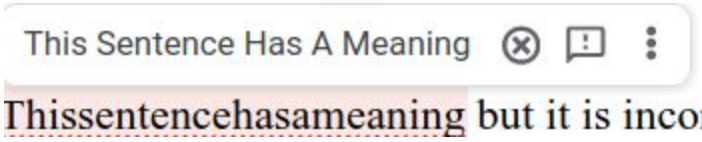
Name : Abhishek Kushwaha Reg Number : 18BCE0492

Digital Assignment

So, two days back, I was copying my Nasscom DA (Please don't judge this DA due to that, this is fully original, plus even our teacher approved copying), and I faced a very interesting issue.
I was copy and pasting from the pdf, to google docs and I found that the spacing between letters wasn't being recorded(And just imagine if I am copying a paragraph, I can't literally sit down and give spaces between words, defeats the whole purpose of copy pasting in the first part). That time I discovered a small popup feature like this in google docs, the example given below.



Now this kind of looks cool to me, and I thought, maybe this was possible for me to implement.

So I started googling……..

And I discovered some **good** stuff.

I did a lot of futile searches, and finally got something technical enough to proceed with : ***Zipf's Law***
This is the first pdf that helped me get a bit of background regarding this Zipf's Mystery:

https://www.ling.upenn.edu/~kroch/courses/lx107/The_mystery_of_Zipf.pdf

A worthwhile interesting observation from this pdf was:

"*Given a large body of language (that is, a long book — or every word uttered by Plus employees during the day), the frequency of each word is close to inversely proportional to its rank in the frequency table*"

This can be summarized by the following formula :

$$P_n \propto 1/n^a$$

where a is close to 1. This is known as a "**power law**" (heard something similar in maths too, JEE days as far as I remember) and suggests that the most frequent word will occur approximately twice as often as the second most frequent word, which occurs twice as often as the fourth most frequent word, etc.

Some more references to this that I did refer (but not maybe that much as this pdf, were) :

1) https://www.sciencedaily.com/releases/2017/08/170810082147.htm
2) https://en.wikipedia.org/wiki/Zipf%27s_law

Finally, I got inspired to code a little on my own(tbh, I didn't have time to research if something like this exists or not, cause time was short, but I did end up putting a bit of my CP skills and knowledge of python to use).

My basic objective was to get a long sequence of characters into some meaningful text automatically(My program should space my copy pasted document for me, not myself) .

An approximation that I made was that distribution of all words is independent.

Now to get a most probable sentence, I have to break the sequence in such a way that the product of the probability of each individual word is maximum and it's

easy to compute it with a little bit of competitive coding. Instead of directly using the probability I used a ‘cost’ - defined as the logarithm of the inverse of the probability to avoid overflows.

By the way, I used this : https://controlc.com/c1666a6b for building my cost Dictionary. I found this on the web, it doesn't have a big corpus like Brown, it’s just a subset of wikipedia with around 125k words.

Finally, time to put it in action :

Step 1 : Making the cost dictionary :

```
with gzip.open(path_of_my_custom_wordlist) as f:
     words = f.read().decode().split()
     wordcost = dict((k, log((i+1)*log(len(words)))) for i,k in
enumerate(words))
     maxword = max(len(x) for x in words)
```

Step 2 : Finding the best match for the i first characters, assuming cost has been built for the i-1 first characters

```
def best_match(i):
     candidates = enumerate(reversed(cost[max(0, i-maxword):i]))
     return min((c + self.wordcost.get(s[i-k-1:i].lower(), 9e999), k+1)
for k,c in candidates)
```

```
   cost = [0]
   for i in range(1,len(s)+1):
     c,k = best_match(i)
     cost.append(c)
```

Step 3 : Now we use backtracking to recover the minimal cost string(Handling apostrophe and digits as well)

```
output = []
   i = len(s)
   while i>0:
     c,k = best_match(i)
     assert c == cost[i]
     counter = True
     if not s[i-k:i] == "'":
       if len(output) > 0:
         if output[-1] == "'s" or (s[i-1].isdigit() and
out[-1][0].isdigit()):
           output[-1] = s[i-k:i] + out[-1
           counter = False
     if counter:
       output.append(s[i-k:i])
     i -= k
```

This ‘output’ needs to be reversed and sent back

Now the 2nd and third step can be made a function, like func1 and utilise for the space inference:

```
l = [func1(x) for x in re.compile("[^a-zA-Z0-9']+").split(s)]
print(" ".join([item for sublist in l for item in sublist]))
```

Here we have made use of dynamic programming to infer location of spaces.

Finally to test it out :

This model could still suffer due to proper nouns not splitting properly or improper splitting, as well as losing out of the capitalization(though it can be tweaked by storing positions of the capital letters in previous operations and capitalizing it in the end).