2 1/2 2x 2 dx = h 0 dx ∫ 0 ∫ 0 1/2 h 0 = 2 2x 2 dx ∫ 0 1 h0 = 6

For the other domain where {η = x } in Eq. 2.5.4, we again use the orthogonal condition, 2.5 Worked Examples of Conditional Expectation and Mean Square Error Optimization

79

E {η =x } (( ξ (x) − h(x))x) = 0

1 (2x 2 − h(x))xdx = 0 ∫ 1/2

h(x) = 2x2

Assembling the solution gives,

1 1 for 0 ≤ x < E( ξ|η (x)) = 6 2 1 { 2x 2 for 2 < x ≤ 1

although this result is not explicitly written as a function of η .

**2.5.5 Example**

This is Exercise 2.6 in Brzezniak. Find E( ξ|η ) where

ξ (x) = 2x2

η (x) = 1 − | 2x − 1 |

and X is uniformly distributed in the unit interval. We can write this out as a piecewise function in the following,

2x for 0 ≤ x < 1 = 2 η 1 { 2 − 2x for 2 < x ≤ 1

The discontinuity is at x = 1/2. Let’s start with the {η = 2x } domain.

E {η =2x } ((2x 2 − h(2x))2x) = 0

1/2 (2x 2 − h(2x))2xdx = 0 ∫ 0

We can make this explicitly a function of η by a change of variables ( η = 2x) which gives

1 ( η 2 /2 − h( η )) η d η =0 ∫ 0 2

Thus, for this domain, h( η ) = η 2 /2. Note that due to the change of variables, h( η ) is valid deﬁned over η ∈ [ 0, 1 ] . 80

2

Probability

For the other domain where {η = 2 − 2x } , we have

E {η =2−2x } ((2x 2 − h(2 − 2x))(2 − 2x)) = 0

1 (2x 2 − h(2 − 2x))(2 − 2x)dx = 0 ∫ 1/2

Once again, a change of variables makes the η dependency explicit using η = 2−2x which gives

1 ((2 − η ) 2 /2 − h( η )) η d η =0 ∫ 0 2

h( η ) = (2 − η ) 2 /2

Once again, the change of variables means this solution is valid over η ∈ [ 0, 1 ] . Thus, because both pieces are valid over the same domain ( η ∈ [ 0, 1 ] ), we can just add them to get the ﬁnal solution,

h( η ) = η 2 − 2 η + 2

A quick simulation can help bear this out.

>>> from pandas import DataFrame

>>> import numpy as np

>>> d = DataFrame(columns=[’xi’,’eta’,’x’,’h’,’h1’,’h2’])

>>> # 100 random samples

>>> d.x = np.random.rand(100)

>>> d.xi = d.eval(’2\*x\*\*2’)

>>> d.eta =1-abs(2\*d.x-1)

>>> d.h1=d[(d.x<0.5)].eval(’eta\*\*2/2’)

>>> d.h2=d[(d.x>=0.5)].eval(’(2-eta)\*\*2/2’)

>>> d.fillna(0,inplace=True)

>>> d.h = d.h1+d.h2

>>> d.head() xi eta x h 0 1.102459 0.515104 0.742448 1.102459 1 0.239610 0.692257 0.346128 0.239610 2 1.811868 0.096389 0.951806 1.811868 3 0.000271 0.023268 0.011634 0.000271 4 0.284240 0.753977 0.376988 0.284240

h1 h2

0.000000 1.102459

0.239610 0.000000

0.000000 1.811868

0.000271 0.000000

0.284240 0.000000

Note that we have to be careful where we apply the individual solutions using the slice (d.x<0.5) index. The fillna part ensures that the default NaN that ﬁlls out the empty row-etries is replaced with zero before combining the individual solutions. Otherwise, the NaN values would circulate through the rest of the computation. The following is the essential code that draws Fig. 2.7. 2.5 Worked Examples of Conditional Expectation and Mean Square Error Optimization

81

Fig. 2.7 The diagonal line shows where the conditional expectation equals the ξ function

from matplotlib.pyplot import subplots fig,ax=subplots() ax.plot(d.xi,d.eta,’.’,alpha=.3,label=’$\eta$’) ax.plot(d.xi,d.h,’k.’,label=’$h(\eta)$’) ax.legend(loc=0,fontsize=18) ax.set\_xlabel(’$2 xˆ2$’,fontsize=18) ax.set\_ylabel(’$h(\eta)$’,fontsize=18)

**Programming Tip**

A Basic L T E X formatting works for the labels in Fig. 2.7. The loc=0 in the legend function is the code for the best placement for the labels in the legend. The individual labels should be speciﬁed when the elements are drawn individually, otherwise they will be hard to separate out later. This is accomplished using the label keyword in the plot commands.

Figure 2.7 shows the ξ data plotted against η and h( η ) = E( ξ|η ). Points on the diagonal are points where ξ and E( ξ|η ) match. As shown by the dots, there is no agreement between the raw η data and ξ . Thus, one way to think about the conditional expectation is as a functional transform that bends the curve onto the diagonal line. The black dots plot ξ versus E( ξ|η ) and the two match everywhere along the diagonal line. This is to be expected because the conditional expectation is the MSE best estimate for ξ among all functions of η . 82

2

Probability

**2.5.6 Example**

This is Exercise 2.14 from Brzezniak. Find E( ξ|η ) where

ξ (x) = 2x2

2x if 0 ≤ x < 1 = 2 η 1 { 2x − 1 if 2 < x ≤ 1

and X is uniformly distributed in the unit interval. This is the same as the last example and the only difference here is that η is not continuous at x = 2 1 , as before. The ﬁrst part is exactly the same as the ﬁrst part of the prior example so we will skip it here. The second part follows the same reasoning as the last example, so we will just write the answer for the {η = 2x − 1 } case as the following

(1 + )2 h( η ) = η , ∀ η ∈ [ 0, 1 ] 2

and then adding these up as before gives the full solution:

1 h( η ) = 2 + + 2 η η

The interesting part about this example is shown in Fig. 2.8. The dots show where η is discontinuous and yet the h( η ) = E( ξ|η ) solution is equal to ξ (i.e., matches the diagonal). This illustrates the power of the orthogonal inner product technique,

Fig. 2.8 The diagonal line shows where the conditional expectation equals the ξ function 2.5 Worked Examples of Conditional Expectation and Mean Square Error Optimization

83

which does not need continuity or complex set-theoretic arguments to calculate solutions. By contrast, I urge you to consider Brzezniak’s solution to this problem which requires such methods.

Extending projection methods to random variables provides multiple ways for calculating solutions to conditional expectation problems. In this section, we also worked out corresponding simulations using a variety of Python modules. It is always advisable to have more than one technique at hand to cross-check potential solutions. We worked out some of the examples in Brzezniak’s book using our methods as a way to show multiple ways to solve the same problem. Comparing Brzezniak’s measuretheoretic methods to our less abstract techniques is a great way to get a handle on both concepts, which are important for advanced study in stochastic process.

**2.6 Useful Distributions**

**2.6.1 Normal Distribution**

Without a doubt, the normal (Gaussian) distribution is the most important and foundational probability distribution. The one-dimensional form is the following:

e− (x− 2σ μ 2 ) 2 f (x) = √ 2πσ2

where E(x) = μ and V(x) = σ 2 . The multidimension al version for x ∈ R n is the following,

1 f (x) = e− 1 2 (x− μ ) T R −1 (x− μ ) 1 det(2πR) 2

where R is the covariance matrix with entries

R i, j = E [ (x i − ¯x i )(x j − ¯x j ) ]

A key property of the normal distribution is that it is completely speciﬁed by its ﬁrst two moments. Another key property is that the normal distribution is preserved under linear transformations. For example,

y = Ax

means y ∼ N(Ax, AR x A T ). This means that it is easy to do linear algebra and matrix operations with normal distributed random variables. There are many intuitive geometric relationships that are preserved with normal distributed random variables, as discussed in the Gauss-Markov chapter. 84

2

Probability

**2.6.2 Multinomial Distribution**

The Multinomial distribution generalized the Binomial distribution. Recall that the Binomial distribution characterizes the number of heads obtained in n trials.

Consider the problem of n balls to be divided among r available bins where each bin may accommodate more than one ball. For example, suppose n = 10 and and r = 3, then one possible valid conﬁguration is N 10 = [ 3, 3, 4 ] . The probability that a ball lands in the i th bin is p i , where ∑ p i = 1. The Multinomial distribution characterizes the probability distribution of N n . The Binomial distribution is a special case of the Multinomial distribution with n = 2. The Multinomial distribution is implemented in the scipy.stats module as shown below,

>>> from scipy.stats import multinomial

>>> rv = multinomial(10,[1/3]\*3)

>>> rv.rvs(4) array([[2, 2, 6], [4, 2, 4], [2, 4, 4], [2, 6, 2]])

Note that the sum across the columns is always n

>>> rv.rvs(10).sum(axis=1) array([10, 10, 10, 10, 10, 10, 10, 10, 10, 10])

To derive the probability mass function, we deﬁne the occupancy vector, e i ∈ Rr which is a binary vector with exactly one non-zero component (i.e., a unit vector). Then, the N n vector can be written as the sum of n vectors X, each drawn from the set { e j } j=1 r ,

n N n = ∑ Xi i=1

where the probability P(X = e j ) = p j . Thus, N n has a discrete distribution over the set of vectors with non-negative components that sum to n. Because the X vectors are independent and identically distributed, the probability of any particular N n =

⊤ [ x 1 , x 2 , . . . , x r ] = x is

P(N n = x) = C n p 1 x 1 p 2 x 2 · · · pr x r

where C n is a combinatorial factor that accounts for all the ways a component can sum to x j . Consider that there are (x n 1 ) ways that the ﬁrst component can be chosen. This leaves n − x 1 balls left for the rest of the vector components. Thus, the second component has ( n−x x 2 1 ) 2 ways to pick a ball. Following the same pattern, the third

component has ( n−x x 1 3 −x

) ways and so forth,

n n − x 1 n − x − x 2 n − x 1 − x 2 − ··· − xr−1 C n = ··· ( x 1 )( x 2 )( x 3 1 ) ( x r ) 2.6 Useful Distributions

85

simpliﬁes to the following,

n! Cn = x 1 ! ···x r !

Thus, the probability mass function for the Multinomial distribution is the following,

n! P(N n = x) = p 2 x 2 · · · pr x r p1 x 1 x 1 ! ···x r !

The expectation of this distribution is the following,

n E(N n ) = ∑ E(X i ) i=1

by the linearity of the expectation. Then,

r E(X i ) = ∑ p j e j = Ip = p j=1

where p j are the components of the vector p and I is the identity matrix. Then, because this is the same for any X i , we have

E(N n ) = np

For the covariance of Nn

, we need to compute the following,

Cov(N n ) = E N n ⊤ )E(N n )⊤ Nn E(Nn ( )

For the ﬁrst term on the right, we have

n n N n ⊤ ⎛ ( X i )( X j ⊤ ) ⎞ E Nn = E ( ) ∑ ∑ ⎝ i=1 j=1 ⎠

and for i = j, we have

E(X i X i ⊤ ) = diag(p)

and for i = ̸ j, we have

E(X i X j ⊤ ) = pp ⊤

Note that this term has elements on the diagonal. Then, combining the above two equations gives the following,

E(N n N n ⊤ ) = ndiag(p) + (n 2 − n)pp⊤ 86

2

Probability

Now, we can assemble the covariance matrix,

Cov(N n ) = ndiag(p) + (n 2 − n)pp ⊤ − n 2 pp ⊤ = ndiag(p) − npp⊤

Speciﬁcally, the off-diagonal terms are np i p j and the diagonal terms are np i (1− p i ).

**2.6.3 Chi-square Distribution**

The χ 2 distribution appears in many different contexts so it’s worth understanding. Suppose we have n independent random variables Xi such that X i ∼ N(0, 1). We are interested in the following random variable $R = \sqrt{X\_i^2}$. The joint probability density of Xi is the following,

X i 2 e− 2 1 ∑ i f X (X) = n (2π) 2

√ ∑i

X i 2 . The joint probability

where the X represents a vector of X i random variables. You can think of R as the radius of an n-dimensional sphere. The volume of this sphere is given by the the following formula,

n

π2 V n (R) = Rn Γ ( n + 1) 2

To reduce the amount of notation we deﬁne,

n π2

A := Γ ( n + 1) 2

The differential of this volume is the following,

dV n (R) = nAR n−1 d R

In term of the X i coordinates, the probability (as always) integrates out to one.

f X (X)dV n (X) = 1 ∫

In terms of R, the change of variable provides,

f X (R)nAR n−1 d R ∫ 2.6 Useful Distributions

87

Thus,

en − 2 1 R 2 f R (R) := f X (R) = nARn−1 (2π) 2

But we are interested in the distribution Y = R 2 . Using the same technique again,

dY f R (R)d R = f R ( √ Y) ∫ ∫ 2 √ Y

Finally,

n−1 e − 1 2 Y 1 f Y (Y) := nAY 2 n (2π) 2 2 √ Y

Then, ﬁnally substituting back in A gives the χ 2 distribution with n degrees of freedom,

n n π 2 e n 2 − 1 2 Y 1 2 − 2 −1 n −1 f Y (Y) = n Yn 2 Yn/2−1 = e−Y/2 Γ ( n + 1) (2π) 2 Γ ( n + 1 ) 2 2

**Example**: Hypothesis testing is a common application of the χ 2 distribution. Consider Table 2.1 which tabulates the infection status of a certain population. The hypothesis is that these data are distributed according to the multinomial distribution with the following rates for each group, p 1 = 1/4 (mild infection), p 2 = 1/4 (strong infection), and p 3 = 1/2 (no infection). Suppose n i is the count of persons in the ith column and ∑ i n i = n = 684. Let k denote the number of columns. Then, in order to apply the Central Limit Theorem, we want to sum the n i random variables, but these all sum to n, a constant, which prohibits using the theorem. Instead, suppose we sum the n i variables up to k − 1 terms. Then,

k−1 z = ∑ ni i=1

is asymptotically normally distributed by the theorem with mean E(z) = ∑ i=1 k−1 np i . Using our previous results and notation for multinomial random variables, we can write this as

z = [ 1 k−1 ⊤ , 0 ] Nn

Table 2.1 Diagnosis table

Mild infection Strong infection 128 136

No infection 420

Total 684 88

2

Probability

where 1 k−1 is a vector of all ones of length k − 1 and Nn we have

∈

R k . With this notation,

k−1 E(z) = n [ 1 k−1 ⊤ , 0 ] p = ∑ np i = n(1 − p k ) i=1

We can get the variance of z using the same method,

V(z) = [ 1 k−1 ⊤ , 0 ] Cov(N n ) [ 1 k−1 ⊤ , 0 ] ⊤

which gives,

V(z) = [ 1 k−1 ⊤ , 0 ] (ndiag(p) − npp ⊤ ) [ 1 k−1 ⊤ , 0 ]⊤

The variance is then,

V(z) = n(1 − p k )pk

With the mean and variance established we can subtract the hypothesize mean for each column under the hypothesis and create the transformed variable,

k−1 n i − npi ′ z = ∑ n(1 − p k )p k ∼ N(0, 1) i=1 √

by the Central Limit Theorem. Likewise,

k−1 (n i − np i )2 2 ∑ n(1 − p k )p k ∼ χk−1 i=1

With all that established, we can test the hypothesis that the data in the table follow the hypothesized multinomial distribution.

>>> from scipy import stats

>>> n = 684

>>> p1 = p2 = 1/4

>>> p3 = 1/2

>>> v = n\*p3\*(1-p3)

>>> z = (128-n\*p1)\*\*2/v + (136-n\*p2)\*\*2/v

>>> 1-stats.chi2(2).cdf(z)

0.00012486166748693073

This value is very low and suggests that the hypothesized multinomial distribution is not a good one for this data. Note that this approximation only works when n is large in comparison to the number of columns in the table. 2.6 Useful Distributions

89

**2.6.4 Poisson and Exponential Distributions**

The Poisson distribution for a random variable X represents a number of outcomes occurring in a given time interval (t).

e−λt (λt)x p(x ; λt) = x!

The Poisson distribution is closely related to the binomial distribution, b(k ; n, p) where p is small and n is large. That is, when there is a low-probability event but many trials, n. Recall that the binomial distribution is the following,

n b(k ; n, p) = (1 − p)n−k pk ( k )

for k = 0 and taking the logarithm of both sides, we obtain

n λ log b(0 ; n, p) = (1 − p) n = 1 ( n )

Then, the Taylor expansion of this gives the following,

λ2 log b(0 ; n, p) ≈ −λ · · · 2n

For large n, this results in,

b(0 ; n, p) ≈ e−λ

A similar argument for k leads to the Poisson distribution. Conveniently, we have E(X) = V(X) = λ. For example, suppose that the average number of vehicles passing under a toll-gate per hour is 3. Then, the probability that 6 vehicles pass under the gate in a given hour is = 6 ; = 3) = ≈ 0.05. p(x λt 81 30e 3 The Poisson distribution is available from the scipy.stats module. The following code computes the last result,

>>> from scipy.stats import poisson

>>> x = poisson(3)

>>> print(x.pmf(6))

0.05040940672246224

The Poisson distribution is important for applications involving reliability and queueing. The Poisson distribution is used to compute the probability of speciﬁc numbers of events during a particular time period. In many cases the time period (X) itself is the random variable. For example, we might be interested in understanding the time X between arrivals of vehicles at a checkpoint. With the Poisson distribution, the probability of no events occurring in the span of time up to time t is given by the 90

2

Probability

following,

p(0 ; λt) = e−λt

Now, suppose X is the time to the ﬁrst event. The probability that the length of time until the ﬁrst event will exceed x is given by the following,

P(X > x) = e−λx

Then, the cumulative distribution function is given by the following,

P(0 ≤ X ≤ x) = F X (x) = 1 − e−λx

Taking the derivative gives the exponential distribution,

f X (x) = λe−λx

1 where E(X) = 1/λ and V(X) = λ 2 . For example, suppose we want to know the probability of a certain component lasting beyond T = 10 years where T is modeled as a an exponential random variable with 1/λ = 5 years. Then, we have 1 − F X (10) = e −2 ≈ 0.135. The exponential distribution is available in the scipy.stats module. The following code computes the result of the example above. Note that the parameters are described in slightly different terms as above, as described in the corresponding documentation for expon.

>>> from scipy.stats import expon

>>> x = expon(0,5) # create random variable object

>>> print(1 - x.cdf(10))

0.1353352832366127

**2.6.5 Gamma Distribution**

We have previously discussed how the exponential distribution can be created from thePoissonevents.Theexponentialdistributionhasthememorylessproperty,namely,

P(T > t 0 + t | T > t 0 ) = P(T > t)

For example, given T as the random variable representing the time until failure, this means that a component that has survived up through t 0 has the same failure probability of lasting t units beyond that point. To derive this result, it is easier to compute the complementary event,

P(t 0 < T < t 0 + t | T > t 0 ) = P(t 0 < T < t 0 + t) = e −λt − 1 eλt ( ) 2.6 Useful Distributions

91

Then, one minus this result shows the memoryless property, which, unrealistically, does not account for wear over the ﬁrst t hours. The gamma distribution can remedy this.

Recall that the exponential distribution describes the time until the occurrence of a Poisson event, the random variable X for the time until a speciﬁed number of Poisson events (α) is described by the gamma distribution. Thus, the exponential distribution is a special case of the gamma distribution when α = 1 and β = 1/λ. For x > 0, the gamma distribution is the following,

xα−1 e− β x β−α f (x ; α, β ) = Γ (α)

and f (x ; α, β ) = 0 when x ≤ 0 and Γ is the gamma function. For example, suppose that vehicles passing under a gate follows a Poisson process, with an average of 5 vehicles passing per hour, what is the probability that at most an hour will have passed before 2 vehicles pass the gate? If X is time in hours that transpires before the 2 vehicles pass, then we have β = 1/5 and α = 2. The required probability P(X < 1) ≈ 0.96. The gamma distribution has E(X) = α β and V(X) = α β2 The following code computes the result of the example above. Note that the parameters are described in slightly different terms as above, as described in the corresponding documentation for gamma.

>>> from scipy.stats import gamma

>>> x = gamma(2,scale=1/5) # create random variable object

>>> print(x.cdf(1))

0.9595723180054873

2.6.6 Beta Distribution

The uniform distribution assigns a single constant value over the unit interval. The Beta distribution generalizes this to a function over the unit interval. The probability density function of the Beta distribution is the following,

1 f (x) = (1 − x)b−1 xa−1 β (a, b)

where

1 β (a, b) = x a−1 (1 − x) b−1 dx ∫ 0

Note that a = b = 1 yields the uniform distribution. In the special case for integers where 0 ≤ k ≤ n, we have 92

2

Probability

1 n 1 xk (1 − x) n−k dx = ∫ ( k ) n + 1 0

To get this result without calculus, we can use an experiment by Thomas Bayes. Start with n white balls and one gray ball. Uniformly at random, toss them onto the unit interval. Let X be the number of white balls to the left of the gray ball. Thus, X ∈ { 0, 1, . . . , n } . To compute P(X = k), we condition on the probability of the position B of the gray ball, which is uniformly distributed over the unit interval ( f (p) = 1). Thus, we have

1 1 n P(X = k) = P(X = k | B = p) f (p)dp = (1 − p) n−k dp pk 0 0 ∫ ∫ ( k )

Now, consider a slight variation on the experiment where we start with n + 1 white balls and again toss them onto the unit interval and then later choose one ball at random to color gray. Using the same X as before, by symmetry, because any one of the n + 1 balls is equally likely to be chosen, we have

1 P(X = k) = n + 1

for k ∈ { 0, 1, . . . , n } . Both situations describe the same problem because it does not matter whether we paint the ball before or after we throw it. Setting the last two equations equal gives the desired result without using calculus.

1 n 1 pk (1 − p) n−k dp = ∫ ( k ) n + 1 0

The following code shows where to get the Beta distribution from the scipy module.

>>> from scipy.stats import beta

>>> x = beta(1,1) # create random variable object

>>> print(x.cdf(1))

1.0

Given this experiment, it is not too surprising that there is an intimate relationship between the Beta distribution and binomial random variables. Suppose we want to estimate the probability of heads for coin-tosses using Bayesian inference. Using this approach, all unknown quantities are treated as random variables. In this case, the probability of heads (p) is the unknown quantity that requires a prior distribution. Let us choose the Beta distribution as the prior distribution, Beta(a, b). Then, conditioning on p, we have

X | p ∼ binom(n, p) 2.6 Useful Distributions

93

which says that X is conditionally distributed as a binomial. To get the posterior probability, f (p | X = k), we have the following Bayes rule,

P(X = k | p) f (p) f (p | X = k) = P(X = k)

with the corresponding denominator,

1 n P(X = k) = (1 − p) n−k f (p)dp pk 0 ∫ ( k )

Note that unlike with our experiment before, f (p) is not constant. Without substituting in all of the distributions, we observe that the posterior is a function of p which means that everything else that is not a function of p is a constant. This gives,

f (p | X = k) ∝ p a + k−1 (1 − p)b + n−k−1

which is another Beta distribution with parameters a + k, b + n − k. This special relationship in which the beta prior probability distribution on p on data that are conditionallybinomialdistributedyieldstheposteriorthatisalsobinomialdistributed is known as conjugacy. We say that the Beta distribution is the conjugate prior of the binomial distribution.

**2.6.7 Dirichlet-Multinomial Distribution**

The Dirichlet-multinomial distribution is a discrete multivariate distribution also known as the multivariate Polya distribution. The Dirichlet-multinomial distribution arises in situations where the usual multinomial distribution is inadequate. For example, if a multinomial distribution is used to model the number of balls that land in a set of bins and the multinomial parameter vector (i.e., probabilities of balls landing in particular bins) varies from trial to trial, then the Dirichlet distribution can be used to include variation in those probabilities because the Dirichlet distribution is deﬁned over a simplex that describes the multinomial parameter vector.

Speciﬁcally, suppose we have K rival events, each with probability μ k . Then, the probability of the vector μ given that each event has been observed α k times is the following,

K P( μ| α) ∝ ∏ μk α k −1 k=1

where 0 ≤ μ k ≤ 1 and ∑ μ k = 1. Note that this last sum is a constraint that makes the distribution K − 1 dimensional. The normalizing constant for this distribution is the multinomial Beta function, 94

2

Probability

Fig. 2.9 One thousand samples from a Dirichlet distribution with α = [ 1, 1, 1 ]

Γ (α k ) Beta(α) = ∏k=1 K Γ ( ∑ k=1 K α k )

The elements of the α vector are also called concentration parameters. As before, the Dirichlet distribution can be found in the scipy.stats module,

>>> from scipy.stats import dirichlet

>>> d = dirichlet([ 1,1,1 ])

>>> d.rvs(3) # get samples from distribution array([[0.33938968, 0.62186914, 0.03874119], [0.21593733, 0.54123298, 0.24282969], [0.37483713, 0.07830673, 0.54685613]])

Note that each of the rows sums to one. This is because of the ∑ μ k = 1 constraint. We can generate more samples and plot this using Axes3D in Matplotlib in Fig. 2.9.

Notice that the generated samples lie on the triangular simplex shown. The corners of the triangle correspond to each of the components in the μ . Using, a non-uniform α = [ 2, 3, 4 ] vector, we can visualize the probability density function using the pdf method on the dirichlet object as shown in Fig. 2.10. By choosing the α ∈ R 3 , the peak of the density function can be moved within the corresponding triangular simplex.

We have seen that the Beta distribution generalizes the uniform distribution over the unit interval. Likewise, the Dirichlet distribution generalizes the Beta distribution over a vector with components in the unit interval. Recall that binomial distribution and the Beta distribution form a conjugate pair for Bayesian inference because with p ∼ Beta, X | p ∼ Binomial(n, p)

That is, the data conditioned on p, is binomial distributed. Analogously, the multinomial distribution and the Dirichlet distribution also form such a conjugate pair with 2.6 Useful Distributions

95

Fig. 2.10 Probability density function for the Dirichlet distribution with α = [ 2, 3, 4 ]

multinomial parameter p ∼ Dirichlet,

X | p ∼ multinomial(n, p)

For this reason, the Dirichlet-multinomial distribution is popular in machine learning text processing because non-zero probabilities can be assigned to words not speciﬁcally contained in speciﬁc documents, which helps generalization performance.

**2.7 Information Entropy**

We are in a position to discuss information entropy. This will give us a powerful perspective on how information passes between experiments, and will prove important in certain machine learning algorithms.

There used to be a TV game show where the host would hide a prize behind one of three doors and the contestant would have to pick one of the doors. However, before opening the door of the contestant’s choice, the host would open one of the other doors and ask the contestant if she wanted to change her selection. This is the classic Monty Hall problem. The question is should the contestant stay with her original choice or switch after seeing what the host has revealed? From the information theory perspective, does the information environment change when the host reveals what is behind one of the doors? The important detail here is that the host never opens the door with the prize behind it, regardless of the contestant’s choice. That is, the host knows where the prize is, but he does not reveal that information directly to the contestant. This is the fundamental problem information theory addresses — how to aggregate and reason about partial information. We need a concept of information that can accommodate this kind of question. 96

2

Probability

**2.7.1 Information Theory Concepts**

The Shannon information content of an outcome x is deﬁned as,

1 h(x) = log2 P(x)

where P(x) is the probability of x. The entropy of the ensemble X is deﬁned to be the Shannon information content of

1 = H(X) ∑ P(x) log 2 P(x) x

It is no accident that the entropy has this functional form as the expectation of h(x). It leads to a deep and powerful theory of information.

To get some intuition about what information entropy means, consider a sequence of three-bit numbers where each individual bit is equally likely. Thus, the individual information content of a single bit is h(x) = log 2 (2) = 1. The units of entropy are bits so this says that information content of a single bit is one bit. Because the three-bit number has elements that are mutually independent and equally likely, the information entropy of the three-bit number is h(X) = 2 3 × log 2 (2 3 )/8 = 3. Thus, the basic idea of information content at least makes sense at this level.

A better way to interpret this question is as how much information would I have to provide in order to uniquely encode an arbitrary three-bit number? In this case, you would have to answer three questions: Is the ﬁrst bit zero or one? Is the second bit zero or one? Is the third bit zero or one? Answering these questions uniquely speciﬁes the unknown three-bit number. Because the bits are mutually independent, knowing the state of any of the bits does not inform the remainder.

Next, let’s consider a situation that lacks this mutual independence. Suppose in a group of nine otherwise identical balls there is a heavier one. Furthermore, we also have a measuring scale that indicates whether one side is heavier, lighter, or equal to the other. How could we identify the heavier ball? At the outset, the information content, which measures the uncertainty of the situation is log 2 (9) because one of the nine balls is heavier. Figure 2.11 shows one strategy. We could arbitrarily select out one of the balls (shown by the square), leaving the remaining eight to be balanced. The thick, black horizontal line indicates the scale. The items below and above this line indicate the counterbalanced sides of the scale.

If we get lucky, the scale will report that the group of four walls on either side of the balance are equal in weight. This means that the ball that was omitted is the heavier one. This is indicated by the hashed left-pointing arrow. In this case, all the uncertainty has evaporated, and the informational value of that one weighing is equal to log 2 (9). In other words, the scale has reduced the uncertainty to zero (i.e., found the heavy ball). On the other hand, the scale could report that the upper group of four balls is heavier (black, upward-pointing arrow) or lighter (gray, downward-pointing 2.7 Information Entropy

97

Fig. 2.11 One heavy ball is hidden among eight identical balls. By weighing groups sequentially, we can determine the heavy ball

Fig. 2.12 For this strategy, the balls are broken up into three groups of equal size and subsequently weighed

arrow). In this case, we cannot isolate the heavier ball until we perform all of the indicated weighings, moving from left-to-right. Speciﬁcally, the four balls on the heavier side have to be split by a subsequent weighing into two balls and then to one ball before the heavy ball can be identiﬁed. Thus, this process takes three weighings. The ﬁrst one has information content log 2 (9/8), the next has log 2 (4), and the ﬁnal one has log 2 (2). Adding all these up sums to log 2 (9). Thus, whether or not the heavier ball is isolated in the ﬁrst weighing, the strategy consumes log 2 (9) bits, as it must, to ﬁnd the heavy ball.

However, this is not the only strategy. Figure 2.12 shows another. In this approach, the nine balls are split up into three groups of three balls apiece. Two groups are weighed. If they are of equal weight, then this means the heavier ball is in the group 98

2

Probability

that was left out (dashed arrow). Then, this group is split into two groups, with one element left out. If the two balls on the scale weigh the same, then it means the excluded one is the heavy one. Otherwise, it is one of the balls on the scale. The same process follows if one of the initially weighed groups is heavier (black upward-facing arrow) or lighter (gray lower-facing arrow). As before the information content of the situation is log 2 (9). The ﬁrst weighing reduces the uncertainty of the situation by log 2 (3) and the subsequent weighing reduces it by another log 2 (3). As before, these sum to log 2 (9), but here we only need two weighings whereas the ﬁrst strategy in Fig. 2.11 takes an average of 1/9 + 3 ∗ 8/9 ≈ 2.78 weighings, which is more than two from the second strategy in Fig. 2.12.

Why does the second strategy use fewer weighings? To reduce weighings, we need each weighing to adjudicate equally probable situations as many times as possible. Choosing one of the nine balls at the outset (i.e, ﬁrst strategy in Fig. 2.11) does not do this because the probability of selecting the correct ball is 1/9. This does not create a equiprobable situation in the process. The second strategy leaves an equally probable situation at every stage (see Fig. 2.12), so it extracts the most information out of each weighing as possible. Thus, the information content tells us how many bits of information have to be resolved using any strategy (i.e., log 2 (9) in this example). It also illuminates how to efﬁciently remove uncertainty; namely, by adjudicating equiprobable situations as many times as possible.

**2.7.2 Properties of Information Entropy**

Now that we have the ﬂavor of the concepts, consider the following properties of the information entropy,

H(X) ≥ 0

with equality if and only if P(x) = 1 for exactly one x. Intuitively, this means that when just one of the items in the ensemble is known absolutely (i.e., with P(x) = 1), the uncertainty collapses to zero. Also note that entropy is maximized when P is uniformly distributed across the elements of the ensemble. This is illustrated in Fig. 2.13 for the case of two outcomes. In other words, information entropy is maximized when the two conﬂicting alternatives are equally probable. This is the mathematical reason why using the scale in the last example to adjudicate equally probable situations was so useful for abbreviating the weighing process.

Most importantly, the concept of entropy extends jointly as follows,

1 = H(X, Y) ∑ P(x, y) log 2 P(x, y) x,y

If and only if X and Y are independent, entropy becomes additive,

H(X, Y) = H(X) + H(Y) 2.7 Information Entropy

99

Fig. 2.13 The information entropy is maximized when

p = 1/2

**2.7.3 Kullback–Leibler Divergence**

Notions of information entropy lead to notions of distance between probability distributions that will become important for machine learning methods. The KullbackLeibler divergence between two probability distributions P and Q that are deﬁned over the same set is deﬁned as,

P(x) D K L ∑ P(x)(P, Q) = log 2 Q(x) x

Note that D K L (P, Q) ≥ 0 with equality if and only if P = Q. Sometimes the Kullback–Leibler divergence is called the Kullback–Leibler distance, but it is not formally a distance metric because it is asymmetrical in P and Q. The KullbackLeibler divergence deﬁnes a relative entropy as the loss of information if P is modeled in terms of Q. There is an intuitive way to interpret the Kullback–Leibler divergence and understand its lack of symmetry. Suppose we have a set of messages to transmit, each with a corresponding probability { (x 1 , P(x 1 )), (x 2 , P(x 2 )), . . . , (x n , P(x n )) } . Based on what we know about information entropy, it makes sense to encode the 1 length of the message by log 2 p(x) bits. This parsimonious strategy means that more frequent messages are encoded with fewer bits. Thus, we can rewrite the entropy of the situation as before,

1 = H(X) ∑ P(x k ) log 2 P(x k ) k

Now, suppose we want to transmit the same set of messages, but with a different set of probability weights, { (x 1 , Q(x 1 )), (x 2 , Q(x 2 )), . . . , (x n , Q(x n )) } . In this situation, we can deﬁne the cross-entropy as

1 H q ∑ P(xk (X) = ) log 2 Q(x k ) k 100

2

Probability

Note that only the purported length of the encoded message has changed, not the probabilityofthatmessage.ThedifferencebetweenthesetwoistheKullback–Leibler divergence,

P(x) D K L ∑ P(x)(P, Q) = H q (X) − H(X) = log 2 Q(x) x

In this light, the Kullback–Leibler divergence is the average difference in the encoded lengths of the same set of messages under two different probability regimes. This should help explain the lack of symmetry of the Kullback–Leibler divergence — left to themselves, P and Q would provide the optimal-length encodings separately, but there can be no necessary symmetry in how each regime would rate the informational value of each message (Q(x i ) versus P(x i )). Given that each encoding is optimallength in its own regime means that it must therefore be at least sub-optimal in another, thus giving rise to the Kullback–Leibler divergence. In the case where the encoding length of all messages remains the same for the two regimes, then the Kullback–Leibler divergence is zero.2

**2.7.4 Cross-Entropy as Maximum Likelihood**

Reconsideringmaximumlikelihoodfromourstatisticschapterinmoregeneralterms, we have

n θ ML = arg max ∑ log p model (x i ; θ) θ i=1

where p model is the assumed underlying probability density function parameterized by θ for the x i data elements. Dividing the above summation by n does not change the derived optimal values, but it allows us to rewrite this using the empirical density function for x as the following,

θ ML = arg max E x∼ p data ˆ (log p model (x i ; θ)) θ

Note that we have the distinction between p data and pdata ˆ where the former is the unknown distribution of the data and the latter is the estimated distribution of the data we have on hand.

The cross-entropy can be written as the following,

D K L (P, Q) = E X∼P (log P(x)) − E X∼P (log Q(x))

2 The best, easy-to-understand presentation of this material is chapter four of Mackay’s text [7]. Another good reference is chapter four of [8]. 2.7 Information Entropy

101

where X ∼ P means the random variable X has distribution P. Thus, we have

θ ML = arg max D K L ( pdata ˆ , p model )

θ

That is, we can interpret maximum likelihood as the cross-entropy between the pmodel and the pdata ˆ distributions. The ﬁrst term has nothing to do with the estimated θ so maximizing this is the same as minimizing the following,

E x∼ p data ˆ (log p model (x i ; θ))

because information entropy is always non-negative. The important interpretation is that maximum likelihood is an attempt to choose θ model parameters that make the empirical distribution of the data match the model distribution.

**2.8 Moment Generating Functions**

Generating moments usually involves integrals that are extremely difﬁcult to compute. Moment generating functions make this much, much easier. The moment generating function is deﬁned as,

M(t) = E(exp(t X))

The ﬁrst moment is the mean, which we can easily compute from M(t) as,

dM(t) d d = E(exp(t X)) = E (exp(t X)) dt dt dt

= E(X exp(t X))

Now, we have to set t = 0 and we have the mean,

M (1) (0) = E(X)

continuing this derivative process again, we obtain the second moment as,

M (2) (t) = E(X 2 exp(t X))

M (2) (0) = E(X 2 )

With this in hand, we can easily compute the variance as,

V(X) = E(X 2 ) − E(X) 2 = M (2) (0) − M (1) (0)2 102

2

Probability

Example. Returning to our favorite binomial distribution, let’s compute some moments using Sympy.

>>> import sympy as S

>>> from sympy import stats

>>> p,t = S.symbols(’p t’,positive=True)

>>> x=stats.Binomial(’x’,10,p)

>>> mgf = stats.E(S.exp(t\*x))

Now, let’s compute the ﬁrst moment (aka, mean) using the usual integration method and using moment generating functions,

>>> print(S.simplify(stats.E(x))) 10\*p

>>> print(S.simplify(S.diff(mgf,t).subs(t,0))) 10\*p

Otherwise, we can compute this directly as follows,

>>> print(S.simplify(stats.moment(x,1))) # mean 10\*p

>>> print(S.simplify(stats.moment(x,2))) # 2nd moment 10\*p\*(9\*p + 1)

In general, the moment generating function for the binomial distribution is the following,

M X (t) = ( p ( e t − 1 ) + 1 )n

A key aspect of moment generating functions is that they are unique identiﬁers of probability distributions. By the Uniqueness theorem, given two random variables X and Y, if their respective moment generating functions are equal, then the corresponding probability distribution functions are equal.

Example. Let’s use the uniqueness theorem to consider the following problem. Suppose we know that the probability distribution of X given U = p is binomial with parameters n and p. For example, suppose X represents the number of heads in n coin ﬂips, given the probability of heads is p. We want to ﬁnd the unconditional distribution of X. Writing out the moment generating function as the following,

E(e t X | U = p) = (pe t + 1 − p)n

Because U is uniform over the unit interval, we can integrate this part out

1 E(e t X ) = (pe t + 1 − p) n dp ∫ 0

1 et(n + 1)−1 = n + 1 e t −1

1 = + e 2t + e 3t + ··· + ent ) (1 et + n + 1 2.8 Moment Generating Functions

103

Thus, the moment generating function of X corresponds to that of a random variable that is equally likely to be any of the values 0, 1, . . . , n. This is another way of saying that the distribution of X is discrete uniform over { 0, 1, . . . , n } . Concretely, suppose we have a box of coins whose individual probability of heads is unknown and that we dump the box on the ﬂoor, spilling all of the coins. If we then count the number of coins facing heads-up, that distribution is uniform.

Moment generating functions are useful for deriving distributions of sums of independent random variables. Suppose X 1 and X 2 are independent and Y = X 1 + X 2 . Then, the moment generating function of Y follows from the properties of the expectation,

M Y (t) = E(e tY ) = E(e t X 1 + t X 2 )

= E(e t X 1 e t X 2 ) = E(e t X 1 )E(e t X 2 )

= M X 1 (t)M X 2 (t)

Example. Suppose we have two normally distributed random variables, X 1 ∼ N( μ 1 , σ 1 ) and X 2 ∼ N( μ 2 , σ 2 ) with Y = X 1 + X 2 . We can save some tedium by exploring this in Sympy,

>>> S.var(’x:2’,real=True) (x0, x1)

>>> S.var(’mu:2’,real=True) (mu0, mu1)

>>> S.var(’sigma:2’,positive=True) (sigma0, sigma1)

>>> S.var(’t’,positive=True) t

>>> x0=stats.Normal(x0,mu0,sigma0)

>>> x1=stats.Normal(x1,mu1,sigma1)

**Programming Tip**

The S.var function deﬁnes the variable and injects it into the global namespace. This is sheer laziness. It is more expressive to deﬁne variables explicitly as in x = S.symbols(’x’). Also notice that we used the Greek names for the mu and sigma variables. This will come in handy later when we want to render the equations in the Jupyter notebook which understands how to type-A set these symbols in L T E X. The var(’x:2’) creates two symbols, x0 and x1. Using the colon this way makes it easy to generate array-like sequences of symbols.

In the next block we compute the moment generating functions

>>> mgf0=S.simplify(stats.E(S.exp(t\*x0)))

>>> mgf1=S.simplify(stats.E(S.exp(t\*x1)))

>>> mgfY=S.simplify(mgf0\*mgf1) 104

2

Probability

The moment generating functions an individual normally distributed random variable is the following,

eμ 0 t + σ 0 2 2 t 2

Note the coefﬁcients of t. To show that Y is normally distributed, we want to match the moment generating function of Y to this format. The following is the form of the moment generating function of Y,

t 2 2 M Y (t) = e 2 ( 2 μ 0 + 2 μ 1 + σ 0 t + σ 1 t )

We can extract the exponent using Sympy and collect on the t variable using the following code,

>>> S.collect(S.expand(S.log(mgfY)),t) t\*\*2\*(sigma0\*\*2/2 + sigma1\*\*2/2) + t\*(mu0 + mu1)

Thus, by the Uniqueness theorem, Y is normally distributed with μ Y = μ0

σ Y 2 = σ 0 2 + σ 1 2 .

**Programming Tip**

+ μ1

and

When using the Jupyter notebook, you can do S.init\_printing to get the mathematical typesetting to work in the browser. Otherwise, if you A want to keep the raw expression and to selectively render to L T E X, then you can from IPython.display import Math, and then use Math(S.latex(expr)) to see the typeset version of the expression.

**2.9 Monte Carlo Sampling Methods**

So far, we have studied analytical ways to transform random variables and how to augment these methods using Python. In spite of all this, we frequently must resort to purely numerical methods to solve real-world problems. Hopefully, now that we have seen the deeper theory, these numerical methods will feel more concrete. Suppose we want to generate samples of a given density, f (x), given we already can generate samples from a uniform distribution, U [ 0, 1 ] . How do we know a random sample v comes from the f (x) distribution? One approach is to look at how a histogram of samples of v approximates f (x). Speciﬁcally,

Hasta ahora, hemos estudiado formas analíticas de transformar variables aleatorias y cómo aumentar estos métodos usando Python. A pesar de todo esto, frecuentemente debemos recurrir a métodos puramente numéricos para resolver problemas del mundo real. Con suerte, ahora que hemos visto la teoría más profunda, estos métodos numéricos parecerán más concretos. Supongamos que queremos generar muestras de una densidad determinada, f (x), dado que ya podemos generar muestras a partir de una distribución uniforme, U [ 0, 1 ] . ¿Cómo sabemos que una muestra aleatoria v proviene de la distribución f (x)? Un enfoque consiste en observar cómo un histograma de muestras de v se aproxima a f (x). Específicamente,

P(v ∈ N Δ (x)) = f (x)Δx

(2.9.0.1) 2.9 Monte Carlo Sampling Methods

105

which says that the probability that a sample is in some N Δ neighborhood of x is approximately f (x)Δx. Figure 2.14 shows the target probability density function f (x) and a histogram that approximates it. The histogram is generated from samples v. The hatched rectangle in the center illustrates Eq. 2.9.0.1. The area of this rectangle is approximately f (x)Δx where x = 0, in this case. The width of the rectangle is N Δ (x) The quality of the approximation may be clear visually, but to know that v samples are characterized by f (x), we need the statement of Eq. 2.9.0.1, which says that the proportion of samples v that ﬁll the hatched rectangle is approximately equal to f (x)Δx.

Now that we know how to evaluate samples v that are characterized by the density f (x), let’s consider how to create these samples for both discrete and continuous random variables.

**2.9.1 Inverse CDF Method for Discrete Variables**

Suppose we want to generate samples from a fair six-sided die. Our workhouse uniform random variable is deﬁned continuously over the unit interval and the fair six-sided die is discrete. We must ﬁrst create a mapping between the continuous random variable u and the discrete outcomes of the die. This mapping is shown in Fig. 2.15 where the unit interval is broken up into segments, each of length 1/6. Each individual segment is assigned to one of the die outcomes. For example, if u ∈ [ 1/6, 2/6), then the outcome for the die is 2. Because the die is fair, all segments on the unit interval are the same length. Thus, our new random variable v is derived from u by this assignment.

Fig. 2.14 The histogram approximates the target probability density

Fig. 2.15 A uniform distribution random variable on the unit interval is assigned to the six outcomes of a fair die using these segments 106

2

Probability

For example, for v = 2, we have,

P(v = 2) = P(u ∈ [ 1/6, 2/6)) = 1/6

where,inthelanguageoftheEq.2.9.0.1, f (x) = 1(uniformdistribution),Δx = 1/6, and N Δ (2) = [ 1/6, 2/6). Naturally, this pattern holds for all the other die outcomes in { 1, 2, 3, . . . , 6 } . Let’s consider a quick simulation to make this concrete. The following code generates uniform random samples and stacks them in a Pandas dataframe.

>>> import pandas as pd

>>> import numpy as np

>>> from pandas import DataFrame

>>> u= np.random.rand(100)

>>> df = DataFrame(data=u,columns=[’u’])

The next block uses pd.cut to map the individual samples to the set { 1, 2, . . . , 6 } labeled v.

>>> labels = [1,2,3,4,5,6]

>>> df[’v’]=pd.cut(df.u,np.linspace(0,1,7),

...

include\_lowest=True,labels=labels)

This is what the dataframe contains. The v column contains the samples drawn from the fair die.

>>> df.head() u 0 0.356225 1 0.466557 2 0.776817 3 0.836790 4 0.037928

v 3 3 5 6 1

The following is a count of the number of samples in each group. There should be roughly the same number of samples in each group because the die is fair.

>>> df.groupby(’v’).count() u v 1 17 2 15 3 18 4 20 5 14 6 16

So far, so good. We now have a way to simulate a fair die from a uniformly distributed random variable.

To extend this to unfair die, we need only make some small adjustments to this code. For example, suppose that we want an unfair die so that P(1) = P(2) = P(3) = 1/12 and P(4) = P(5) = P(6) = 1/4. The only change we have to make is with pd.cut as follows, 2.9 Monte Carlo Sampling Methods

107

>>> df[’v’]=pd.cut(df.u,[0,1/12,2/12,3/12,2/4,3/4,1],

... include\_lowest=True,labels=labels) >>> df.groupby(’v’).count()/df.shape[0] u v 1 0.10 2 0.07 3 0.05 4 0.28 5 0.29 6 0.21

where now these are the individual probabilities of each digit. You can take more than 100 samples to get a clearer view of the individual probabilities but the mechanism for generating them is the same. The method is called the inverse CDF3 method because the CDF (namely,[0,1/12,2/12,3/12,2/4,3/4,1]) in the last example has been inverted (using the pd.cut method) to generate the samples. The inversion is easier to see for continuous variables, which we consider next.

**2.9.2 Inverse CDF Method for Continuous Variables**

The method above applies to continuous random variables, but now we have to squeeze the intervals down to individual points. In the example above, our inverse function was a piecewise function that operated on uniform random samples. In this case, the piecewise function collapses to a continuous inverse function. We want to generate random samples for a CDF that is invertible. As before, the criterion for generating an appropriate sample v is the following,

El método anterior se aplica a variables aleatorias continuas, pero ahora tenemos que reducir los intervalos a puntos individuales. En el ejemplo anterior, nuestra función inversa era una función por partes que operaba con muestras aleatorias uniformes. En este caso, la función por partes se colapsa en una función inversa continua. Queremos generar muestras aleatorias para un CDF que sea invertible. Como antes, el criterio para generar una muestra v apropiada es el siguiente,

x + Δx

P(F(x) < v < F(x + Δx)) = F(x + Δx) − F(x) = f (u)du ≈ f (x)Δx ∫ x which says that the probability that the sample v is contained in a Δx interval is approximately equal to f (x)Δx, at that point. Once again, the trick is to use a uniform random sample u and an invertible CDF F(x) to construct these samples. Note that for a uniform random variable u ∼ U [ 0, 1 ] , we have,

P(x < F −1 (u) < x + Δx) = P(F(x) < u < F(x + Δx))

= F(x + Δx) − F(x)

x + Δx = f (p)dp ≈ f (x)Δx ∫ x

This means that v = F −1 (u) is distributed according to f (x), which is what we want.

3 Cumulative density function. Namely, F(x) = P(X < x). 108

2

Probability

Let’s try this to generate samples from the exponential distribution,

f α (x) = αe −αx

which has the following CDF,

F(x) = 1 − e−αx

and corresponding inverse,

1 1 F −1 (u) = ln α (1 − u)

Now, all we have to do is generate some uniformly distributed random samples and then feed them into F −1 .

>>> from numpy import array, log

>>> import scipy.stats

>>> alpha = 1. # distribution parameter

>>> nsamp = 1000 # num of samples

>>> # define uniform random variable

>>> u=scipy.stats.uniform(0,1)

>>> # define inverse function

>>> Finv=lambda u: 1/alpha\*log(1/(1-u))

>>> # apply inverse function to samples

>>> v = array(list(map(Finv,u.rvs(nsamp))))

Now, we have the samples from the exponential distribution, but how do we know the method is correct with samples distributed accordingly? Fortunately, scipy.stats already has a exponential distribution, so we can check our work against the reference using a probability plot (i.e., also known as a quantile-quantile plot). The following code sets up the probability plot from scipy.stats.

fig,ax=subplots() scipy.stats.probplot(v,(1,),dist=’expon’,plot=ax)

Note that we have to supply an axes object (ax) for it to draw on. The result is Fig. 2.16. The more the samples line match the diagonal line, the more they match the reference distribution (i.e., exponential distribution in this case). You may also want to try dist=norm in the code above To see what happens when the normal distribution is the reference distribution.

**2.9.3 Rejection Method**

In some cases, inverting the CDF may be impossible. The rejection method can handle this situation. The idea is to pick two uniform random variables u 1 , u 2 ∼ U [ a, b ] so that 2.9 Monte Carlo Sampling Methods

109

Fig. 2.16 The samples created using the inverse cdf method match the exponential reference distribution

f (u1 ) f (u 1 ) Δx ≈ P ( u 1 ∧ u2 N Δ (x) < M b − ∈ a M )

where we take x = u 1 and f (x) < M. This is a two-step process. First, draw u1 uniformly from the interval [ a, b ] . Second, feed it into f (x) and if u 2 < f (u 1 )/M, then you have a valid sample for f (x). Thus, u 1 is the proposed sample from f that may or may not be rejected depending on u 2 . The only job of the M constant is to scale down the f (x) so that the u 2 variable can span the range. The efﬁciency of this method is the probability of accepting u 1 which comes from integrating out the above approximation,

f (x) 1 1 dx = f (x)dx = ∫ M(b − a) M(b − a) ∫ M(b − a)

This means that we don’t want an necessarily large M because that makes it more likely that samples will be discarded.

Let’s try this method for a density that does not have a continuous inverse.4

(x − 1)2 f (x) = exp − (x + 1)/12 ( 2x )

where x > 0. The following code implements the rejection plan.

4 Note that this example density does not exactly integrate out to one like a probability density function should, but the normalization constant for this is distracting for our purposes here. 110

2

Probability

Fig. 2.17 The rejection method generate samples in the histogram that nicely match the target distribution. Unfortunately, the efﬁciency is not so good

>>> import numpy as np

>>> x = np.linspace(0.001,15,100)

>>> f= lambda x: np.exp(-(x-1)\*\*2/2./x)\*(x+1)/12.

>>> fx = f(x)

>>> M=0.3

>>> u1 = np.random.rand(10000)\*15

>>> u2 = np.random.rand(10000)

>>> idx,= np.where(u2<=f(u1)/M)

>>> v = u1[idx]

# scale factor # uniform random samples scaled out # uniform random samples # rejection criterion

Figure 2.17 shows a histogram of the so-generated samples that nicely ﬁts the probability density function. The title in the ﬁgure shows the efﬁciency (the number of rejected samples), which is poor. It means that we threw away most of the proposed samples. Thus, even though there is nothing conceptually wrong with this result, the low efﬁciency must be ﬁxed, as a practical matter. Figure 2.18 shows where the proposed samples were rejected. Samples under the curve were retained (i.e., u 2 < f (u 1 ) ) but the vast majority of the samples are outside this umbrella. M

The rejection method uses u 1 to select along the domain of f (x) and the other u 2 uniform random variable decides whether to accept or not. One idea would be to choose u 1 so that x values are coincidentally those that are near the peak of f (x), instead of uniformly anywhere in the domain, especially near the tails, which are low probability anyway. Now, the trick is to ﬁnd a new density function g(x) to sample from that has a similiar concentration of probability density. One way it to familiarize oneself with the probability density functions that have adjustable parameters and fast random sample generators already. There are lots of places to look and, chances are, there is likely already such a generator for your problem. Otherwise, the family of β densities is a good place to start.

Tobeexplicit,whatwewantisu 1 ∼ g(x)sothat,returningtoourearlierargument,

f (u1 ) ) f (u1 P ∈ ≈ ( u 1 ∧ u2 N Δ (x) < M ) g(x)Δx M 2.9 Monte Carlo Sampling Methods

111

Fig. 2.18 The proposed samples under the curve were accepted and the others were not. This shows the majority of samples were rejected

but this is not what we need here. The problem is with the second part of the logical ∧ conjunction. We need to put something there that will give us something proportional to f (x). Let us deﬁne the following,

f (x) h(x) = g(x)

(2.9.3.1)

with corresponding maximum on the domain as h max and then go back and construct the second part of the clause as

h(u1 ) ) h(u1 P u 1 ∈ N Δ (x) ∧ u 2 < ≈ g(x)Δx = f (x)/hmax ( h max ) hmax

Recall that satisfying this criterion means that u 1 = x. As before, we can estimate the probability of acceptance of the u 1 as 1/h max .

Now, how to construct the g(x) function in the denominator of Eq. 2.9.3.1? Here’s where familiarity with some standard probability densities pays off. For this case, we choose the χ 2 distribution. The following plots the g(x) and f (x) (left plot) and the corresponding h(x) = f (x)/g(x) (right plot). Note that g(x) and f (x) have peaks that almost coincide, which is what we are looking for (Fig.2.19).

>>> ch=scipy.stats.chi2(4) # chi-squared

>>> h = lambda x: f(x)/ch.pdf(x) # h-function

Now, let’s generate some samples from this χ 2 distribution with the rejection method.

>>> hmax=h(x).max()

>>> u1 = ch.rvs(5000) # samples from chi-square distribution

>>> u2 = np.random.rand(5000)# uniform random samples

>>> idx = (u2 <= h(u1)/hmax) # rejection criterion # keep these only

>>> v = u1[idx] 112

2

Probability

Fig. 2.19 The plot on the right shows h(x) = f (x)/g(x) and the one on the left shows f (x) and g(x) separately

Fig. 2.20 Using the updated method, the histogram matches the target probability density function with high efﬁciency

Fig. 2.21 Fewer proposed points were rejected in this case, which means better efﬁciency

Using the χ 2 distribution with the rejection method results in throwing away less than 10% of the generated samples compared with our prior example where we threw out at least 80%. This is dramatically more efﬁcient! Figure 2.20 shows that the histogram and the probability density function match. For completeness, Fig. 2.21 shows the samples with the corresponding threshold h(x)/h max that was used to select them. 2.10 Sampling Importance Resampling

113

2.10 Sampling Importance Resampling

An alternative to the Rejection Method that does not involve rejecting samples or coming up with M bounds or bounding functions is the Sampling Importance Resampling (SIR) method. Choose a tractable g probability density function and draw a n samples from it, { x i } i=1 n . Our objective is to derive samples f . Next, compute the following,

wi q i = ∑ wi

where

f (xi ) w i = g(x i )

The q i deﬁne a probability mass function whose samples approximate samples from f . To see this, consider,

n P(X ≤ a) = ∑ q i I (−∞,a ] (x i ) i=1 wi i=1 n I (−∞,a (x i ) = ∑i=1 n ∑ wi ] 1 f (x i ) g(x i ) I (−∞,a ] (x i ) n ∑i=1 n = f (x ) 1 i n ∑ i=1 n g(x i )

Because the samples are generated from the g probability distribution, the numerator is approximately,

f (x) a E g f (x)dx = ( g(x) ) ∫ −∞

which gives

a P(X ≤ a) = f (x)dx ∫ −∞

which shows that the samples generated this way are f -distributed. Note more samples have to be generated from this probability mass function the further away g is from the desired function f . Further, because there is no rejection step, we no longer have the issue of efﬁciency.

For example, let us choose a beta distribution for g, as in the following code,

>>> g = scipy.stats.beta(2,3)

This distribution does not bear a strong resemblance to our desired f function from last section. as shown in the Fig. 2.22. Note that we scaled the domain of the beta distribution to get it close to the support of f . 114

2

Probability

Fig. 2.22 Histogram of samples generated using SIR comparted to target probability density function

Fig. 2.23 Histogram and probability density function using SIR

In the next block, we sample from the g distribution and compute the weights as described above. The ﬁnal step is to sample from this new probability mass function. The resulting normalized histogram is shown compared to the target f probability density function in Fig. 2.23.

>>> xi = g.rvs(500)

>>> w = np.array([f(i\*15)/g.pdf(i) for i in xi])

>>> fsamples=np.random.choice(xi\*15,5000,p = w/w.sum())

In this section, we investigated how to generate random samples from a given distribution, beit discrete or continuous. For the continuous case, the key issue was whether or not the cumulative density function had a continuous inverse. If not, we had to turn to the rejection method, and ﬁnd an appropriate related density that we could easily sample from to use as part of a rejection threshold. Finding such a 2.10 Sampling Importance Resampling

115

function is an art, but many families of probability densities have been studied over the years that already have fast random number generators.

The rejection method has many complicated extensions that involve careful partitioning of the domains and lots of special methods for corner cases. Nonetheless, all of these advanced techniques are still variations on the same fundamental theme we illustrated here [9, 10].

**2.11 Useful Inequalities**

In practice, few quantities can be analytically calculated. Some knowledge of bounding inequalities helps ﬁnd the ballpark for potential solutions. This sections discusses three key inequalities that are important for probability, statistics, and machine learning.

En la práctica, pocas cantidades pueden calcularse analíticamente. Cierto conocimiento de las desigualdades acotadas ayuda a encontrar el terreno de juego para posibles soluciones. Esta sección analiza tres desigualdades clave que son importantes para la probabilidad, la estadística y el aprendizaje automático.

**2.11.1 Markov’s Inequality**

Let X be a non-negative random variable and suppose that E(X) < ∞. Then, for any t > 0,

E(X)

P(X > t) ≤ t

This is a foundational inequality that is used as a stepping stone to other inequalities. It is easy to prove. Because X > 0, we have the following,

∞ t ∞ E(X) = x f x (x)dx = x f x (x)dx + x f x (x)dx ∫ 0 ∫ 0 ∫ t

︶

︵︵

omit this

︶

∞ ∞ ≥ x f x (x)dx ≥ t f x (x)dx = tP(X > t) ∫ t ∫ t

t The step that establishes the inequality is the part where the ∫ 0 x f x (x)dx is omitted. For a particular f x (x) that may be concentrated around the [ 0, t ] interval, this could be a lot to throw out. For that reason, the Markov Inequality is considered a loose inequality, meaning that there is a substantial gap between both sides of the inequality. For example, as shown in Fig. 2.24, the χ 2 distribution has a lot of its mass on the left, which would be omitted in the Markov Inequality. Figure 2.25 shows the two curves established by the Markov Inequality. The gray shaded region is the gap between the two terms and indicates that looseness of the bound (fatter shaded region) for this case. 116

2

Probability

Fig. 2.24 The χ 1 2 density has much of its weight on the left, which is excluded in the establishment of the Markov Inequality

Fig. 2.25 The shaded area shows the region between the curves on either side of the Markov Inequality

**2.11.2 Chebyshev’s Inequality**

Chebyshev’s Inequality drops out directly from the Markov Inequality. Let μ = E(X) and σ 2 = V(X). Then, we have

2 σ P( | X − μ| ≥ t) ≤ t2

Note that if we normalize so that Z = (X − μ )/σ, we have P( | Z | ≥ k) ≤ 1/k 2 . In particular, P( | Z | ≥ 2) ≤ 1/4. We can illustrate this inequality using Sympy statistics module, 2.11 Useful Inequalities

117

>>> import sympy

>>> import sympy.stats as ss

>>> t=sympy.symbols(’t’,real=True)

>>> x=ss.ChiSquared(’x’,1)

To get the left side of the Chebyshev inequality, we have to write this out as the following conditional probability,

>>> r = ss.P((x-1) > t,x>1)+ss.P(-(x-1) > t,x<1)

We could take the above expression, which is a function of t and attempt to compute the integral, but that would take a very long time (the expression is very long and complicated, which is why we did not print it out above). In this situation, it’s better to use the built-in cumulative density function as in the following (after some rearrangement of the terms),

>>> w=(1-ss.cdf(x)(t+1))+ss.cdf(x)(1-t)

To plot this, we can evaluated at a variety of t values by using the .subs substitution method, but it is more convenient to use the lambdify method to convert the expression to a function.

>>> fw=sympy.lambdify(t,w)

Then, we can evaluate this function using something like

>>> [fw(i) for i in [0,1,2,3,4,5]] [1.0,0.157299207050285,(0.08326451666355039+0j),(0.045500263 89635842+0j),(0.0253473186774682+0j),(0.014305878435429631+0 j)]

to produce the following Fig. 2.26.

Fig. 2.26 The shaded area shows the region between the curves on either side of the Chebyshev Inequality 118

2

Probability

Programming Tip

Note that we cannot use vectorized inputs for the lambdify function because it contains embedded functions that are only available in Sympy. Otherwise, we could have used lambdify(t,fw,numpy) to specify the corresponding functions in Numpy to use for the expression.

**2.11.3 Hoeffding’s Inequality**

Hoeffding’s Inequality is similar, but less loose, than Markov’s Inequality. Let X 1 , . . . , X n be iid observations such that E(X i ) = μ and a ≤ X i ≤ b. Then, for any > 0, we have

P( | X n − μ| ≥ ) ≤ 2 exp(−2n 2 /(b − a) 2 )

1 where X n = n ∑ i n X i . Note that we further assume that the individual random variables are bounded.

Corollary. If X 1 , . . . , X n are independent with P(a ≤ Xi E(X i ) = μ . Then, we have

≤

b) = 1 and all with

c 2 | X n − μ| ≤ log √ 2n δ

where c = (b−a) 2 . We will see this inequality again in the machine learning chapter. Figure 2.27 shows the Markov and Hoeffding bounds for the case of ten identically and uniformly distributed random variables, X i ∼ U [ 0, 1 ] . The solid line shows P( | X n − 1/2 | > ). Note that the Hoeffding Inequality is tighter than the Markov Inequality and that both of them merge when gets big enough.

ProofofHoeffding’sInequality.WewillneedthefollowinglemmatoproveHoeffding’s inequality.

Lemma. Let X be a random variable with E(X) = 0 and a ≤ X ≤ b. Then, for any s > 0, we have the following,

E(e sX ) ≤ es 2 (b−a) 2 /8

(2.11.3.1)

Because X is contained in the closed interval [ a, b ] , we can write it as a convex combination of the endpoints of the interval.

X =α1 a + α2 b 2.11 Useful Inequalities

119

Fig. 2.27 This shows the Markov and Hoeffding bounds for the case of ten identically and uniformly distributed random variables

where α1

+

α 2 = 1. Solving for the α i terms, we have

x −a α1 = b−a b x α2 = b−a

From Jensen’s inequality, for a convex functions f , we know that

f ∑ α i x i ∑ α i f (x i ) ≤ ( )

Given the convexity of e X , we therefore have,

e sX ≤ α1 e sa + α2 esb

With E(X) = 0, we can write the expectation of both sides

E(e sX ) ≤ E(α 1 )e sa + E(α 2 )esb

with E(α 1 ) =

b

b−a

−a and E(α 2 ) = b−a . Thus, we have

b a E(e sX ) ≤ esa esb b−a b−a

Using p :=

−a b−a , we can rewrite the following,

b a esa − = (1 − p)e sa + pe sb =: eφ (u) esb b−a b−a 120

2

Probability

where

φ (u) = −pu + log(1 − p + pe u )

and u = s(b − a). Note that φ (0) = φ ′ (0) = 0. Also, φ ′′ (0) = p(1 − p) ≤ 1/4. 2 Thus, the Taylor expansion of φ (u) ≈ u 2 2 φ ′′ (t) ≤ u 8 for t ∈ [ 0, u ] . To prove Hoeffding’s inequality, we start with Markov’s inequality,

E(X) P(X ≥ ) ≤

Then, given s > 0, we have the following,

E(esX ) P(X ≥ ) = P(e sX ≥ e s ) ≤ es

We can write the one-sided Hoeffding inequality as the following,

n s P(X n − μ ≥ ) ≤ e −s E(exp( ∑ (X i − E(X i )))) n i=1

n s = e −s ∏ E(e n (X i −E(X i )) ) i=1 n 2 s 2 (b−a) /8 e −s ∏ e n 2 ≤ i=1 2 = e −s e s n 2 (b−a) /8

Now, we want to pick s > 0 to minimize this upper bound. Then, with s =

4n

P(X n − μ ≥ ) ≤ e− (b−a) 2n 2 2

2

(b−a)

The other side of the inequality follows similarly to obtain Hoeffding’s inequality.

**References**

1. F. Jones, Lebesgue Integration on Euclidean Space. Jones and Bartlett Books in Mathematics. (Jones and Bartlett, London, 2001)
2. G. Strang, Linear Algebra and Its Applications (Thomson, Brooks/Cole, 2006)
3. N. Edward, Radically Elementary Probability Theory. Annals of Mathematics Studies (Princeton University Press, Princeton, 1987)
4. T. Mikosch, Elementary Stochastic Calculus with Finance in View. Advanced Series on Statistical Science & Applied Probability (World Scientiﬁc, Singapore, 1998)
5. H. Kobayashi, B.L. Mark, W. Turin, Probability, Random Processes, and Statistical Analysis: Applications to Communications, Signal Processing, Queueing Theory and Mathematical Finance. EngineeringPro Collection (Cambridge University Press, Cambridge, 2011)
6. Z.Brzezniak,T.Zastawniak,BasicStochasticProcesses:ACourseThroughExercises.Springer Undergraduate Mathematics Series (Springer, London, 1999)
7. D.J.C. MacKay, Information Theory, Inference and Learning Algorithms (Cambridge University Press, Cambridge, 2003)
8. T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics (Springer, New York, 2013)
9. W.L. Dunn, J.K. Shultis, Exploring Monte Carlo Methods (Elsevier Science, Boston, 2011)
10. N.L. Johnson, S. Kotz, N. Balakrishnan, Continuous Univariate Distributions. Wiley Series in Probability and Mathematical Statistics: Applied Probability and Statistics, vol. 2. (Wiley, New York, 1995)