You use autocorrect every day on your cell phone and computer. In this project:

- Get a word count given a corpus
- Get a word probability in the corpus
- Manipulate strings
- Filter strings
- Implement Minimum edit distance to compare strings and to help find the optimal path for the edits.

0.1 - Edit Distance

we will implement models that correct words that are 1 and 2 edit distances away.

- We say two words are n edit distance away from each other when we need n edits to change one word into another.

An edit could consist of one of the following options:

- Delete (remove a letter): ‘hat’ => ‘at, ha, ht’
- Switch (swap 2 adjacent letters): ‘eta’ => ‘eat, tea,...’
- Replace (change 1 letter to another): ‘jat’ => ‘hat, rat, cat, mat, ...’
- Insert (add a letter): ‘te’ => ‘the, ten, ate, ...’

You will be using the four methods above to implement an Auto-correct.

- To do so, we will compute probabilities that a certain word is correct given an input.

The goal of our spell check model is to compute the following probability:

$$P(c|w) = \frac{P(w|c) \times P(c)}{P(w)} \tag{Eqn-1}$$

- Equation 1 says that the probability of a word being correct $P(c|w)$ is equal to the probability of having a certain word w, given that it is correct $P(w|c)$, multiplied by the probability of being correct in general $P(C)$ divided by the probability of that word w appearing $P(w)$ in general.
- To compute equation 1, you will first import a data set and then create all the probabilities that you need using that data set.

1 - Data Preprocessing

```
import re
from collections import Counter
import numpy as np
import pandas as pd
```

a - process_data

Implement the function `process_data` which

1) Reads in a corpus (text file)

2) Changes everything to lowercase

3) Returns a list of words.

```
def process_data(file_name):

    words = []

    with open(file_name) as fh:
        data=fh.read()



    data=data.lower()
    #Convert every word to lower case and return them in a list.
    words=re.findall(r'\w+', data)
    # we split the data into a list words while removing the new line character


    return words
```

```
word_l = process_data('shakespeare.txt')
vocab = set(word_l)  # this will be your new vocabulary
print(f"The first ten words in the text are: \n{word_l[0:10]}")
print(f"There are {len(vocab)} unique words in the vocabulary.")
```

```
The first ten words in the text are:
['this', 'is', 'the', '100th', 'etext', 'file', 'presented', 'by', 'project', 'gutenberg']
There are 23902 unique words in the vocabulary.
```

b - get_count

Implement a `get_count` function that returns a dictionary

- The dictionary's keys are words
- The value for each word is the number of times that word appears in the corpus.

```
def get_count(word_l):

    word_count_dict = {}

    word_count_dict=Counter(word_l)
    # counter for every unique word in the text document
    return word_count_dict
```

```
word_count_dict = get_count(word_l)
print(f"There are {len(word_count_dict)} key values pairs")
print(f"The count for the word 'thee' is {word_count_dict.get('thee',0)}")
```

```
There are 23902 key values pairs
The count for the word 'thee' is 3181
```

c - get_probs

Given the dictionary of word counts, compute the probability that each word will appear if randomly selected from the corpus of words.

$$P(w_i) = \frac{C(w_i)}{M} \tag{Eqn-2}$$

where

$C(w_i)$ is the total number of times w_i appears in the corpus.

M is the total number of words in the corpus.

```
def get_probs(word_count_dict):

    probs = {}

    m=sum(word_count_dict.values())
    for word in word_count_dict:
        probs[word]=word_count_dict.get(word)/m
    # probability of each word occuring in the text document
    return probs
```

```
probs = get_probs(word_count_dict)
print(f"Length of probs is {len(probs)}")
print(f"P('thee') is {probs['thee']:.4f}")
```

```
Length of probs is 23902
P('thee') is 0.0034
```

2 - String Manipulations

Now that you have computed $P(w_i)$ for all the words in the corpus,we gave writtn e a few functions to manipulate strings so that we can edit the erroneous strings and return the right spellings of the words.WE will use functions:

- `delete_letter`: given a word, it returns all the possible strings that have **one character removed**.
- `switch_letter`: given a word, it returns all the possible strings that have **two adjacent letters switched**.
- `replace_letter`: given a word, it returns all the possible strings that have **one character replaced by another different letter**.
- `insert_letter`: given a word, it returns all the possible strings that have an **additional character inserted**.

a - delete_letter

given a word it returns a list of strings with one character deleted.

For example, given the word **nice**, it would return the set: {'ice', 'nce', 'nic', 'nie'}.

```
def delete_letter(word, verbose=False):


    delete_l = []
    split_l = []


    split_l=[(word[:i],word[i:]) for i in range(len(word)+1)]
    delete_l=[L+R[1:] for L,R in split_l if R]

    if verbose: print(f"input word {word}, \nsplit_l = {split_l}, \ndelete_l = {delete_l}")

    return  delete_l


delete_word_l = delete_letter(word="bend",
                        verbose=True)
```

```
input word bend,
split_l = [('', 'bend'), ('b', 'end'), ('be', 'nd'), ('ben', 'd'), ('bend', '')],
delete_l = ['end', 'bnd', 'bed', 'ben']
```

b - switch_letter

It takes in a word and returns a list of all the possible switches of two letters that are adjacent to each other

- For example, given the word 'eta', it returns {'eat', 'tea'}, but does not return 'ate'.

```
def switch_letter(word, verbose=False):


    switch_l = []
    split_l = []


    split_l=[(word[:i],word[i:]) for i in range(len(word)+1)]

    for i in range(0,len(word)-1):
        s=word[:i]+word[i+1]+word[i]+word[i+2:]
        switch_l.append(s)



    if verbose: print(f"Input word = {word} \nsplit_l = {split_l} \nswitch_l = {switch_l}")

    return switch_l


switch_word_l = switch_letter(word="eta",
                         verbose=True)
```

```
Input word = eta
split_l = [('', 'eta'), ('e', 'ta'), ('et', 'a'), ('eta', '')]
switch_l = ['tea', 'eat']
```

c - replace_letter

we implement a function that takes in a word and returns a list of strings with one **replaced letter** from the original word.

```
def replace_letter(word, verbose=False):

    letters = 'abcdefghijklmnopqrstuvwxyz'


    split_l = []
```

```
    split_l=[(word[:i],word[i:]) for i in range(len(word))]
    replace_set=[L+C+R[1:] for L,R in split_l if R for C in letters]

    replace_l = sorted(list(replace_set))
    replace_l= [i for i in replace_l if i != word]
    if verbose: print(f"Input word = {word} \nsplit_l = {split_l} \nreplace_l {replace_l}")

    return replace_l


replace_l = replace_letter(word='can',
                              verbose=True)
```

```
Input word = can
split_l = [('', 'can'), ('c', 'an'), ('ca', 'n')]
replace_l ['aan', 'ban', 'caa', 'cab', 'cac', 'cad', 'cae', 'caf', 'cag', 'cah', 'cai', 'caj', 'cak', 'cal', 'cam', 'cao', 'cap', 'caq'
```

d- insert_letter

Now we implement a function that takes in a word and returns a list with a letter inserted at every offset.

```
def insert_letter(word, verbose=False):

    letters = 'abcdefghijklmnopqrstuvwxyz'
    insert_l = []
    split_l = []


    split_l=[(word[:i],word[i:]) for i in range(len(word)+1)]
    insert_l=[L+C+R for L,R in split_l  for C in letters]



    if verbose: print(f"Input word {word} \nsplit_l = {split_l} \ninsert_l = {insert_l}")

    return insert_l


insert_l = insert_letter('at', True)
```

```
Input word at
split_l = [('', 'at'), ('a', 't'), ('at', '')]
insert_l = ['aat', 'bat', 'cat', 'dat', 'eat', 'fat', 'gat', 'hat', 'iat', 'jat', 'kat', 'lat', 'mat', 'nat', 'oat', 'pat', 'qat', 'rat
```

3 - Edit one letter

Now that we have implemented the string manipulations, we will create two functions that, given a string, will return all the possible single and double edits on that string. These will be `edit_one_letter()` and `edit_two_letters()`.

```
def edit_one_letter(word, allow_switches = True):


    edit_one_set = set()


    edit_one_set.update(delete_letter(word))
    edit_one_set.update(switch_letter(word))
    edit_one_set.update(insert_letter(word))
    edit_one_set.update(replace_letter(word))

    return set(edit_one_set)


tmp_word = "at"
tmp_edit_one_set = edit_one_letter(tmp_word)

tmp_edit_one_l = sorted(list(tmp_edit_one_set))

print(f"input word {tmp_word} \nedit_one_l \n{tmp_edit_one_l}\n")
```

```
input word at
edit_one_l
['a', 'aa', 'aat', 'ab', 'abt', 'ac', 'act', 'ad', 'adt', 'ae', 'aet', 'af', 'aft', 'ag', 'agt', 'ah', 'aht', 'ai', 'ait', 'aj', 'ajt',
```

a - Edit Two Letters

Exercise 9 - edit_two_letters

wegeneralize this to implement to get two edits on a word. To do so, we would have to get all the possible edits on a single word and then for each modified word, we would have to modify it again.

```
def edit_two_letters(word, allow_switches = True):


    edit_two_set = set()


    edit_one_set=set()
    edit_one_set.update(delete_letter(word))
    edit_one_set.update(switch_letter(word))
    edit_one_set.update(insert_letter(word))
    edit_one_set.update(replace_letter(word))


    L=list(edit_one_set)
    for i in L:
        edit_two_set.update(insert_letter(i))
        edit_two_set.update(replace_letter(i))
        edit_two_set.update(switch_letter(i))
        edit_two_set.update(delete_letter(i))




    return set(edit_two_set)


tmp_edit_two_set = edit_two_letters("a")
tmp_edit_two_l = sorted(list(tmp_edit_two_set))
print(f"Number of strings with edit distance of two: {len(tmp_edit_two_l)}")
print(f"First 10 strings {tmp_edit_two_l[:10]}")
print(f"Last 10 strings {tmp_edit_two_l[-10:]}")
```

```
Number of strings with edit distance of two: 2654
First 10 strings ['', 'a', 'aa', 'aaa', 'aab', 'aac', 'aad', 'aae', 'aaf', 'aag']
Last 10 strings ['zv', 'zva', 'zw', 'zwa', 'zx', 'zxa', 'zy', 'zya', 'zz', 'zza']
```

4 - Suggest Spelling Suggestions

Now you will use your `edit_two_letters` function to get a set of all the possible 2 edits on your word. You will then use those strings to get the most probable word you meant to type a.k.a your typing suggestion.

```
def get_corrections(word, probs, vocab, n=4, verbose = False):

    suggestions = []
    n_best = []

    ### START CODE HERE ###
    #Step 1: create suggestions as described above
    one_edit=list(edit_one_letter(word))
    two_edit=list(edit_two_letters(word))
    list1=set()
    for i in vocab:
        if(i ==word):
            list1.add(i)
            break
    list2=set()
    for i in one_edit:
        if i in vocab:
            list2.add(i)
```

```
    list3=set()
    for i in two_edit:
        if i in vocab:
            list3.add(i)
    suggestions=list1 or list2 or list3
    best_words={}
    for i in list(suggestions):
        if i in probs:
            best_words[i]=probs.get(i)
        else:
            best_words[i]=0

    keys = list(best_words.keys())
    values = list(best_words.values())
    sorted_value_index = np.argsort(values)[::-1]
    sorted_dict = {keys[i]: values[i] for i in sorted_value_index}
    if (n>len(sorted_dict)):
        for i in sorted_dict:
            n_best.append((i,sorted_dict.get(i)))
    else:

        d1= dict(list(sorted_dict.items())[0:n])
        for i in d1:
            n_best.append((i,d1.get(i)))




    if verbose: print("entered word = ", word, "\nsuggestions = ", suggestions)

    return n_best


my_word = 'dys'
tmp_corrections = get_corrections(my_word, probs, vocab, 2, verbose=True)
for i, word_prob in enumerate(tmp_corrections):
    print(f"word {i}: {word_prob[0]}, probability {word_prob[1]:.6f}")

print(f"data type of corrections {type(tmp_corrections)}")
```

```
entered word =  dys
suggestions =  {'dye', 'dy', 'dis', 'des', 'days'}
word 0: days, probability 0.000225
word 1: dis, probability 0.000005
data type of corrections <class 'list'>
```

4a - Minimum Edit Distance

Now that we have implemented our auto-correct, we need to evaluate the similarity between two strings. For example: 'waht' and 'what'

```
def min_edit_distance(source, target, ins_cost = 1, del_cost = 1, rep_cost = 2):

    m = len(source)
    n = len(target)
    #initialize cost matrix with zeros and dimensions (m+1,n+1)
    D = np.zeros((m+1, n+1), dtype=int)



    # Fill in column 0, from row 1 to row m, both inclusive
    for row in range(1,m+1):
        D[row,0] = D[row-1,0]+del_cost

    # Fill in row 0, for all columns from 1 to n, both inclusive
    for col in range(1,n+1):
        D[0,col] = D[0,col-1]+ins_cost

    # Loop through row 1 to row m, both inclusive
    for row in range(1,m+1):

        # Loop through column 1 to column n, both inclusive
        for col in range(1,n+1):

            # Intialize r_cost to the 'replace' cost that is passed into this function
            r_cost = rep_cost
```

```
            # Check to see if source character at the previous row
            # matches the target character at the previous column,
            if (source[row-1]==target[col-1]):
                # Update the replacement cost to 0 if source and target are the same
                r_cost = 0

            # Update the cost at row, col based on previous entries in the cost matrix

            D[row,col] = min(D[row-1,col]+del_cost,D[row,col-1]+ins_cost,D[row-1,col-1]+r_cost)

    # Set the minimum edit distance with the cost found at row m, column n
    med = D[m,n]


    return D, med


source =  'play'
target = 'stay'
matrix, min_edits = min_edit_distance(source, target)
print("minimum edits: ",min_edits, "\n")
idx = list('#' + source)
cols = list('#' + target)
df = pd.DataFrame(matrix, index=idx, columns= cols)
print(df)
```

```
minimum edits:  4

   #  s  t  a  y
#  0  1  2  3  4
p  1  2  3  4  5
l  2  3  4  5  6
a  3  4  5  4  5
y  4  5  6  5  4
```

+ Code + Text

✓ 0s completed at 18:26