IDS-F15 (/github/xfzhou/IDS-F15/tree/master) / lab1 (/github/xfzhou/IDS-F15/tree/master/lab1)

Joining DataFrames in Pandas

In previous labs, we've explored the power tables as a data management abstraction, in particular with the Pandas DataFrame object. 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.

Remember to fill out the DIYs here (https://ufl.instructure.com/courses/320501/quizzes/463144/) at the end when you are ready!

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

Today we'll be using a small data set of restaurants. Download the data from here (https://ufl.instructure.com/courses/320501/files/folder/Lab1). 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 [ ]:
```
import Levenshtein as L
```

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 [ ]:
```
import pandas as pd

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

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

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

DIY

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

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

Joining the Restaurant Data

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

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

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

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 fourth column, **cluster**, 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 value 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.

> Note: a join between a table and itself is called a *self-join*.

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

DIY

2. There are still extra 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. Filter *clusters* so that we only keep one instance of each matching pair (HINT: use the IDs again).

In [ ]:
```

```

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."

> SQL Aside: In SQL, all of joins are supported in about the same way as equijoins are. Essentially, you write a boolean expression on columns from the join-tables, and whenever that expression is true, you join the records together. This is very similar to writing an **if** statement in python or Java.

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 [ ]:

```
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
```

In [ ]:

```
prod[:10]
```

DIY

3. Like we did with *clusters* remove "extra" record pairs in DIY 2, e.g., ones with the same IDs.

In [ ]:

```

```

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

In the homework assignment, we used a string similarity metric called *cosine similarity* which measured how many "tokens" two strings shared in common. Now, we're going to use an alternative measure of string similarity called *edit-distance*. Edit-distance (http://en.wikipedia.org/wiki/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 [ ]:

```
import Levenshtein as L

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

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

In [ ]:

```
# This takes 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. As in the first homework assignment, 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.

In [ ]:

```
%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')
```

DIY

4. Another common way to visualize the tradeoff between precision and recall is to plot them directly against each other. Create a scatterplot with precision on one axis and recall on the other (this graph generated is called precision-recall curve). Where are "good" points on the plot, and where are "bad" ones.(use recall as horizontal axis and precision as vertical axis)

In [ ]:

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 [ ]:

```

```

Finally, remember to fill out the DIYs 1-5 here (https://ufl.instructure.com/courses/320501/quizzes/463144/) when you are ready!