Discrete probability distributions

Rachid Hamadi, CSE, UNSW

COMP9021 Principles of Programming, Term 3, 2019

[1]:
```
from collections.abc import Set
from fractions import Fraction
```

A *finite probability distribution* is a mapping μ from a finite set S to the set of nonnegative real numbers with $\Sigma_{e \in S} \mu(e) = 1$. An *event* is a subset of S, and its probability is defined as $\Sigma_{e \in E} \mu(e)$.

Let us restrict ourselves to finite probability distributions that assign rational numbers to all members of its domain. We can then accurately compute the probablity of an event thanks to the `Fraction` class of the `fractions` module. This class allows for the creation of `fraction` objects from a variety of arguments, which the usual arithmetic operators can be applied to:

[2]:
```
Fraction(2, 6), Fraction('6/10'), Fraction('1.234')
Fraction(Fraction(3, 2), Fraction(5, 4))
Fraction(1, 3) + Fraction(1, 4)
Fraction(4, 4) - Fraction(2, 5)
Fraction(1, 2) * Fraction(3, 4)
Fraction(-3, 7) / Fraction(7, 3)
Fraction(1, 2) ** 4
```

[2]:
```
(Fraction(1, 3), Fraction(3, 5), Fraction(617, 500))
```

[2]:
```
Fraction(6, 5)
```

[2]:
```
Fraction(7, 12)
```

[2]:
```
Fraction(3, 5)
```

[2]:
```
Fraction(3, 8)
```

[2]:
```
Fraction(-9, 49)
```

[2]:
```
Fraction(1, 16)
```

Let us cater for the creation of probability distributions either from a nonempty finite set S, in which case the distribution will be *uniform*, assigning 1 over the number of elements in S to each member of S, or from a dictionary whose keys are the members of S and whose values are the associated probabilities, required to all be `Fraction` objects that add up to 1. If those conditions are not met, it is natural to raise an exception. We create a new type of Exception specifically designed for that purpose:

[3]:
```
class FiniteProbabilityDistributionError(Exception):
    pass
```

[4]:
```
raise FiniteProbabilityDistributionError('Not happy with this!')
```

```
↪----------------------------------------------------------------------------

      FiniteProbabilityDistributionError       Traceback (most recent call␣
↪last)

      <ipython-input-4-6466140b56ff> in <module>
    ----> 1 raise FiniteProbabilityDistributionError('Not happy with this!')


      FiniteProbabilityDistributionError: Not happy with this!
```

One way to gracefully terminate the execution of a program when an exception is raised, simply printing out the associated error message, is to use the `except ... as ...` syntax:

[5]:
```
try:
    raise FiniteProbabilityDistributionError('Not happy with this!')
except FiniteProbabilityDistributionError as error:
    print(error)
```

```
Not happy with this!
```

To test the type of an object, we can use the `isinstance()` function; it takes the object as first argument and the type as second argument. If one intends to work with sets, then it is natural to provide `set` as first argument to `isinstance()`, but a more robust solution is to instead test against the `Set` class from the `collections.abc` module. `Set` is an abstract class that demands that 3 special methods to be implemented, based on the idea that a set is any object with the following properties:

- one can ask how many elements it contains;
- one can ask of something whether it belongs to it;
- one can request that its elements be generated, in an arbitrary order.

[6]:
```
class TwoElementSet(Set):
    def __init__(self, a, b):
        self.a = a
        self.b = b

    # Three abstract methods have to be implemented:
    # * __contains()__
    # * __iter()__
    # * __len()__

S = TwoElementSet(3, 5)
```

```
↪----------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call last)

<ipython-input-6-67c643390a0c> in <module>
  9     # * __len()__
 10
---> 11 S = TwoElementSet(3, 5)


TypeError: Can't instantiate abstract class TwoElementSet with abstract methods __contains__, __iter__, __len__
```

[7]:
```
class TwoElementSet(Set):
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __contains__(self, x):
        return x == self.a or x == self.b

    def __iter__(self):
        yield self.a
        yield self.b

    def __len__(self):
        return 2

S = TwoElementSet(3, 5)

3 in S, 4 in S
list(S)
len(S)
isinstance(S, set), isinstance(S, Set)
isinstance({3, 5}, set), isinstance({3, 5}, Set)
```

[7]:
```
(True, False)
```

[7]:
```
[3, 5]
```

[7]:
```
2
```

[7]:
```
(False, True)
```

[7]:
```
(True, True)
```

We now have everything we need to create objects meant to represent a finite probability distribution, with a dictionary as an object attribute to record the function:

[8]:
```
class FiniteProbabilityDistribution:
    def __init__(self, μ):
        if isinstance(μ, Set):
            if not len(μ):
                raise FiniteProbabilityDistributionError(
                                    'The sample space should not be empty'
                                                        )
            self.μ = {outcome: Fraction(1, len(μ)) for outcome in μ}
        else:
            if any(not isinstance(fraction, Fraction)
                        for fraction in μ.values()
                  ):
                raise FiniteProbabilityDistributionError(
                                    'Probabilities should be Fractions'
                                                        )
            if sum(μ.values()) != 1:
                raise FiniteProbabilityDistributionError(
                                    'Probabilities should add up to 1'
                                                        )
            self.μ = μ

    def __repr__(self):
            return f'FiniteProbabilityDistribution({self.μ})'

    def __str__(self):
        return '\n'.join(outcome + ' : ' + str(self.μ[outcome])
                                for outcome in sorted(self.μ)
                         )

try:
    pd = FiniteProbabilityDistribution(set())
except FiniteProbabilityDistributionError as error:
    print(error)
try:
    pd = FiniteProbabilityDistribution({'a': 0.5, 'b': 0.5})
except FiniteProbabilityDistributionError as error:
    print(error)
try:
    pd = FiniteProbabilityDistribution({'a': Fraction(1, 2),
                                        'b': Fraction(1, 3)
                                       }
                                      )
except FiniteProbabilityDistributionError as error:
    print(error)

pd_1 = FiniteProbabilityDistribution({'a': Fraction(3, 8), 'b': Fraction(1, 8),
                                      'c': Fraction(1, 3), 'd': Fraction(1, 6)
```

```
                                        }
                                       )
pd_1
print(pd_1)

pd_2 = FiniteProbabilityDistribution(set('abcdefghijkl'))
pd_2
print(pd_2)
```

```
The sample space should not be empty
Probabilities should be Fractions
Probabilities should add up to 1
```

[8]:
```
FiniteProbabilityDistribution({'a': Fraction(3, 8), 'b': Fraction(1, 8), 'c':
Fraction(1, 3), 'd': Fraction(1, 6)})
```

```
a : 3/8
b : 1/8
c : 1/3
d : 1/6
```

[8]:
```
FiniteProbabilityDistribution({'c': Fraction(1, 12), 'h': Fraction(1, 12), 'k':
Fraction(1, 12), 'e': Fraction(1, 12), 'f': Fraction(1, 12), 'l': Fraction(1,
12), 'b': Fraction(1, 12), 'd': Fraction(1, 12), 'g': Fraction(1, 12), 'i':
Fraction(1, 12), 'a': Fraction(1, 12), 'j': Fraction(1, 12)})
```

```
a : 1/12
b : 1/12
c : 1/12
d : 1/12
e : 1/12
f : 1/12
g : 1/12
h : 1/12
i : 1/12
j : 1/12
k : 1/12
l : 1/12
```

We create a class for events of a finite probability distribution, making sure it consists of members of the domain of the latter, with the empty event as a default, and computing the probability of an event on creation:

[9]:
```
class Event:
    def __init__(self, distribution, outcomes=set()):
        if any(outcome not in distribution.μ for outcome in outcomes):
            raise FiniteProbabilityDistributionError(
                                'Event not for probability distribution'
```

```
                                                    )
        self.distribution = distribution
        self.outcomes = outcomes
        self.probability = sum(self.distribution.μ[outcome]
                               for outcome in self.outcomes
                           )

    def __repr__(self):
            return f'Event({repr(self.distribution)}, {self.outcomes})'

    def __str__(self):
        return ''.join(('{', ', '.join(sorted(self.outcomes)), '} : ',
                        str(self.probability)
                       )
                      )

try:
    E = Event(pd_1, set('aCd'))
except FiniteProbabilityDistributionError as error:
    print(error)

E_1 = Event(pd_1, {'a', 'b'})
E_1
print(E_1)

E_2 = Event(pd_1, {'b', 'c', 'd'})
E_2
print(E_2)

E_3 = Event(pd_1, {'b', 'd'})
E_3
print(E_3)

F_1 = Event(pd_2, set('abcdijkl'))
F_1
print(F_1)

F_2 = Event(pd_2, set('abcdef'))
F_2
print(F_2)

F_3 = Event(pd_2, set('bce'))
F_3
print(F_3)
```

```
Event not for probability distribution
```

[9]:
```
Event(FiniteProbabilityDistribution({'a': Fraction(3, 8), 'b': Fraction(1, 8),
'c': Fraction(1, 3), 'd': Fraction(1, 6)}), {'a', 'b'})
```

```
{a, b} : 1/2
```

[9]:
```
Event(FiniteProbabilityDistribution({'a': Fraction(3, 8), 'b': Fraction(1, 8),
'c': Fraction(1, 3), 'd': Fraction(1, 6)}), {'c', 'd', 'b'})
```

```
{b, c, d} : 5/8
```

[9]:
```
Event(FiniteProbabilityDistribution({'a': Fraction(3, 8), 'b': Fraction(1, 8),
'c': Fraction(1, 3), 'd': Fraction(1, 6)}), {'d', 'b'})
```

```
{b, d} : 7/24
```

[9]:
```
Event(FiniteProbabilityDistribution({'c': Fraction(1, 12), 'h': Fraction(1, 12),
'k': Fraction(1, 12), 'e': Fraction(1, 12), 'f': Fraction(1, 12), 'l':
Fraction(1, 12), 'b': Fraction(1, 12), 'd': Fraction(1, 12), 'g': Fraction(1,
12), 'i': Fraction(1, 12), 'a': Fraction(1, 12), 'j': Fraction(1, 12)}), {'c',
'k', 'b', 'd', 'i', 'l', 'a', 'j'})
```

```
{a, b, c, d, i, j, k, l} : 2/3
```

[9]:
```
Event(FiniteProbabilityDistribution({'c': Fraction(1, 12), 'h': Fraction(1, 12),
'k': Fraction(1, 12), 'e': Fraction(1, 12), 'f': Fraction(1, 12), 'l':
Fraction(1, 12), 'b': Fraction(1, 12), 'd': Fraction(1, 12), 'g': Fraction(1,
12), 'i': Fraction(1, 12), 'a': Fraction(1, 12), 'j': Fraction(1, 12)}), {'c',
'e', 'f', 'b', 'd', 'a'})
```

```
{a, b, c, d, e, f} : 1/2
```

[9]:
```
Event(FiniteProbabilityDistribution({'c': Fraction(1, 12), 'h': Fraction(1, 12),
'k': Fraction(1, 12), 'e': Fraction(1, 12), 'f': Fraction(1, 12), 'l':
Fraction(1, 12), 'b': Fraction(1, 12), 'd': Fraction(1, 12), 'g': Fraction(1,
12), 'i': Fraction(1, 12), 'a': Fraction(1, 12), 'j': Fraction(1, 12)}), {'c',
'e', 'b'})
```

```
{b, c, e} : 1/4
```

From a probability distribution μ and an event E with a nonzero probability, on can define the conditionalisation of μ on E, another probability distribution. To this aim, we add a `conditionalised_by()` method to the `FiniteProbabilityDistribution` class:

[10]:
```
class FiniteProbabilityDistribution(FiniteProbabilityDistribution):
    def conditionalised_by(self, event):
        if event.distribution.μ is not self.μ:
```

```
            raise FiniteProbabilityDistributionError(
                        'Cannot conditionalise distribution on event '
                        'for another probability distribution'
                                                    )
        if not event.probability:
            raise FiniteProbabilityDistributionError(
                                    'Cannot conditionalise on event with '
                                    'probability mass of 0'
                                                    )
        return FiniteProbabilityDistribution(
                    {outcome: self.μ[outcome] / event.probability
                        for outcome in event.outcomes
                    }
                                            )

pd_1 = FiniteProbabilityDistribution({'a': Fraction(3, 8), 'b': Fraction(1, 8),
                                      'c': Fraction(1, 3), 'd': Fraction(1, 6)
                                     }
                                    )
E_1 = Event(pd_1, {'a', 'b'})

pd_2 = FiniteProbabilityDistribution(set('abcdefghijkl'))
F_1 = Event(pd_2, set('abcdijkl'))

try:
    pd = pd_1.conditionalised_by(F_1)
except FiniteProbabilityDistributionError as error:
    print(error)

print()

print(pd_1.conditionalised_by(E_1))

print()

print(pd_2.conditionalised_by(F_1))
```

```
Cannot conditionalise distribution on event for another probability distribution

a : 3/4
b : 1/4

a : 1/8
b : 1/8
c : 1/8
d : 1/8
i : 1/8
```

```
j : 1/8
k : 1/8
l : 1/8
```

A probability distribution and events A and B determine the events $\overline{A}$, $A \cap B$, $A \cup B$ and $A|B$ and their associated probabilities. Python has a number of operators, in particular, $\sim$, &, $+$ and $|$, which can have as operands objects created from a class C provided that C implements the special methods `__invert()__`, `__and()__`, `__add()__` and `__or()__`, respectively. We consider that $\sim A$, $A\&B$ and $A + B$ are reasonable notational alternatives to $\overline{A}$, $A \cap B$ and $A \cup B$ and implement all four special methods. For $A \cap B$, $A \cup B$ and $A|B$, we first check that A and B are events for the same probability distribution μ. The implementation immediately follows from the definition of all four events. Note that $A|B$ is an event not for μ, but for μ given B:

[11]:
```
class Event(Event):
    def __invert__(self):
        return Event(self.distribution,
                     set(self.distribution.μ) - self.outcomes
                    )

    def __and__(self, other):
        if self.distribution.μ is not other.distribution.μ:
            raise FiniteProbabilityDistributionError(
                              'Cannot take intersection of events for '
                              'distinct probability distributions'
                                                     )
        return Event(self.distribution, self.outcomes & other.outcomes)

    def __add__(self, other):
        if self.distribution.μ is not other.distribution.μ:
            raise FiniteProbabilityDistributionError(
                             'Cannot take union of events for distinct '
                             'probability distributions'
                                                     )
        return Event(self.distribution, self.outcomes | other.outcomes)

    def __or__(self, other):
        if self.distribution.μ is not other.distribution.μ:
            raise FiniteProbabilityDistributionError(
                             'Cannot conditionalise event on event for '
                             'another probability distribution'
                                                     )
        return Event(self.distribution.conditionalised_by(other),
                     self.outcomes & other.outcomes
                    )

E_1 = Event(pd_1, {'a', 'b'})
E_2 = Event(pd_1, {'b', 'c', 'd'})
E_3 = Event(pd_1, {'b', 'd'})
```

```
F_1 = Event(pd_2, set('abcdijkl'))
F_2 = Event(pd_2, set('abcdef'))
F_3 = Event(pd_2, set('bce'))

try:
    print(E_1 & F_1)
except FiniteProbabilityDistributionError as error:
    print(error)
try:
    print(E_1 + F_1)
except FiniteProbabilityDistributionError as error:
    print(error)
try:
    print(E_1 | F_1)
except FiniteProbabilityDistributionError as error:
    print(error)
try:
    print(E_1 | Event(pd_1))
except FiniteProbabilityDistributionError as error:
    print(error)

print()

print(~E_1)
print(E_1 & E_2)
print(E_1 + E_2)
print(E_3 | E_2)
print(E_1 | E_2)
print(E_1 | ~E_1)

print()

print(~F_1)
print(F_1 & F_2)
print(F_1 + F_2)
print(F_3 | F_2)
print(F_1 | F_2)
```

```
Cannot take intersection of events for distinct probability distributions
Cannot take union of events for distinct probability distributions
Cannot conditionalise event on event for another probability distribution
Cannot conditionalise on event with probability mass of 0

{c, d} : 1/2
{b} : 1/8
{a, b, c, d} : 1
{b, d} : 7/15
```

```
{b} : 1/5
{} : 0

{e, f, g, h} : 1/3
{a, b, c, d} : 1/3
{a, b, c, d, e, f, i, j, k, l} : 5/6
{b, c, e} : 1/2
{a, b, c, d} : 2/3
```

The following function checks whether two events A and B are for the same probability distribution μ, and in case they are, determines whether they are independent, that is, whether $\mu(A) \times \mu(B) = \mu(A \cap B)$:

[12]:
```
def are_independent(event_1, event_2):
    if event_1.distribution.μ is not event_2.distribution.μ:
        raise FiniteProbabilityDistributionError(
                    'Both events not for same probability distribution'
                                                 )
    return event_1.probability * event_2.probability ==\
                                                (event_1 & event_2).probability

try:
    are_independent(E_1, F_1)
except FiniteProbabilityDistributionError as error:
    print(error)

are_independent(E_1, E_2)
are_independent(F_1, F_2)
```

```
Both events not for same probability distribution
```

[12]:
```
False
```

[12]:
```
True
```