The Hedonometer

discussion 04/29/2021 David Lauren Rashi

Sentiment analysis is a common tool for studying natural language. The basic question of sentiment analysis is: how **positive** or **happy** is this set of text? For example,

> I'm having such a good time at the apple orchard!!

has a "positive feeling," while

> I'm not mad, just disappointed.

doesn't sound nearly as pleasant. In sentiment analysis, we try to quantify the difference between these two sentences.

Sentiment analysis is a very common tool in business marketing, where it can be useful to examine, say, the sentiment of tweets in response to your ad. Within research, one of the most famous sentiment analysis projects is the **Hedonometer** (https://hedonometer.org/timeseries/en_all/) from the Computational Story Lab at the University of Vermont. Their analysis suggests that the summer of 2020, amidst a pandemic and controversy over police brutality, has been one of the true low points of the last decade -- at least on Twitter, where they collect their data.

In this Discussion activity, we'll create a much simpler sentiment analysis tool. To do this, we'll actually use a sentiment dictionary provided by the Computational Story Lab.

Part (A)

Run the following block of code. This code downloads some additional code, containing a statement initializing a large dictionary. Then, we `exec` ute that code to create the dictionary in our global namespace.

In [1]:
```
# run this block

# need this library for grabbing things from the internet
import urllib

# location of the file we want
url = "https://philchodrow.github.io/PIC16A/discussion/happiness_dictionary.py"

# get the data from the web (it's a .py file)
filedata = urllib.request.urlopen(url)

# run the code to define the dictionary
exec(filedata.read())
```

If you ran the code above, you now have a variable in your global scope called `happiness_dictionary` . It has a lot of entries!

In [2]:
```
len(happiness_dictionary)
```

Out[2]:
```
10221
```

The dictionary looks like this:

```
{'laughter' : 8.5,
 'happiness': 8.44,
 'love'     : 8.42,
 'happy'    : 8.3,
 'laughed'  : 8.26,
 'laugh'    : 8.22,
 'laughing' : 8.2,
 'excellent': 8.18,
 'laughs'   : 8.18,
 'joy'      : 8.16,
 ...
}
```

Each entry gives a score to a specific English word. For example, `"laughter"` is considered in this dictionary to be a slightly more positive word that `"happiness"` or " `love` ". The details of how these scores are chosen are very interesting, and people are still innovating in this space.

Part (B)

Create a class called `happiness_meter` (a less-sophisticated version of "Hedonometer"). Give it an `__init__()` method that accepts a single argument `D` . `D` should be a dictionary; if it's not, raise a `TypeError` . Otherwise, create an instance variable to hold `D` . Check that you can instantiate your class, using the `happiness_dictionary` as an argument.

In [35]:
```
# define your class here
class happiness_meter:
    def __init__(self, D):
        if type(D) != dict:
            raise TypeError("must be a dict")
        self.myDict = D

    @classmethod
    def strip_punctuation(self, s):
        return(s.translate(str.maketrans('', '', string.punctuation)))

    def score_sentence(self, s):
        myString = s.strip_punctuation
        myString = myString.split()

        temp = 0;
        for word in myString:
            if word in self.myDict.keys():



        pass
```

In [10]:
```
# check that you can instantiate it here

D = [1,2,3]
hello = happiness_meter(D)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-386c5706868c> in <module>
      2
      3 D = [1,2,3]
----> 4 hello = happiness_meter(D)

<ipython-input-8-41e9763e41a7> in __init__(self, D)
      3     def __init__(self, D):
      4         if type(D) != dict:
----> 5             raise TypeError("must be a dict")
      6         self.myDict = D
      7

TypeError: must be a dict
```

In [9]:
```
h = happiness_meter(happiness_dictionary)
len(h.myDict)
```

Out[9]:
```
10221
```

Part (C)

The following function strips out punctuation from a string. Modify this function and add it as a `strip_punctuation()` method of the `happy_meter` class. You can just add it to your class definition from Part (B).

```
def strip_punctuation(s):
    return(s.translate(str.maketrans('', '', string.punctuation)))
```

It's not necessary for this assignment for you to understand how this function works, but you are welcome to check the documentation for `str.translate()` and `str.maketrans()` if you're interested.

Compose a sentence with some punctuation in it and test out your implementation. You will need to start by `import` ing the `string` module.

In [34]:
```
import string
myString = "hello, today . sdjlfkjs."
a = happiness_meter.strip_punctuation(myString)

print(a)
b = a.split()

for word in b:
    print(word)
```

```
hello today  sdjlfkjs
hello
today
sdjlfkjs
```

Part (D)

Add a `score_sentence()` method to your class. This is the "main point" of the class and the most complex part of the assignment, so it's fine to spend some extra time here. If you make it through this part, then you are doing very well.

Here's what `score_sentence()` should do :

1. Accept a single argument `s` :
2. For each word in `s` , check whether the word is in the dictionary stored as an instance variable. If so, log in some way the happiness score associated with `s` . Don't forget that all the entries of `happiness_dictionary` are lower case. **For this part, you should use a `try` - `except` block** to handle the case in which a given word is not present in the dictionary.

3. Return the **average** score of all the matches found in `s`.

Again, you can just add this method to your class definition beneath Part (B).

Test `score_sentence()` on the examples at the beginning of the assignment and show the results. Do they match your expectations?

Your Solution

```
In [ ]:
```

If you've made it this far, then you've done very well. If it's close to the end of the scheduled Discussion period, add comments and docstrings to your assignment and turn it in. If there are more than 10 minutes left, proceed to Parts (E) and (F).

Part (E)

Add a `score_passage(p)` method, which scores a string with multiple sentences.

`score_passage(p)` should return a dictionary, whose keys are the individual sentences in passage `p` and whose values are the happiness scores associated with those sentences. You may assume that all sentences end with a period `.`, and that the period `.` does not appear in any other contexts.

For example:

```
HM = happy_meter(happiness_dictionary)
HM.score_passage("I'm having such a good time at the apple orchard. I'm not mad, just dis
appointed.")

# output
{"I'm having such a good time at the apple orchard": 5.717777777777777,
 "I'm not mad, just disappointed.": 3.9}
```

Add your implementation of `score_passage()` to your class definition under Part (B). Then, test out your implementation on the first two paragraphs of Dr. Martin Luther King's famous "I have a Dream" speech, which are initialized as `p` in the block below.

Your Solution

```
In [ ]: p = "I am happy to join with you today in what will go down in history as the greatest demonstrat
```

Part (F)

Examine the output of the `score_passage()` function from Part (E). Do you agree with its relative rankings of the sentences? What are some of the limitations of using word-by-word happiness scores?

The method of measuring sentences using only the individual words contained in them, without regard for the order of the words, is called the *bag-of-words* model. What are some of the benefits and drawbacks of using the bag-of-words model? Discuss and write down a few of your considerations.

Your Response Here