Lab 3: Joining Multiple Tables (Python & Pandas)

In class, we've explored the power of tables as a data management abstraction, and we have already explored the Pandas DataFrame object in previous lab 2. Tables let us select rows and columns of interest, group data, and measure aggregates.

But what happens when we have more than one table? Traditional relational databases usually contain many tables. Moreover, when integrating multiple data sets, we necessarily need tools to combine them.

In this lab, we will use Panda's take on the database join operation to see how tables can be linked together. Specifically, we're going to perform a "fuzzy join" based on string edit-distance as another approach to finding duplicate records.

Setup

This notebook

Download this notebook so you can edit it. (If you are viewing it via nbviewer.ipython.org, then use the link in the upper right corner.) To edit this notebook, in your VM terminal, type "ipython notebook" and in your prompted brower, click the notebook file to open and edit it.

Data

We'll be using a small data set of restaurants. Download the data from here (https://ufl.instructure.com/courses/331578/files/folder/Lab%203). Put the data file, "restaurants.csv", in the same directory as this notebook.

Edit Distance

We're going to be using a string-similarity python library to compute "edit distance".

To test that it works, the following should run OK:

In [1]:
```
import Levenshtein as L
```

More Operations on DataFrame

In previous lab, we have learned some basic operation on Pandas DataFrame, including row filtering (selection), column filtering (projection) and groupby.

In this lab we first introduce some additional operation.

Aggregate

You should probably know to do aggregates: Pandas use python's lambda functions to define the function, and 'apply' to use the function.

In [2]:
```
import pandas as pd
df = pd.DataFrame( { 'a' : [1, 2, 1, 2], 'b': [ 'x', 'x', 'y', 'y'] })
df
```

Out[2]:

	a	b
0	1	x
1	2	x
2	1	y
3	2	y

```
In [3]: df['value'] = df.apply(lambda row: str(row.a) + ' + ' + row.b, axis=1)
        df
```

Out[3]:

	a	b	value
0	1	x	1 + x
1	2	x	2 + x
2	1	y	1 + y
3	2	y	2 + y

Pivot

Suppose we want to reshape the above DataFrame, so that in the new DataFrame we can have an index of different values in 'a' column to identify rows, and unique values in 'b' as column names. We can use **pivot** function to achieve this:

```
In [4]: new_df = df.pivot(index='a', columns='b', values='value')
        new_df
```

Out[4]:

b	x	y
a		
1	1 + x	1 + y
2	2 + x	2 + y

Joins

A **join** is a way to connect rows in two different data tables based on some criteria. Suppose the university has a database for student records with two tables in it: Students and Grades.

```
In [5]: Students = pd.DataFrame({'student_id': [1, 2], 'name': ['Alice', 'Bob']})
        Students
```

Out[5]:

	name	student_id
0	Alice	1
1	Bob	2

```
In [6]: Grades = pd.DataFrame({'student_id': [1, 1, 2, 2], 'class_id': [1, 2, 1, 3], 'grade': ['A', 'C', 'B', 'B
        ']})
        Grades
```

Out[6]:

	class_id	grade	student_id
0	1	A	1
1	2	C	1
2	1	B	2
3	3	B	2

Let's say we want to know all of Bob's grades. Then, we can look up Bob's student ID in the Students table, and with the ID, look up his grades in the Grades table. Joins naturally express this process: when two tables share a common type of column (student ID in this case), we can join the tables together to get a complete view.

In Pandas, we can use the **merge** method to perform a join. Pass the two tables to join as the first arguments, then the "on" parameter is set to the join column name.

In [7]:
```
pd.merge(Students, Grades, on='student_id')
```

Out[7]:

	name	student_id	class_id	grade
0	Alice	1	1	A
1	Alice	1	2	C
2	Bob	2	1	B
3	Bob	2	3	B

Question 1

Use merge to join Grades with the Classes table below, and find out what class Alice got an A in.

In [8]:
```
Classes = pd.DataFrame({'class_id': [1, 2, 3], 'title': ['Math', 'English', 'Spanish']})
# Your answer here:

mergedCG = pd.merge(Classes, Grades)
mergedCGS = pd.merge(Students, mergedCG, on = 'student_id')
mergedCGS[mergedCGS['name'] == 'Alice'][mergedCGS['grade'] == 'A']["title"]
```

```
/Users/abhisheknigam/anaconda/lib/python3.5/site-packages/ipykernel/__main__.py:6: UserWarning: Boolean
Series key will be reindexed to match DataFrame index.
```

Out[8]:
```
0    Math
Name: title, dtype: object
```

Question 2

Suppose we want to get students' grades for each class in a more straightforward way, and decide to get a Dataframe that:

1. has one row for each student name,
2. different class titles as columns names,
3. the grades of students in different classes as the values (NaN if not applicable).

how do we get such a DataFrame?

In [9]:
```
# Your answer here:
modifiedCGS = mergedCGS.pivot(index='name', columns='title', values='grade')
modifiedCGS = modifiedCGS.replace([None],"NaN")
modifiedCGS
```

Out[9]:

title	English	Math	Spanish
name			
Alice	C	A	NaN
Bob	NaN	B	B

Joining the Restaurant Data

Now let's load the restaurant data that we will be analyzing:

In [10]:
```
resto = pd.read_csv('restaurants.csv')
resto.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 858 entries, 0 to 857
Data columns (total 4 columns):
id         858 non-null int64
cluster    858 non-null int64
name       858 non-null object
city       858 non-null object
dtypes: int64(2), object(2)
memory usage: 26.9+ KB
```

In [11]:
```
resto[:10]
```

Out[11]:

	id	cluster	name	city
0	560	453	2223	san francisco
1	781	675	103 west	atlanta
2	279	172	20 mott	new york
3	43	23	21 club	new york
4	44	23	21 club	new york city
5	280	173	9 jones street	new york
6	486	379	abbey	atlanta
7	145	74	abruzzi	atlanta
8	146	74	abruzzi	atlanta
9	561	454	acquarello	san francisco

The restaurant data has four columns. **id** is a unique ID field (unique for each row), **name** is the name of the restaurant, and **city** is where it is located. The **name** and **city** columns are noisy and not very accurate.

However the **cluster** column is a "gold standard" column. If two records have the same cluster, that means they are both about the same restaurant.

The type of join we made above between Students and Grades, where we link records with equal values in a common column, is called an **equijoin**. Equijoins may join on more than one column, too (both values have to match).

Let's use an equijoin to find pairs of duplicate restaurant records. We join the data to itself, on the cluster column.

The result ("clusters" below) has a lot of extra records in it. For example, since we're joining a table to itself, every record matches itself. We can filter on IDs to get rid of these extra join results. Note that when Pandas joins two tables that have columns with the same name, it appends "_x" and "_y" to the names to distinguish them.

In [12]:
```
clusters = pd.merge(resto, resto, on='cluster')
clusters = clusters[clusters.id_x != clusters.id_y]
clusters[:10]
```

Out[12]:

	id_x	cluster	name_x	city_x	id_y	name_y	city_y
4	43	23	21 club	new york	44	21 club	new york city
5	44	23	21 club	new york city	43	21 club	new york
10	145	74	abruzzi	atlanta	146	abruzzi	atlanta
11	146	74	abruzzi	atlanta	145	abruzzi	atlanta
20	184	94	alain rondelli	san francisco	185	alain rondelli	san francisco
21	185	94	alain rondelli	san francisco	184	alain rondelli	san francisco
36	186	95	aqua	san francisco	187	aqua	san francisco
37	187	95	aqua	san francisco	186	aqua	san francisco
40	45	24	aquavit	new york	46	aquavit	new york city
41	46	24	aquavit	new york city	45	aquavit	new york

There are still redundant records in 'clusters' above: if records A and B match each other, then we will get both (A, B) and (B, A) in the output, this is redundant since we don't care about the order of A and B.

Filter clusters so that we only keep one instance of each matching pair (HINT: use the IDs again).

In [13]:
```
clusters = clusters[clusters.id_x > clusters.id_y]
clusters[:10]
```

Out[13]:

	id_x	cluster	name_x	city_x	id_y	name_y	city_y
5	44	23	21 club	new york city	43	21 club	new york
11	146	74	abruzzi	atlanta	145	abruzzi	atlanta
21	185	94	alain rondelli	san francisco	184	alain rondelli	san francisco
37	187	95	aqua	san francisco	186	aqua	san francisco
41	46	24	aquavit	new york city	45	aquavit	new york
47	2	0	arnie morton's of chicago	los angeles	1	arnie morton's of chicago	los angeles
50	4	1	art's deli	studio city	3	art's delicatessen	studio city
59	48	25	aureole	new york city	47	aureole	new york
63	148	75	bacchanalia	atlanta	147	bacchanalia	atlanta
78	6	2	bel-air hotel	bel air	5	hotel bel-air	bel air

Fuzzy Joins

Sometimes an equijoin isn't good enough.

Say you want to match up records that are almost equal in a column. Or where a function of a columns is equal. Or maybe you don't care about equality: maybe "less than" or "greater than or equal to" is what you want. These cases call for a more general join than equijoin.

We are going to make one of these joins between the restaurants data and itself. Specifically, we want to match up pairs of records whose restaurant names are almost the same. We call this a **fuzzy join**.

To do a fuzzy join in Pandas we need to go about it in a few steps:

1. Join every record in the first table with every record in the second table. This is called the **Cartesian product** of the tables, and it's simply a list of all possible pairs of records.
2. Add a column to the Cartesian product that measures how "similar" each pair of records is. This is our **join criterion**.
3. Filter the Cartesian product based on when the join criterion is "similar enough."

Let's do an example to get the hang of it.

1. Join every record in the first table with every record in the second table.

We use a "dummy" column to compute the Cartesian product of the data with itself. dummy takes the same value for every record, so we can do an equijoin and get back all pairs.

In [14]:
```
resto['dummy'] = 0
prod = pd.merge(resto, resto, on='dummy')

# Clean up
del prod['dummy']
del resto['dummy']

# Show that prod is the size of "resto" squared:
print (len(prod), len(resto)**2)
```

```
736164 736164
```

```
In [15]: prod[:10]
```

Out[15]:

	id_x	cluster_x	name_x	city_x	id_y	cluster_y	name_y	city_y
0	560	453	2223	san francisco	560	453	2223	san francisco
1	560	453	2223	san francisco	781	675	103 west	atlanta
2	560	453	2223	san francisco	279	172	20 mott	new york
3	560	453	2223	san francisco	43	23	21 club	new york
4	560	453	2223	san francisco	44	23	21 club	new york city
5	560	453	2223	san francisco	280	173	9 jones street	new york
6	560	453	2223	san francisco	486	379	abbey	atlanta
7	560	453	2223	san francisco	145	74	abruzzi	atlanta
8	560	453	2223	san francisco	146	74	abruzzi	atlanta
9	560	453	2223	san francisco	561	454	acquarello	san francisco

Question 3

Remove pairs with the same IDs and redundant pairs in 'prod'.

```
In [16]: # Your answer here:
         prod = prod[prod.id_x != prod.id_y]
         prod = prod[prod.id_x > prod.id_y]

         len(prod)

Out[16]: 367653
```

2. Add a column to the Cartesian product that measures how "similar" each pair of records is.

We're going to use a measure of string similarity called **edit-distance**. Edit-distance counts the number of simple changes you have to make to a string to turn it into another string.

Import the edit distance library:

```
In [17]: import Levenshtein as L

         L.distance('Hello, World!', 'Hallo, World!')

Out[17]: 1
```

Next, we add a computed column, named **distance**, that measures the edit distance between the names of two restaurants:

```
In [18]: # This might take a minute or two to run
         prod['distance'] = prod.apply(lambda r: L.distance(r['name_x'], r['name_y']), axis=1)
```

3. Filter the Cartesian product based on when the join criterion is "similar enough."

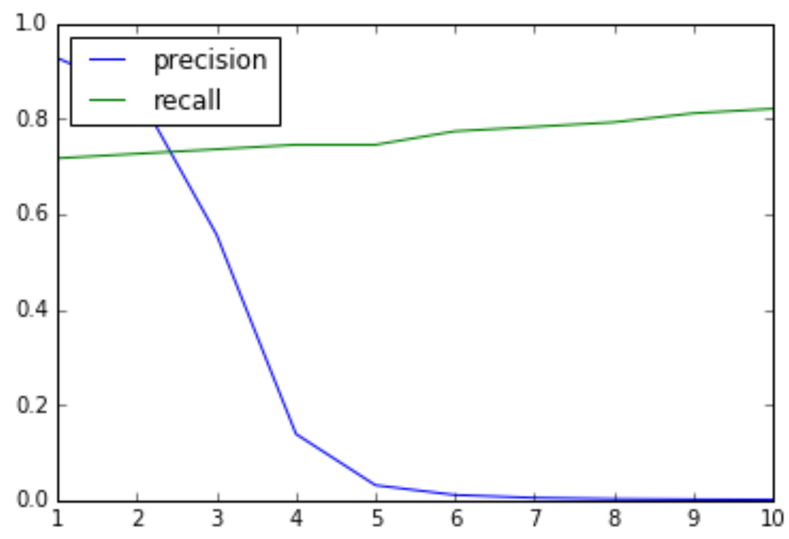
Now we complete the join by filtering out pairs or records that aren't equal enough for our liking. We can only figure out how similar is "similar enough" by trying out some different options.

Let's try maximum edit-distance from 0 to 10 and compute precision and recall (https://www.quora.com/What-is-the-best-way-to-understand-the-terms-precision-and-recall/answer/Joel-Chan?srid=nJbo).

Remember that we are fuzzy-joining based on inaccurate **name** column, thus pairs that we deem similar may not be actually about the same restaurant, and we can use the 'golden standard' column **cluster** to determine if our prediction based on edit-distance of name columns are right or not.

In [19]:
```
%matplotlib inline
import pylab

def accuracy(max_distance):
    similar = prod[prod.distance < max_distance]
    correct = float(sum(similar.cluster_x == similar.cluster_y))
    precision = correct / len(similar)
    recall = correct / len(clusters)
    return (precision, recall)

thresholds = range(1, 11)
p = []
r = []

for t in thresholds:
    acc = accuracy(t)
    p.append(acc[0])
    r.append(acc[1])

pylab.plot(thresholds, p)
pylab.plot(thresholds, r)
pylab.legend(['precision', 'recall'], loc='upper left')
```

Out[19]:
```
<matplotlib.legend.Legend at 0x112fd8e80>
```

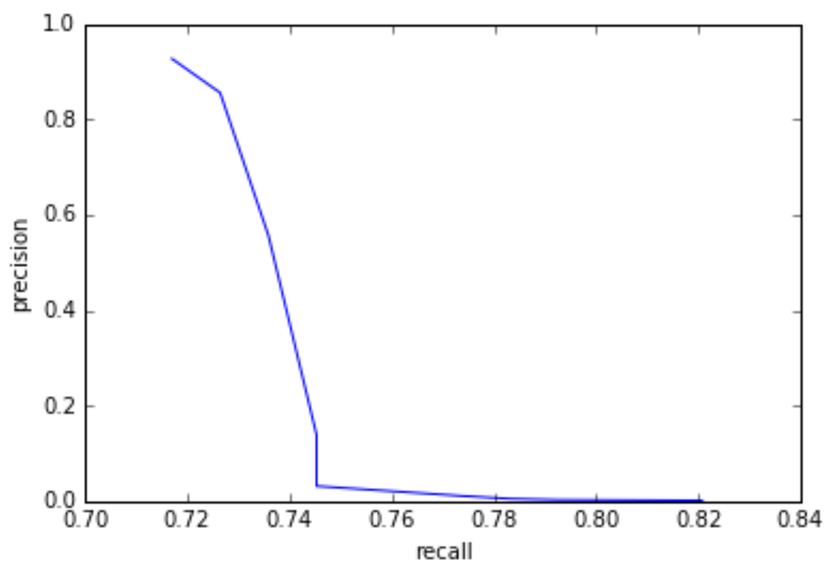


Question 4

Another common way to visualize the tradeoff between precision and recall is to plot them directly against each other. Create a line plot with precision on one axis and recall on the other (this graph generated is called precision-recall curve (https://www.quora.com/What-is-Precision-Recall-PR-curve)). Where are "good" points on the plot, and where are "bad" ones.(use recall as horizontal axis and precision as vertical axis)

```
In [20]: # Your answer here:

         ### The good points in the graph would depend on the requirement of precision and recall.
         #An algorithm with a high precision and recall is preferred. In the following graph the points
         #in the middle of the graph have a high precision and recall value. As we move
         #towards the edges then the value of either recall or precision starts to dip. So we can
         #consider for this case that the points in the middle of the line graph will be
         # the good values where precision and recall are comparitive.

         %matplotlib inline
         import pylab
         import matplotlib.pyplot as pp

         def accuracy(max_distance):
             similar = prod[prod.distance < max_distance]
             correct = float(sum(similar.cluster_x == similar.cluster_y))
             precision = correct / len(similar)
             recall = correct / len(clusters)
             return (precision, recall)

         thresholds = range(1, 11)
         p = []
         r = []

         for t in thresholds:
             acc = accuracy(t)
             p.append(acc[0])
             r.append(acc[1])

         pylab.plot(r,p)
         pylab.xlabel("recall")
         pylab.ylabel("precision")
```

```
Out[20]: <matplotlib.text.Text at 0x113356518>
```

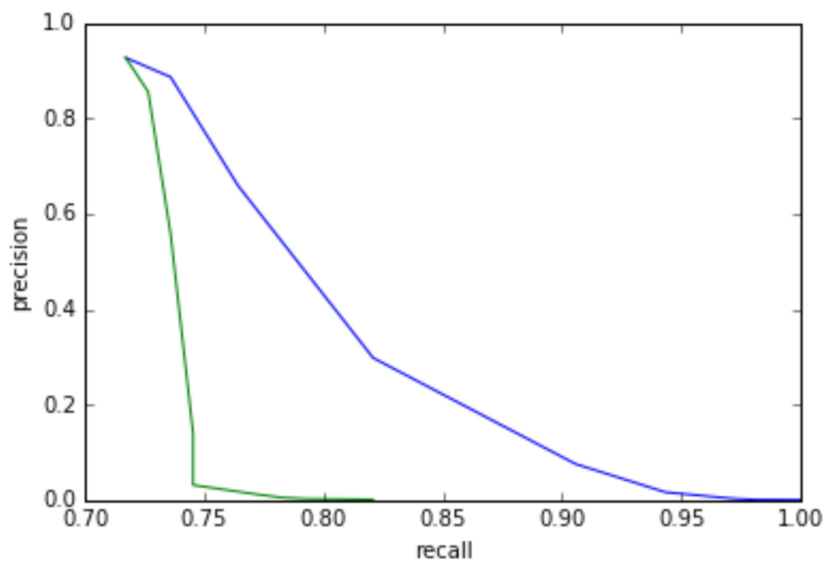


Question 5

The python Levenshtein library provides another metric of string similarity called "ratio" (use L.ratio(s1, s1)). ratio gives a similarity score between 0 and 1, with **higher meaning more similar**. Add a column to "prod" with the ratio similarities of the name columns, and redo the precision/recall tradeoff analysis with the new metric. (Note: you will have to alter the accuracy method and the threshold range.) On this data, does Levenshtein.ratio do better than Levenshtein.distance? (Plot the two precision-recall curves together in one graph to compare them)

In [21]:
```
#Your answer here
#The graph of Levenshtein.ratio shows a consistency between precision and recall which
#is better than the Levenshtein.distance. Here the precision and recall have a better
#ratio as compared to the precision vs recall graph in the previous question. But based
#on the requirement of hight precision at the cost of recall or high recall at the cost
#of precision, any of the graphs can be choosen.

prod['ratio'] = prod.apply(lambda r: L.ratio(r['name_x'], r['name_y']), axis=1)

%matplotlib inline
import pylab
import numpy as np
import matplotlib.pyplot as pp

def accuracy_ratio(max_ratio):
    similar = prod[prod.ratio >= max_ratio]
    correct = float(sum(similar.cluster_x == similar.cluster_y))
    precision = correct / len(similar)
    recall = correct / len(clusters)
    return (precision, recall)

thresholds_ratio = np.arange(0.1,1.1,0.1)
p_ratio = []
r_ratio = []

for t in thresholds_ratio:
    acc_ratio = accuracy_ratio(t)
    p_ratio.append(acc_ratio[0])
    r_ratio.append(acc_ratio[1])

pylab.plot(r_ratio,p_ratio)
pylab.plot(r,p)
pylab.xlabel("recall")
pylab.ylabel("precision")
```

Out[21]:
```
<matplotlib.text.Text at 0x1134482e8>
```



Homework

Please answer the questions 1 - 5 inlined in this ipython notebook, and submit a pdf document containing your answers and graphs generated (if applicable).

You can just do a screenshot of the graph generated and copy to a google doc along with your code, no need to import additional libs for generating pdf from ipython notebook.