1. Discrete Distribution 101

Overview

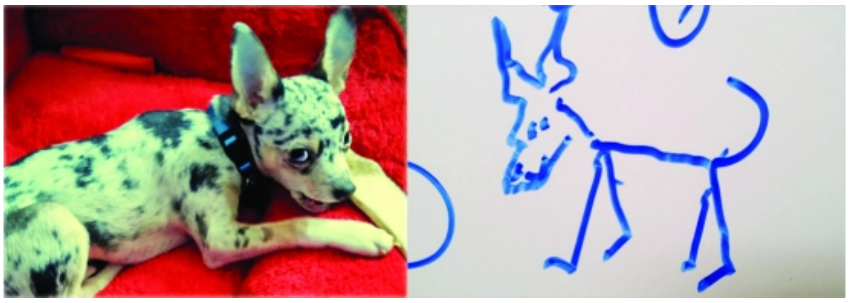
In this lecture, we discuss the basic concepts of random variables and distribution. This lecture also serves as an introduction to `NumPy` array.

Learning objective:

- A review of random variable and distribution.
- Introduction to the `NumPy`.
- Studying a discrete distribution using Python.

Random variable and distribution

In this course, we focus on simulating the outcomes of real-world systems. As the real world can be very complicated, we usually need to construct an abstract representation before we can even move onto the simulation stage.

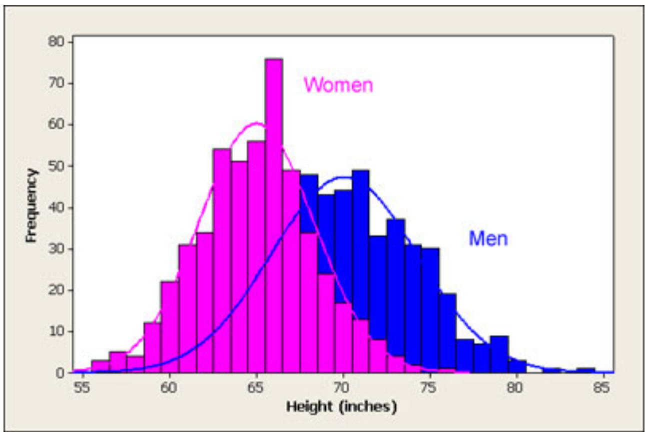


credit: gusdoggydogg

Let's assume that we would like to sample the height of 10 people, we need to think about the range of possible height and how likely each outcome is going to happen. We can then move on to our sampling process to draw the actual samples. In this example, the height of a person is a random variable, while the combination of the possible outcomes and how likely the outcomes are going to happen is called a distribution. More formally, we define random variable and distribution in the following way.

- A **random variable** is a variable whose possible values are outcomes of a random phenomenon. A random variable can be either discrete or continuous depending on whether the number of possible outcomes is countable or not.
- A **distribution** gives the possible outcomes of a random variable and how likely different outcomes could appear.

Constructing a distribution is both science and art and could require your domain knowledge and empirical data. For example, for the sampling of height, we might first plot the empirical data to give some rough idea of how the distribution looks.



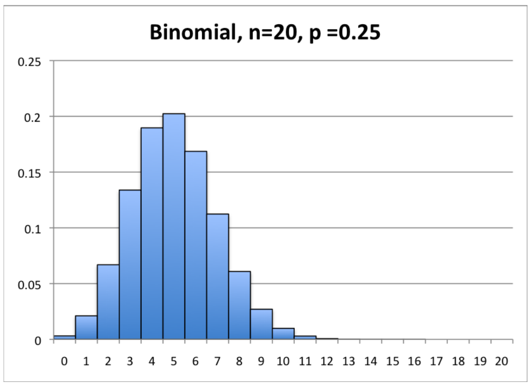
Discussion: What distribution do you think the height follows? Be as specific as possible. How do you think you should sample the height if you already know how to sample from the distribution?

As more and more people agree on the results, this will become a domain knowledge. For example, when people discuss the number of exposures within a unit time, Poisson distribution becomes one of the default options because it has been so widely applied.

Discrete distribution

Discrete Distribution is a distribution with a countable number of outcomes.

To define a discrete distribution, we need to give the values of the outcomes and their associated probabilities, which also called probability mass function. The probability of each outcome should be between 0 and 1. The probabilities of all possible outcomes should sum up to one.



The probabilities might be presented using a table, or using math expressions.

Example

In a specific week, the number of days of overstock in a factory follows the following distribution

Days	Probability
0	0.6
1	0.15
2	0.10
3	0.05
4	0.04
5	0.03
6	0.02
7	0.01

For this distribution, the outcome variable gives the number of days the factory has overstock.

For each possible outcome, there will be a probability associated with it.

In order to represent this distribution using Python, we will need to define two vectors:

- A vector that gives the values of the outcomes
- A vector that gives the values of the associated probabilities.

We can use `list` or `array` structure to store these two vectors

"Pythonize" a discrete distribution

In order to store the outcomes the PMF of a discrete distribution, we need a structure that supports vectors.

The first option is list.

List is a container that holds a number of other objects, in a given order.

We can use the following command to define an empty list

- `[]` or `list()`

▸ Run PYTHON

```python
a1=[]
a2=list()
print(a1,a2)

```

We can also define a list that contains known elements.

- `[a,b,c]`

In the following example, we create a list with 1, 2, 3, 4.

▸ Run PYTHON

```python
b=[1,2,3,4]
print(b)
```

The second option is `NumPy` array.

`NumPy` is a library that supports large and multi-dimensional arrays and matrices. Comparing to list, `Numpy` library alows us to define an array in a much easier way and contains many functions that allow us to perform array operations and computations very efficiently.

One direct way to define a `NumPy` array is to directly convert a list using `np.array(list)` function.

- `np.array([a,b,c,…])`

Directly define an array with elements a, b, c...

This is doing nothing but converting a list to an array by calling a function "array" that resides inside `NumPy` library. To do so, we need to first import `NumPy` library and shorten the name to "np" so that we can refer to this library using `np`.

We convert a list to a NumPy array, because NumPy package provides many efficient operations we could use.

The following example gives an array that contains 1, 2, 3.

Run PYTHON

```python
import numpy as np
a=np.array([1,2,3])
print(a)
```

We can also define evenly spaced numbers using the following method:

- `np.arange(start=0,stop,step=1)`

Define a range with the distance between consecutive values equals `step`. The range always includes start but excludes end.

The default value for `start` is 0, while the default value for `step` is 1.

The following example gives an array that contains 1, 2, 3 as well

Run PYTHON

```python
import numpy as np
#full specification
a=np.arange(start=1,stop=4,step=1)
print(a)
#a shorter version
a=np.arange(1,4)
print(a)
```

Array functions

NumPy library has many functions that can help us perform array computations on single array. These functions are highly efficient, one reason `NumPy` is so commonly used in the data science field.

Function	**Description**
np.sum(A)	Add all elements together
np.prod(A)	Multiply all elements together
np.cumprod(A)	A cumulative product: for each element, multiply all elements so far
np.cumsum(A)	A cumulative sum: for each element, add all elements so far
np.diff(A)	Difference between adjacent elements
np.exp(A)	Exponentiate each element
np.log(A)	Take the natural logarithm of each element
np.sqrt(A)	Take the square root of each element
np.min(A)	Find the minimum number of the array
np.max(A)	Find the maximum number of the array

Discussion: Think about a case when we will be using

- np.cumsum(A)
- np.diff(A)

Operation between arrays

We can also perform array operations very easily. The operation can happen between two arrays of the same length, between an array and a number.

Operation between arrays

Assuming that Array1 and Array2 have the same length

(1) Array1+Array2 will perform an element-wise addition
(2) Array1-Array2 will perform an element-wise subtraction
(3) Array1*Array2 will perform an element-wise multiplication
(4) Array1/Array2 will perform an element-wise division
(5) Array1**Array2 will perform an element-wise exponentiation

Operation between an array and a number

When performing the operation between a number and an array. The number will be expanded to an array of the same length as the other array. Then element-wise operation will be performed.

Important terms

Support

Support is a set that contains all and only the possible values the random variable could take.

Expected Value

The formula for the expected value of a discrete random variable follows:
$E(x) = \sum_{i=1}^{n} p_i x_i$
To apply this formula, we will need to first multiple all the outcomes with their probabilities. We will then sum up all the products.

Variance and standard deviation
The variance a discrete random variable can be calculated as follows:
$Var(x) = E(x^2) - (E(x))^2 = \sum_{i=1}^{n} p_i x_i^2 - (E(x))^2$
where $E(x)$ is has been computed above.
The standard deviation of a random variable is the square root of its variance.
$Std(x) = \sqrt{Var(x)}$

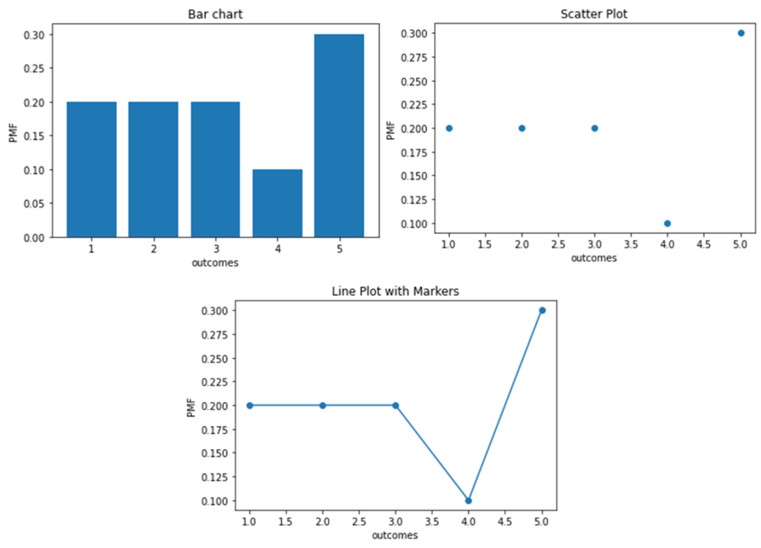
Visualizing a discrete distribution

Theoretical distribution

For a discrete distribution, when visualizing the PMF, we need to create two arrays: outcome array and the probability array.

We can use `matplotlib.pyplot` for visualization. We need to first import the package using

`import matplotlib.pyplot as plt`



In this case, we have three choices:

(1) bar chart

- `plt.bar(x,y)`

(2) scatter plot

- `plt.scatter(x,y)`

(3) line plot with makers.

- `plt.plot(x,y,marker="o")`
- We need to make sure that the outcome array follows an increasing order.

The following table gives useful functions to use for visualization

Function	**Description**
plt.bar(x,y)	Bar chart with x values on the $x$axis and y values on the y ax
plt.scatter(x,y)	Scatter chart with x values on the $x$axis and y values on the
plt.line(x,y,marker=”o”)	Line plot with markers with x values on the x axis and y values on the y axis
plt.xlabel(string)	Add a label to the x axis
plt.ylabel(string)	Add a label to the y axis
plt.title(string)	Add a title to the plot
plt.show()	End of the plot. Show plot

Exercise I (Overstock)

Now let's use what we have learned so far to study the overstock example:

Days	**Probability**
0	0.6
1	0.15
2	0.10
3	0.05
4	0.04
5	0.03
6	0.02
7	0.01

Exercise II (Binomial)

In this exercise, we explore a commonly used distribution: Binomial distribution.

Binomial distribution models the number of successes k out of n trials, where each trial is independent and has a probability of success equals p.

The Probability Mass Function (PMF) for a Binomial distribution follows:
$P(k) = \frac{n!}{k!(n-k)!}p^k(1-p)^{n-k}$ for $k = 0, 1, 2, ..., n$.

We have $E(k) = np$ and $Var(k) = np(1-p)$

To construct the PMF for all different values of k, we need to insert different values for $k = 0, 1, 2, 3, \ldots$
into $P(k) = \frac{n!}{k!(n-k)!}p^k(1-p)^{n-k}$

We can see that the above calculation is repetitive for each k value. This is usually a signal that we can use `NumPy` calculation to improve efficiency.

To translate the above operation to an array operation is very easy. We need to do three steps.

1. Make an array of all the possible outcomes. Let's name it `outcome`.
2. Replace `outcome` with k, which is the value of a single outcome.
3. Translate the functions/operations to the corresponding numpy functions/operation.

Especially, for the factorial part, we will use `scipy.special.factorial(outcome)` , which will compute the factorial of every element in an array.

Notice that $\frac{n!}{k!(n-k)!}$ is always between 0 and 1. However, when n is large, $n!$ will grow factorially. An alternative way is to directly compute $\frac{n!}{k!(n-k)!}$ using `scipy.special.comb(n,k)`.