W02_01_Histograms

September 12, 2022

1 Example: histograms

As we know, a *histogram* is a graph representing the frequency of occurrence of data, segregated into categories. If the data are real numbers, the categories are *bins*, e.g. $[0, 10]$, $[10, 20]$, $[20, 30]$. If the data are discrete, then the data values may be the categories, e.g. `A`, `B`, `C`.

In fact, a histogram is really the underlying data structure – a number representing frequency of occurrence, for each category – not the graphical representation of it.

In this notebook we'll develop a histogram and refine it a bit, illustrating several useful Python features and good Python style.

[1]:
```
def histogram(s):
    h = {} # represent histogram as a dict
    for c in s: # assume s is iterable
        if c in h:
            h[c] += 1 # increment
        else:
            h[c] = 1  # create key
    return h
```

[2]:
```
# test our histogram out
histogram([31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31])
```

[2]:
```
{31: 7, 28: 1, 30: 4}
```

1.0.1 Duck typing

One nice feature of Python is *duck typing*. "If it looks like a duck, and walks like a duck, and quacks like a duck, then treat it like a duck".

Contrary to some misunderstandings, Python objects do have types and these types never change. But a variable is really a *name* which points to an object. A variable can point to one object, and later on in the same program, to a different object of a different type. So *objects* have permanent types but *names* don't.

This becomes very useful when we write functions like `histogram(s)`. Here, the object referred to by the name `s` can be any iterable type. We don't care whether it's a string, list, tuple, or some other type we've never even heard of. As long as it knows how to iterate ("quacks like a duck"), then our code `for c in s` will work ok.

[3]:
```
histogram("mississippi") # can iterate over a string
```

[3]:
```
{'m': 1, 'i': 4, 's': 4, 'p': 2}
```

1.0.2 `collections.defaultdict`

A common pattern occurred above: we had to use an `if-else` (four lines) to take care of the special case where the key didn't already exist.

```
if c in h:
    h[c] += 1 # increment
else:
    h[c] = 1  # create key
```

This pattern is so common that they created a special `dict`-like type, `collections.defaultdict`, which can make our definition a bit nicer. When creating it, we pass in a function (of no arguments) which will be called to create a default when needed.

[4]:
```
from collections import defaultdict
def histogram(s):
    h = defaultdict(int) # int() -> 0
    for c in s: # assume s is iterable
        h[c] += 1
    return h
```

[5]:
```
int() # what does int mean as a function?
```

[5]:
```
0
```

1.0.3 Canonicalisation

Suppose we wanted to use our histogram to count letter frequencies in a large corpus.

[6]:
```
s = "It was the best of times, it was the worst of times..."
print(histogram(s))
```

```
defaultdict(<class 'int'>, {'I': 1, 't': 8, ' ': 11, 'w': 3, 'a': 2, 's': 6,
'h': 2, 'e': 5, 'b': 1, 'o': 3, 'f': 2, 'i': 3, 'm': 2, ',': 1, 'r': 1, '.': 3})
```

Look what happened: the count for `I` and for `i` are separate!

How to fix this? One solution is to *canonicalise* the data, that is for any data which can occur in multiple forms, map them all to a single, "canonical", form.

[7]:
```
def histogram(s):
    h = defaultdict(int) # int() -> 0
    for c in s: # assume s is iterable
        c = c.lower() # canonicalise. doesn't change s
        h[c] += 1
    return h
```

[8]:
```
print(histogram(s))
```

```
defaultdict(<class 'int'>, {'i': 4, 't': 8, ' ': 11, 'w': 3, 'a': 2, 's': 6,
'h': 2, 'e': 5, 'b': 1, 'o': 3, 'f': 2, 'm': 2, ',': 1, 'r': 1, '.': 3})
```

This works, but now our function is less general than before: it assumes that the elements of `s` are strings, or at least they quack like strings (they have a `.lower()` method). We can fix our problem and still retain generality like this:

[9]:
```
def histogram(s, canonicalise=None):
    h = defaultdict(int) # int() -> 0
    for c in s: # assume s is iterable
        if canonicalise:
            c = canonicalise(c)
        h[c] += 1
    return h
```

[10]:
```
def canonicalise_case(s): return s.lower()
print(histogram(s, canonicalise=canonicalise_case))
```

```
defaultdict(<class 'int'>, {'i': 4, 't': 8, ' ': 11, 'w': 3, 'a': 2, 's': 6,
'h': 2, 'e': 5, 'b': 1, 'o': 3, 'f': 2, 'm': 2, ',': 1, 'r': 1, '.': 3})
```

This is neat, because our function becomes much more *combinable* – more *lego-like*. For example:

[11]:
```
histogram([17.3, 17.4, 19.1, 19.2, 20.5, 20.6, 20.7],
          canonicalise=round)
```

[11]:
```
defaultdict(int, {17: 2, 19: 2, 20: 1, 21: 2})
```

1.0.4 Normalisation

Mapping numerical data from one range to another: * 0-1 Normalisation: map $[a, b]$ to $[0, 1]$ * -1 to 1: map $[a, b]$ to $[-1, 1]$ * z-scores: map so that data has mean 0 and variance 1 * vector-length normalisation: map so that resulting data, treated as vector, has length 1 * map to probabilities: map so that sum of resulting vector is 1

Normalisation means a few different things (sometimes the same as canonicalisation!), but one possible meaning is to map numerical data from by dividing by the sum, so that the new sum is 1. In a histogram, that means that instead of counting *occurences*, we will count *frequencies*. We can implement this as below.

[12]:
```
def histogram(s, normalise=False, canonicalise=None):
    h = defaultdict(int) # int() -> 0
    for c in s: # assume s is iterable
        if canonicalise:
            c = canonicalise(c)
        h[c] += 1
    if normalise:
```

```
        total = len(s)
        for c in h:
            h[c] /= total # normalise
    return h
```

[13]:
```
histogram("mississippi", normalise=True)
```

[13]:
```
defaultdict(int,
            {'m': 0.09090909090909091,
             'i': 0.36363636363636365,
             's': 0.36363636363636365,
             'p': 0.18181818181818182})
```

As before, we are able to add new functionality but retain generality, because this normalisation is optional. We are taking advantage of Python's optional keyword arguments.

By the way, it is the common idiom to use a `bool` argument to switch behaviours on or off (like `normalise`), but to provide `None` as the default value for an optional *function* (like `canonicalise`).

1.0.5 Sampling from a histogram

To *sample from a histogram* means to choose one of the keys with probability *weighted by the count*.

A common idea in AI is to learn a distribution from data and then sample from it. If the data is purely numerical, then of course that is familiar to us in statistics. The data could also be characters, words, tuples, or something else. Our histogram function can "learn" the distribution of any of these. But how can we then sample from it?

Recall that in a (normalised) histogram, the frequencies will sum to 1. Imagine the interval from 0 to 1 divided up into slots of different lengths. In the following algorithm, we choose a random value `r` $\in [0, 1]$ and see which slot it falls into:

```
[ m |    s    |    i    |  p  ]

0                            1

[                r           ]
```

[14]:
```
import random
def hist_sample(h):
    # we will assume h is normalised, and so sum(h.values()) == 1
    r = random.random() # in [0, 1]
    accum = 0
    for c in h:
        accum += h[c]
        if accum >= r:
            return c
    raise ValueError
```

[15]:
```
h = histogram("mississippi", normalise=True)
for i in range(10):
```

```
    print(hist_sample(h))
```

```
s
s
m
s
i
s
p
i
i
i
```

We can now generate individual letters and they'll be in the right frequencies. If we "learn" from a large corpus of English text, we'll see `e` as the most common letter.

But we might prefer to generate text at the *word level*. We can do it easily! We need to split our input sequence up into tokens and get rid of any punctuation.

[16]:
```
s = "It was the best of times, it was the worst of times..."
def canonicalise_word(w):
    return w.lower().strip(" .,?'()")
h = histogram(s.split(), normalise=True,
              canonicalise=canonicalise_word)

for i in range(10):
    print(hist_sample(h), end=" ")
```

```
the times of the of times was worst times of
```

The text is still nonsense, of course. n-grams is one technique which could be used to make it a bit more realistic.

2 File input/output

So far we have processed very small amounts of data. Let's process a whole book. We can get one in plain text from Project Gutenberg.

We'll use some shell commands to do so. We can execute shell commands directly in a Jupyter Notebook, using the `!` prefix. If you don't have `wget` on your system, don't worry – just download the file manually and put it in the current directory (if you are running in Spyder, you might need to tell Spyder to change the current directory also).

[17]:
```
!wget https://www.gutenberg.org/files/98/98-0.txt
!mv 98-0.txt data/tale.txt
```

```
--2019-09-19 08:23:28--  https://www.gutenberg.org/files/98/98-0.txt
Resolving www.gutenberg.org… 152.19.134.47
Connecting to www.gutenberg.org|152.19.134.47|:443… connected.
HTTP request sent, awaiting response… 200 OK
```

```
Length: 804335 (785K) [text/plain]
Saving to: ‘98-0.txt’

98-0.txt            100%[===================>] 785.48K  1.10MB/s    in 0.7s

2019-09-19 08:23:29 (1.10 MB/s) - ‘98-0.txt’ saved [804335/804335]
```

[20]:
```
fname = "data/tale.txt"
s = open(fname, encoding="utf8").read() # notice utf8 addition!
h = histogram(s.split(), normalise=True, canonicalise=canonicalise_word)

for i in range(10):
    print(hist_sample(h), end=" ")
```

```
imposing passed of the paris i three be being it
```

Above, we used `open()` to open the file for reading. We then used `.read()` to actually read it – all at once. An alternative is to use:

```
f = open(fname) # returns a File object
for line in f: # iterate over its lines
    # process line somehow...
```

To illustrate file *output*, let's generate some text and write it out. We have to pass the `w` (write) flag to `open`:

[19]:
```
gname = "data/tale_generated.txt"
g = open(gname, "w")
for i in range(100): # 100 lines of 50 words each
    output = []
    for j in range(50):
        output.append(hist_sample(h))
    g.write(" ".join(output) + ".\n") # write some text, full-stop, and newline
g.close()
```

Of course, there are many more options for file input/output, and of course file output can be combined with the usual Python string formatting.