HW 3

CS 216, Everything Data, Spring 2020

DUE: Monday Feb. 10 by 4:40 pm (class time)

In this assignment, you will investigate record linkage or matching. In HW 2, you frequently joined tables from the `Congress` database in order to form queries. In that database, we had the luxury of unique IDs such that we could match records between different tables based on these IDs. As discussed in Lecture 3, frequently we wish to merge datasets from multiple sources that may not include exactly identical unique IDs for performing join operations. In this assignment, you will practice techniques for dealing with this problem.

You will include all of your answers for this assignment within this notebook. You will then convert your notebook to a .pdf and a .py file to submit to gradescope (submission instructions are included at the bottom).

Please take note of the course collaboration policy (https://sites.duke.edu/compsci216s2020/policies/). You may work alone or with a single partner. If you work with a partner, you may not split up the assignment; you should work together in-person or complete parts independently and come together to discuss your solutions. In either case, you are individually responsible for your work, and should understand everything in your submission.

Part 1: Getting Started

We will work with the `restaurants_a.csv` and `restaurants_b.csv` datasets for this assignment. Each contain five columns: id (a numeric index serving as a unique id, not correlated across the datasets), name (of the restaurant), address (the street address), city, and type (the type of restaurant). You can access `.csv` files for these datasets from the box folder containing this Jupyter notebook. Make sure that you download the files and have them in the same working directory as your notebook.

This entire assignment can be completed using only `Python`, but you are welcome to use `Pandas` or any other standard libraries to help you.

In [1]:
```
import pandas as pd
df_a = pd.read_csv('restaurants_a.csv')
df_b = pd.read_csv('restaurants_b.csv')
df_a.head()
```

Out[1]:

	id	name	address	city	type
0	0	belvedere the	9882 little santa monica blvd.	beverly hills	pacific new wave
1	1	triangolo	345 e. 83rd st.	new york	italian
2	2	broadway deli	3rd st. promenade	santa monica	american
3	3	lettuce souprise you (at)	3525 mall blvd.	duluth	cafeterias
4	4	otabe	68 e. 56th st.	new york	asian

Part 2: Record Linkage and Similarity Measures

When multiple datasets are drawn from different sources, it is frequently the case that they do not have a common unique identification system like we were lucky to have in the `congress` database from homework 2. That can make it difficult to join the datasets, which requires more complex forms of record linkage.

The simplest approach to record linkage is simply to pick an attribute and join on that. In many cases this is inadequate: there may be mispellings of names, nicknames, or other slight variations such that some records that should be merged do not end up merged. This frequently leaves us with duplicate records.

For the remainder of the assignment, we will think of the task of record linkage as that of determining duplicates after performing an outer join. **Your task will be to identify the pairs of records between the `restaurants_a` and `restaurants_b` datasets that should be merged because they refer to the same entity.** In the end, you will produce a set of such pairs.

Problem A: Using Edit Distance

One way to catch small spelling errors (substituting i and e, swapping the order of letters, omitting an apostrophe, etc.) is to compute the edit distance between strings. Recall from Lecture 3 that the edit distance is the number of single character delete, insert, and substitute operations necessary to transform one string into another. We have included code to compute the edit distance between two strings for you in the file `hw3_utils.py` , you can import the function and see an example of it's use below.

In [6]:
```python
from hw3_utils import edit_dist
print(edit_dist("hello!", "hallo!"))
```

```
1
```

For problem A, use `edit_dist` to try to find likely mispellings of restaurant names between `df_a` and `df_b` . Specifically, print the names of all pairs of records (one from `df_a` and the other from `df_b`) such that the two names have edit distance at least 1 and at most 2 (note that if two strings have edit distance 0, they are exactly the same; you do not need to print these for problem A). Among these, which pairs do you think are "real" mispellings, and which do you think might actually be different restaurants?

In [99]:
```python
from hw3_utils import edit_dist
for x in df_a['name']:
    for y in df_b['name']:
        if 0 < edit_dist(x,y) < 3:
            print(x, ", " , y)
```

```
l'orangerie ,  l orangerie
indigo coast grill ,  indigo coastal grill
boulavard ,  boulevard
drago ,  spago
felidia ,  filidia
march ,  marichu
mesa grill ,  sea grill
uncle nicks ,  uncle nick's
```

l'orangerie, indigo coast grill, boulavard, felidia, uncle nicks are most likely mispellings whereas (drago , spago), (march , marichu), and (mesa grill , sea grill) are most likely different restaurants

Problem B: Trigram-Decomposition

Another way to catch spelling and representation errors we discussed in class is to compute the jaccard similarity of strings as represented by their trigram set decompositions. Recall that the trigram set decomposition of a string is the set of all 3 character contiguous substrings. We will also include the 1 and 2 character contiguous substrings at the beginning and end of the string. For example, the trigram decomposition of the string "hello!" is {'h', 'he', 'hel', 'ell', 'llo', 'lo!', 'o!', '!'}. For problem b, implement a function that computes the trigram decomposition of a string, and verify that it produces the correct output on the example "hello!" (note that the order in a set does not matter, so {'!', 'o!', 'lo!',...} would be equally correct).

If you are still learning Python, you might find the following tips helpful in completing problem B.

- Strings can be indexed and sliced in Python as if they were arrays of characters. So, for example, if you want to get the first character of `my_str = "hello!"`, you could write `my_str[0]`, which would return "h". Similarly, you can use standard slicing syntax to get substrings, for example, `my_str[0:3]` would return "hel". For more about Python strings, see https://docs.python.org/3.7/tutorial/introduction.html#strings (https://docs.python.org/3.7/tutorial/introduction.html#strings).
- Sets are a basic data structure in Python. You can initialize an empty by, for example, `my_set = set()`. Sets can contain arbitrary data types (strings, of course, are what we are interested in here). You can add an element to a set using `add`, for example, `my_set.add('h')` would result in our empty set from before being updated to include the string "h". For more about Python sets, see https://docs.python.org/3.7/tutorial/datastructures.html#sets (https://docs.python.org/3.7/tutorial/datastructures.html#sets).

In [218]:
```
def generate_trigrams(s):
    output = set()
    if(len(s)<=3):
        for x in range(len(s)):
            output.add(s[0:x+1])
        for y in range(len(s)):
            output.add(s[y:])
        return output
    else:
        for x in range(3):
            output.add(s[0:x+1])
        for i in range(len(s)-4):
            output.add(s[i:i+3])
        for i in range(len(s)-4):
            output.add(s[i:i+3])
        output.add(s[len(s)-4:len(s)-1])
        output.add(s[len(s)-3:len(s)])
        output.add(s[len(s)-2:])
        output.add(s[len(s)-1])
        return output

print(generate_trigrams("Hello!"))
```

```
{'H', 'Hel', 'o!', 'lo!', 'llo', '!', 'He', 'ell'}
```

Problem C: Jaccard Similarity

Once we can compute the trigram set decomposition of a string, we can use that to compute the Jaccard similarity between two strings. Recall from lecture 3 that the Jaccard similarity of two sets A and B is the ratio of the size of the intersection of A and B over the size of the union of A and B. In our context, the sets will be the trigram decomposotions we get for two different strings. For problem C, implement a function that computes the Jaccard similarity between two strings. Verify that your function correctly computes that the Jaccard similarity of the strings "hello!" and "hallo!" is 5/11 (~0.455).

Note that Python sets have built in functions for computing intersections and unions; feel free to take advantage: https://docs.python.org/3.7/tutorial/datastructures.html#sets (https://docs.python.org/3.7/tutorial/datastructures.html#sets). The function `len` will also give you the size of a set.

In [220]:
```
def jaccsim(a, b):
    intersect = generate_trigrams(a) & generate_trigrams(b)
    union = generate_trigrams(a) | generate_trigrams(b)
    return(len(intersect)/len(union))
print(jaccsim("hello!", "hallo!"))
```

```
0.45454545454545453
```

Problem D: Using Jaccard Similarity

For problem D, use your function for computing Jaccard similarity from problem C to try to find likely mispellings of restaurant names between `df_a` and `df_b`. Specifically, print the names of all pairs of records (one from `df_a` and the other from `df_b`) such that the two names have Jaccard similarity at least 0.5 and strictly less than 1 (note that if two strings have Jaccard similarity of 1, they are (likely) exactly the same; you do not need to print these for problem D). State the mispellings you found in problem A that you do not find with this method, and also the mispellings you find with this method that you did not find in problem A.

In [225]:
```
def jaccsimlist(a, b):
    listofsims=[]
    for x in a['name']:
        for y in b['name']:
            if (0.5 <= jaccsim(x,y) < 1):
                listofsims.append(x)
                listofsims.append(y)
    return listofsims
print(jaccsimlist(df_a, df_b))
```

```
["l'orangerie", 'l orangerie', 'indigo coast grill', 'indigo coastal gr
ill', 'boulavard', 'boulevard', 'tillerman  the', 'tillerman', 'gotham
bar & grill', 'gotham bar and grill', "pano's & paul's", "pano's and pa
ul's", 'ritz-carlton cafe (buckhead)', 'cafe  ritz-carlton  buckhead',
'felidia', 'filidia', 'uncle nicks', "uncle nick's"]
```

Mispellings from problem A not found with this method: (drago , spago), (march , marichu), mesa grill , sea grill

Mispellings from this method not in problem A: (tillerman the , tillerman), (gotham bar & grill , gotham bar and grill), (pano's & paul's , pano's and paul's), (ritz-carlton cafe (buckhead) , cafe ritz-carlton buckhead)

Part 3: Record Linking the Restaurant Datasets

Now you will compute your overall matching of records that should be merged between the two datasets. **You should format your answer as a *set* of pairs of ids**, one from `df_a` and another from `df_b`, such that you predict the corresponding records should be merged. For example, `{(661, 801), (228, 388), (304, 735), (101, 102)}` identifies that the records with ids 661 and 801 should be merged, 228 and 388 should be merged, 304 and 735 should be merged, and 101 and 102 should be merged. The ids are unique between the datasets, so you don't have to worry about a given id appearing in both. You also don't have to worry about the order of the pairs: `(661, 801)` will be treated equivalently as `(801, 661)` for the purpose of quantifying error (and if you include both orders, the duplicate will be removed when quantifying error).

A couple of tips if you are new to using Python:

- Note that Python has sets as a built-in data structure. See instructions for problems B and C or see https://docs.python.org/3.7/tutorial/datastructures.html#sets (https://docs.python.org/3.7/tutorial/datastructures.html#sets) for more info.
- Python also supports tuples as a built-in data structure. In our case, we are particularly interested in pairs. Say you have two ids: `id_a` and `id_2`, and you want to store the pair. You can do so simply by `id_pair = (id_a, id_b)`. You can subsequently index the pair as if it was a list. For example, `id_pair[0]` would return `id_a`. See https://docs.python.org/3.7/tutorial/datastructures.html#tuples-and-sequences (https://docs.python.org/3.7/tutorial/datastructures.html#tuples-and-sequences) for more info.
- The format of the answer as given above is thus just a set of tuples. You can initialize an empty set with `my_set = set()`, and if you have a pair `id_pair = (id_a, id_b)`, you can add it to the set with `my_set.add(id_pair)`.

In [228]:
```python
def setofpairs(df_a,df_b):
    listofmatches=jaccsimlist(df_a, df_b)
    setofids=set()
    for x in range(len(listofmatches)):
        if x % 2==0:
            for a in range(len(df_a['name'])):
                if (listofmatches[x]==df_a['name'][a]):
                    for b in range(len(df_b['name'])):
                        if(listofmatches[x+1]==df_b['name'][b]):
                            setofids.add((df_a['id'][a],df_b['id'][b]))
    return(setofids)
print(setofpairs(df_a,df_b))
```

```
{(263, 277), (95, 743), (561, 828), (240, 265), (204, 424), (62, 89),
(12, 22), (11, 326), (171, 269)}
```

Quantifying Error

First, we want to make sure we understand how to quantify how "good" a particular matching is. Recall from Lecture 3 that we discussed the *precision* and *recall* of a matching as measures of quality.

- The **precision** is the fraction of our predictions that really should be merged.
- The **recall** is the fraction of the set of true pairs of records that go together that we predict should be merged.

It is easy to do well on just one of these dimensions, but achieving high precision and recall simultaneously can be difficult. The *F1 score* is the harmonic mean of precision and recall. More precisely, $F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$. Getting a high F1 score requires doing well on both measures simultaneously.

We have included code to compute the precision, recall, and F1 score of a set of pairs of formatted as specified above. Below we import them and show the output on the example matching from above.

In [223]:
```
from hw3_utils import precision, recall, f1_score

bad_match = {(661, 801), (228, 388), (304, 735), (101, 102)}
print("Precision: ", precision(bad_match))
print("Recall: ", recall(bad_match))
print("F1 score: ", f1_score(bad_match))
```

```
Precision:  0.75
Recall:  0.04918032786885246
F1 score:  0.0923076923076923
```

Problem E

As you can see, the match above is pretty poor; it covers less than 5% of the pairs that should really be merged. Your task is to compute a better match. For full credit (100%), your match should achieve an F1 score of at least 0.93. Your code below should compute your match, and then print the F1 score achieved (you do not need to print the entire match itself). After that, you should describe in a few sentences what your record linkage code is doing. Note that there is no one "correct" way to compute a good match, you should feel free to use any of the information contained in `df_a` and `df_b`, along with the tools above.

If you need help getting started, here are a few hints:

- You can always start with the records that have exact matches on names.
- You can also use the similarity methods from above with different parameters to get mispelled names.
- Apart from the names and ids, you can look at the city, address, and type of restaurant for every record, and try to match on those as well.
- You can always compute multiple matches and then combine them.
- Check out the precision and recall of your results. If precision is low but recall is high, you are trying to link too many pairs. If recall is low, but precision is high, you are not trying to link enough paris.
- If you are completely stuck, the "answers" (that is, the ground truth matches) are included in `hw3_utils.py`, so you can try checking out the ones you are missing to find out why your code isn't capturing them. Note that you will not receive credit for hard coding these values.

In [229]:
```python
from hw3_utils import precision, recall, f1_score

def jaccsimnamelist(a, b):
    listofsims=[]
    for x in a['name']:
        for y in b['name']:
            if (0.5 <= jaccsim(x,y) <= 1):
                listofsims.append(x)
                listofsims.append(y)
    return listofsims


def jaccsimaddresslist(a, b):
    listofsims=[]
    for x in a['address']:
        for y in b['address']:
            if (0.95<jaccsim(x,y)<= 1):
                listofsims.append(x)
                listofsims.append(y)
    return listofsims

def setofnamepairs(df_a,df_b):
    listofmatches=jaccsimnamelist(df_a, df_b)
    setofids=set()
    for x in range(len(listofmatches)):
        if x % 2==0:
            for a in range(len(df_a['name'])):
                if (listofmatches[x]==df_a['name'][a]):
                    for b in range(len(df_b['name'])):
                        if(listofmatches[x+1]==df_b['name'][b]):
                            setofids.add((df_a['id'][a],df_b['id'][b]))
    return(setofids)

def setofaddresspairs(df_a,df_b):
    listofmatches=jaccsimaddresslist(df_a, df_b)
    setofids=set()
    for x in range(len(listofmatches)):
        if x % 2==0:
            for a in range(len(df_a['address'])):
                if (listofmatches[x]==df_a['address'][a]):
                    for b in range(len(df_b['address'])):
                        if(listofmatches[x+1]==df_b['address'][b]):
                            setofids.add((df_a['id'][a],df_b['id'][b]))
    return(setofids)



our_match = setofnamepairs(df_a,df_b)|setofaddresspairs(df_a,df_b)


print("Precision: ", precision(our_match))
print("Recall: ", recall(our_match))
print("F1 score: ", f1_score(our_match))
```

```
Precision:  0.9076923076923077
Recall:  0.9672131147540983
F1 score:  0.9365079365079365
```

First, we remade the jaccard similarity function into two different fucntions, one to account for the names and one to account for the addresses. For both these new functions, titled jaccsimnamelist and jaccsimaddresslist respectively, we made the range of jaccard similarity to include 1 so as to account for exact matches. Then we remade the functions that created sets of tuples in order to account for the name and address of the restaurant separately. Then we unioned these two sets of tuples. We used union in order to account for both the address and name matches. We titled this union our_match and used the code given above to compute the precision, recall, and F1 score for our_matches.

Submitting HW 3

1. Double check that you have written all of your answers along with your supporting work in this notebook. Make sure you save the complete notebook.
2. Double check that your entire notebook runs correctly and generates the expected output. To do so, you can simply select Kernel -> Restart and Run All.
3. You will download two versions of your notebook to submit, a .pdf and a .py. To create a PDF, we reccomend that you select File --> Download as --> HTML (.html). Open the downloaded .html file; it should open in your web broser. Double check that it looks like your notebook, then print a .pdf using your web browser (you should be able to select to print to a pdf on most major web browsers and operating systems). Check your .pdf for readability: If some long cells are being cut off, go back to your notebook and split them into multiple smaller cells. To get the .py file from your notebook, simply select File -> Download as -> Python (.py) (note, we recognize that you may not have written any Python code for this assignment, but will continue the usual workflow for consistency).
4. Upload the .pdf to gradescope under hw3 report and the .py to gradescope under hw3 code. If you work with a partner, only submit one document for both of you, but be sure to add your partner using the group feature on gradescope (https://www.gradescope.com/help#help-center-item-student-group-members).

```
In [ ]:
```