Website (https://www.coleridgeinitiative.org)

Rayid Ghani, Frauke Kreuter, Julia Lane, Adrianne Bradford, Alex Engler, Nicolas Guetta Jeanrenaud, Graham Henke, Daniela Hochfellner, Clayton Hunter, Brian Kim, Avishek Kumar, Jonathan Morgan, Ridhima Sodhi, and Benjamin Feder.

source to be updated when notebook added to GitHub

Record Linkage

Table of Contents

JupyterLab contains a dynamic Table of Contents that can be accessed by clicking the last of the six icons on the left-hand sidebar.

Introduction

This notebook will provide you with an introduction to record linkage using Python. Upon completion of this notebook, you will be able to apply record linkage techniques using the `recordlinkage` package to combine data from different sources in python. You will go through all the steps necessary for a successful record linkage starting with data preparation, which includes pre-processing, cleaning and standardizing data.

Learning Objectives

We will explore different record linkage techniques in order to match the `tip_2017` table of WIC-approved vendors with `store_info_all` , which contains general vendor information.

The Principles of Record Linkage

The goal of record linkage is to determine if pairs of records describe the same identity. For instance, this is important for removing duplicates from a data source or joining two separate data sources together. Record linkage also goes by the terms data matching, merge/purge, duplication detection, de-duping, reference matching, entity resolution, disambiguation, co-reference/anaphora in various fields.

There are several approaches to record linkage that include:

```
- exact matching
- rule-based linking
- probabilistic linking
```

- An example of **exact matching** is joining records based on social security number, exact name, or geographic code information. You have already have done this in SQL when joining tables on an unique identifier.
- **Rule-based matching** involves applying a cascading set of rules that reflect the domain knowledge of the records being linked.
- In **probabilistic record linkages**, linkage weights are estimated to calculate the probability of a certain match.

In practical applications, you will need record linkage techniques to combine information addressing the same entity that is stored in different data sources. Record linkage will also help you to address the quality of varying data sources. For example, if one of your databases has missing values, you might be able to fill those by finding an identical pair in a different data source. Overall, the main applications of record linkage are:

```
1. Merging two or more data files
2. Identifying the intersection of the two data sets
3. Updating data files (with the data row of the other data files) and imput
ing missing data
4. Entity disambiguation and de-duplication
```

Analytical Approach

For this notebook exercise, we are interested in vendor data.

- **Analytical Exercise**: Find WIC-approved stores in the vendor data supplied by IRI.
- **Data Availability**: We have names, addresses, and years for the various vendors in the two tables.
- **Approach**: We will look at the data available to us and clean & pre-process it to enable better linkage. Afterwards, we will use string matching techniques that are enabled by record linkage package in Python.

Access the Data

Python provides us with some tools we can use for record linkages so we don't have to start from scratch and code our own linkage algorithms. Before we start, we need to load the package `recordlinkage`. We will be adding a few more packages than usual to our import process because the `recordlinkage` package has a few dependencies on other packages.

In [ ]:
```
# general use imports
%pylab inline
import datetime
import numpy as np
import re

# pandas-related imports
import pandas as pd

# record linkage package
import recordlinkage as rl
from recordlinkage.preprocessing import clean

# database interaction imports
from pyathenajdbc import connect

print( "Imports loaded at " + str( datetime.datetime.now() ) )
```

In [ ]:
```
conn = connect(s3_staging_dir = 's3://usda-iri-2019-queryresults/',
               region_name = 'us-gov-west-1',
               LogLevel = '0',
               workgroup = 'workgroup-iri_usda')
```

Data Exploration

In our notebooks thus far, we have not utilized either `tip_2017` or `store_info_all`. To see what data manipulations we may have to perform, let's take a quick look at the two tables.

In [ ]:
```
sql = '''
select *
from iri_usda.store_info_all
where year = '2017'
'''

df_total = pd.read_sql(sql, conn)
```

In [ ]:
```
df_total.head()
```

In [ ]:
```
df_total.describe()
```

In [ ]:
```
# null values by column
df_total.isnull().sum()
# Output: count of null values for all of the columns
```

In [ ]:
```
df_total.isin(['', '.']).sum()
```

Although there are null values for some columns in `store_info_all` in 2017, they are not in any of the variables that we will be using for the record linkage. Therefore, we will not have to do any manipulation with null values for `store_info_all`. Let's see if we have any extra work for `tip_2017`.

In [ ]:
```
sql = '''
select *
from iri_usda.tip_2017
'''

df_approved = pd.read_sql(sql, conn)
```

In [ ]:
```
df_approved.head()
```

In [ ]:
```
df_approved.describe()
```

In [ ]:
```
# Output: count of null values for all of the columns
df_approved.isnull().sum()
```

In [ ]:
```
df_approved.isin(['', '.']).sum()
```

Unfortunately, we were not as lucky for `df_approved`. Let's see what's wrong, i.e. when `vendor_street_number` and/or `vendor_street` are missing.

In [ ]:
```
# when street number is missing
qry = '''
select *
from iri_usda.tip_2017
where vendor_street_number = ''
limit 5
'''

pd.read_sql(qry, conn)
```

In [ ]:
```
# count when all three street identifiers are missing
qry = '''
select count(*)
from iri_usda.tip_2017
where vendor_street_number = '' and vendor_street = '' and vendor_additi
onal_address = ''
'''

pd.read_sql(qry, conn)
```

Since `store_info_all` only has one column for the street number and address, we will have to concatenate `vendor_street_number` and `vendor_street`. However, we can also see that when `vendor_street_number` and `vendor_street` are null, there is a value for `vendor_additional_address`. Therefore, we will overwrite `df_approved` with a SQL query that concatenates `vendor_street_number` and `vendor_street` when they exist, and otherwise uses `vendor_additional_address`.

> In practice, it would be best to separate the street number and name into two separate categories as was described in the lecture. However, that would require additional code and since the purpose of this notebook is to get familiarized with various record linkage techniques, that process will not be covered in this notebook.

In [ ]:
```
# rewrite df_approved with only columns needed for linkage
sql = '''
select vendor_name,
    case when vendor_street_number = '' and vendor_street = '' then vend
or_additional_address
    else concat(vendor_street_number, ' ',vendor_street)
    end as address,
vendor_city, vendor_state, vendor_zip
from iri_usda.tip_2017
'''

df_approved = pd.read_sql(sql, conn)
```

In [ ]:
```
#confirm df_approved is what we want
df_approved.head()
```

The Importance of Pre-Processing

Data pre-processing is an important step in a data analysis project in general, and in record linkage applications, it is particularly crucial. The goal of pre-processing is to transform messy data into a dataset that can be used in a project workflow.

Linking records from different data sources comes with different challenges that need to be addressed by the analyst. The analyst must determine whether or not two entities (individuals, businesses, geographical units) from two different files are the same. This determination is not always easy. In most of the cases there is no common uniquely identifing characteristic for a entity. For example, is Bob Miller from New York the same person as Bob Miller from Chicago in a given dataset? This determination has to be executed carefully because consequences of wrong linkages may be substantial (i.e. Is person X the same person as the person X on the list of identified terrorists?). Pre-processing can help to make better informed decisions.

Pre-processing can be difficult because there are a lot of things to keep in mind. For example, data input errors, such as typos, misspellings, truncation, abbreviations, and missing values need to be corrected. Literature shows that pre-processing can improve matches. In some situations, 90% of the improvement in matching efficiency may be due to pre-processing. The most common reason why matching projects fail is the lack of time and resources for data cleaning.

In the following section, we will walk you through some pre-processing steps. Here we will touch upon practices that include but are not limited to removing spaces, parsing fields, and standardizing strings.

Let's look at the the most recurring store names in the two tables.

In [ ]:
```
df_total['store_name'].value_counts()
```

In [ ]:
```
df_approved['vendor_name'].value_counts()
```

Right away, we notice that the record linkage between the different datasets will not be straightforward. The variable is messy and non-standardized, similar names can be written differently (in upper-case or lower-case characters, with or without suffixes, etc.) The essential next step is to process the variables in order to make the linkage the most effective and relevant possible.

Parsing String Variables

By default, the split method returns a list of strings obtained by splitting the original string on spaces or commas, etc. The record linkage package comes with a build in cleaning function we can also use. In addition, we can extract information from strings for example by using regex search commands.

In [ ]:
```
# Uppercasing names and creating a new column of names to work with
df_total['store_name_clean']=df_total.store_name.str.upper()
df_total['address_clean'] = df_total.address.str.upper()
df_total['city_clean'] = df_total.city.str.upper()
df_total['state_clean'] = df_total.state.str.upper()

# Do same to df_approved
df_approved['vendor_name_clean'] = df_approved.vendor_name.str.upper()
df_approved['address_clean'] = df_approved.address.str.upper()
df_approved['vendor_city_clean'] = df_approved.vendor_city.str.upper()
df_approved['vendor_state_clean'] = df_approved.vendor_state.str.upper()

# see first five entries
df_approved.head()
```

In [ ]:
```
# Cleaning names (using the record linkage package tool, see imports)
# Clean removes any characters such as '-', '.', '/', '\', ':', brackets
of all types.
df_total['store_name_clean']=clean(df_total['store_name_clean'], lowerca
se=False, strip_accents='ascii', \
                                   remove_brackets=False)
df_total['address_clean']=clean(df_total['address_clean'], lowercase=Fal
se, strip_accents='ascii', \
                                   remove_brackets=False)
df_total['city_clean']=clean(df_total['city_clean'], lowercase=False, st
rip_accents='ascii', \
                                   remove_brackets=False)
df_total['state_clean']=clean(df_total['state_clean'], lowercase=False,
strip_accents='ascii', \
                                   remove_brackets=False)

# Do same for df_approved
df_approved['vendor_name_clean']=clean(df_approved['vendor_name_clean'],
lowercase=False, strip_accents='ascii', \
                                   remove_brackets=False)
df_approved['address_clean']=clean(df_approved['address_clean'], lowerca
se=False, strip_accents='ascii', \
                                   remove_brackets=False)
df_approved['vendor_city_clean']=clean(df_approved['vendor_city_clean'],
lowercase=False, strip_accents='ascii', \
                                   remove_brackets=False)
df_approved['vendor_state_clean']=clean(df_approved['vendor_state_clean'
], lowercase=False, strip_accents='ascii', \
                                   remove_brackets=False)

df_total.head()
df_approved.head()
```

Regular Expressions – `regex`

Regular expressions (regex) are a way of searching for a character pattern. They can be used for matching or replacing operations in strings.

When defining a regular expression search pattern, it is a good idea to start out by writing down, explicitly, in plain English, what you are trying to search for and exactly how you identify when you've found a match. For example, if we look at an author field formatted as "<last_name> , <first_name> <middle_name>", in plain English, this is how I would explain where to find the last name: "starting from the beginning of the line, take all the characters until you see a comma."

In a regular expression, there are special reserved characters and character classes. For example:

- " ^ " matches the beginning of the line or cell
- " `.` " matches any character
- " + " means one or more repetitions of the preceding expressions

Anything that is not a special character or class is just looked for explicitly. A comma, for example, is not a special character in regular expressions, so inserting " `,` " in a regular expression will simply match that character in the string.

In our example, in order to extract the last name, the resulting regular expression would be: " `^.+,` ". We start at the beginning of the line (" ^ "), matching any characters (" `.`+ ") until we come to the literal character of a comma (" `,` ").

> *If you want to actually look for one of these reserved characters, it must be escaped. For example, if the expression looks for a literal period, rather than the special regular expression meaning of a period, precede it with a back slash (" \ ") to escape the reserved character in a regular expression. For example, " `\.` " will match a " `.` " character in a string.*

REGEX CHEATSHEET

- abc... Letters
- 123... Digits
- \d Any Digit
- \D Any non-Digit Character
- . Any Character
- \. Period
- [a,b,c] Only a, b or c
- [^a,b,c] Not a,b, or c
- [a-z] Characters a to z
- [0-9] Numbers 0 to 9
- \w any Alphanumeric chracter
- \W any non-Alphanumeric character
- {m} m Repetitions
- {m,n} m to n repetitions
- * Zero or more repetitions
- + One or more repetitions
- ? Optional Character
- \s any Whitespace
- \S any non-Whitespace character
- ^...$ Starts & Ends
- (...) Capture Group
- (a(bc)) Capture sub-Group
- (.*) Capture All
- (abc|def) Capture abc or def

Examples:

- `(\d\d|\D)` will match 22X, 23G, 56H, etc...
- `(\w)` will match any characters between 0-9 or a-z
- `(\w{1-3})` will match any alphanumeric character of a length of 1 to 3.
- `(spell|spells)` will match spell or spells
- `(corpo?) will match corp or corpo
- `(feb 2.)` will match feb 20, feb 21, feb 2a, etc.

Using REGEX to match characters:

In python, to use a regular expression to search for matches in a given string, we use the built-in " `re` " package (https://docs.python.org/2/library/re.html (https://docs.python.org/2/library/re.html)), specifically the " `re.search()` " method. To use " `re.search()` ", pass it the regular expression you want to use to search enclosed in quotation marks, and then the string you want to search within.

Using REGEX for replacing characters:

The `re` package also has an " `re.sub()` " method used to replace regular expressions by other strings. The method can be applied to an entire pandas column (replacing expression1 with expression2) with the following syntax: `df['variable'].str.replace(r'expression1', 'expression2')`. Note the `r` before the first string to signal we are using regular expressions.

In this notebook, we will not have to use regex too much to pre-process our data tables, but in general, knowing regex is essential for cleaning data. Here, we will show you how you can use regex to extract everything inside quotation marks from addresses in `df_approved`.

> The quotation marks were already removed in the previous steps, but this example shows how you can do the same process using regular expressions.

In [ ]:
```
# Extracting address inside quotations in df_approved

# Pattern1
df_approved['address'].str.extract('"(.*)"')

# Breaking the code down:
# .*? ---- tells that we need any character 0 or more times after the fi
rst quotation mark
# () enclosing brackets tell that we need to extract this information in
the new variable
# "" we need to find everything inside of the quotes
```

Do you see any other possible standardizations? Insert them below!

In [ ]:
```

```

Now we are done with the inital data prep work. Please keep in mind that we just provided some examples for you to demonstrate the process. You can add as many further steps to it as necessary.

Record Linkage

The record linkage package is a quite powerful tool for you to use when you want to link records within a dataset or across multiple datasets. It comes with different bulid in distances metrics and comparison functions, however, it also allows you to create your own. In general record linkage is divided in several steps.

In [ ]:
```
# Only keep variables relevant for linkage
df_total = df_total[['store_name_clean', 'zipcode', 'address_clean', 'ci
ty_clean', 'state_clean']]
df_approved = df_approved[['vendor_name_clean', 'vendor_zip', 'address_c
lean', 'vendor_city_clean',
                           'vendor_state_clean']]

#rename df_approved to match df_total for simplicity sake
df_approved = df_approved.rename({'vendor_name_clean':'store_name_clean'
, 'vendor_zip':'zipcode',
                                  'vendor_city_clean':'city_clean','vend
or_state_clean':'state_clean'},
                                 axis = 'columns')
```

In [ ]:
```
df_total.head()
```

In [ ]:
```
df_approved.head()
```

We've already done the pre-processing, so the next step is indexing the data we would like to link. Indexing allows you to create candidate links, which basically means identifying pairs of data rows which might refer to the same real world entity. This is also called the comparison space (matrix). There are different ways to index data. The easiest is to create a full index and consider every pair a match. This is also the least efficient method, because we will be comparing every row of one dataset with every row of the other dataset. Because of how extensive this process is, we will demonstrate it on just stores in New Mexico to limit its runtime.

In [ ]:
```
# subset to just one state to demonstrate the `FullIndex()` method
nm_total = df_total[df_total['state_clean'] == 'NM']
nm_approved = df_approved[df_approved['state_clean'] == 'NM']
```

In [ ]:
```
# Let's generate a full index first (comparison table of all possible li
nkage combinations)
indexer = rl.index.Full()
pairs = indexer.index(nm_total, nm_approved)
# Returns a pandas MultiIndex object
print(len(pairs))
```

We can do better if we actually include our knowledge about the data to eliminate bad links from the start. This can be done through blocking. The `recordlinkage` package gives you multiple options for this. For example, you can block by using variables, which means that only links exactly equal on specified values will be kept. Here, we will block on `zipcode` so we only compare stores with the same zip code.

You can also use a neighborhood index in which the rows in your dataframe are ranked by some value and python will only link between the rows that are close by.

In [ ]:
```
#initialize indexer
indexer = rl.Index()

#block on zipcode
indexer.block('zipcode')

# Returns a pandas MultiIndex object
pairs2 = indexerBL.index(df_total, df_approved)
print(len(pairs2))
```

In [ ]:
```
# Initiate compare object (we are using the blocked ones here)
# You want to give python the name of the MultiIndex and the names of th
e datasets
compare = rl.Compare()
```

Now we have set up our comparison space. We can start to compare our files and see if we find matches. We will demonstrate an exact match and rule based approches using distance measures. Our goal is to create a dataframe for each record pair and if they are listed as a match using different linking methods. To do so, we will include the `label` argument to store the algorithms' outputs as different variables in our dataframe.

As for the different comparative measures we will cover, they include:

- Exact
- Levenshtein
- Jarowinkler

In [ ]:
```
# Exact comparison
# This compares all the pairs of strings for exact matches
# It is similar to a JOIN--
compare.exact('store_name_clean','store_name_clean', label = 'store_name
_clean')
```

In [ ]:
```
# This command gives us the probability of match between two strings bas
ed on the levenshtein distance
# The measure is 0 if there are no similarities in thee string, 1 if i
t's identical
compare.string('store_name_clean','store_name_clean', method='levenshtei
n', label = 'levenshtein_name')
```

In [ ]:
```
# This command gives us the probability of match between two strings bas
ed on the jarowinkler distance
# The measure is 0 if there are no similarities in thee string, 1 if i
t's identical
compare.string('store_name_clean','store_name_clean', method='jarowinkle
r', label = 'jarowinkler_name')
```

In [ ]:
```
#compute levenshtein distance for addresses
compare.string('address_clean', 'address_clean', method = 'levenshtein',
label = 'levenshtein_address')

#we want exact matches for city names
compare.exact('city_clean', 'city_clean', label='city_clean')
```

To actually compute the record pairs, we need to call the `compute` method. This will output exactly what we want to see: each potential pairing when blocked on `zipcode`, with corresponding values for our different comparative metrics we've used.

In [ ]:
```
# compute record pairing scores
features = compare.compute(pairs2, df_total, df_approved)
```

In [ ]:
```
features.head()
```

In [ ]:
```
features.describe()
```

Results

Once we have our comparison measures, we need to classify the measure in matches and non matches for non-exact pairings (Levenshtein and Jarowinkler). A rule-based approach would be to say if the similarity of our indicators is 0.85 or higher we consider this a match, everything else we won't match. This decision need to be made by the analyst. We're going to use .85 for all of our distance computations simply because it is considered to be a standard in the field.

> In practice, you should examine matches around the thresholds you are considering to make sure the threshold lines up with how you theoretically view a match. Different methods return different results, so ideally, you would want to enact different thresholds based on the different comparative algorithms you are employing.

In [ ]:
```
# Impose threshold of .85
features['levenshtein_name'] = [1 if v > .85 else 0 for v in features['l
evenshtein_name']]
features['jarowinkler_name'] = [1 if v > .85 else 0 for v in features['j
arowinkler_name']]
features['levenshtein_address'] = [1 if v > .85 else 0 for v in features
['levenshtein_address']]
```

In [ ]:
```
features.head()
```

Now we need to decide which stores are the same between the two tables. Let's see the comparison results by seeing the distribution of scores (0-5 for the counts of 1 for the five measures).

In [ ]:
```
features.sum(axis=1).value_counts().sort_index(ascending=False)
```

Arbitrarily, let's say that all comparisons with at least 3 metrics scored as a 1 are matches. Let's subset to just those matches.

In [ ]:
```
features[features.sum(axis=1) >= 3]
```

Another way of classiying records is the Fellegi Sunter Method. If Fellegi Sunter is used to classify record pairs, you would follow all the step we have done so far. However, now, we would estimate probabilities to construct weights. These weights will then be applied during the classification to give certain characteristics more importance. For example, we are more certain that very unique names are a match than Bob Millers.

Fellegi Sunter

In [ ]:
```
# let's assume the rows above are our matches
matches = features[features.sum(axis=1) >= 3]
```

In [ ]:
```
len(features[features.sum(axis=1) >= 3])
```

In [ ]:
```
## Generate Training Data and index
ml_pairs = matches[0:4000]
ml_matches_index = ml_pairs.index & pairs2
```

The Naive Bayes classifier is a probabilistic classifier. The probabilistic record linkage framework by Fellegi and Sunter (1969) is the most well-known probabilistic classification method for record linkage. Later, it was proved that the Fellegi and Sunter method is mathematically equivalent to the Naive Bayes method in case of assuming independence between comparison variables.

In [ ]:
```
## Train the classifier
nb = rl.NaiveBayesClassifier()
nb.fit(ml_pairs, ml_matches_index)

## Predict the match status for all record pairs
result_nb = nb.predict(matches)
```

In [ ]:
```
result_nb
```

Evaluation

The last step is to evaluate the results of the record linkage. We will cover this in more detail in the machine learning session. This is just for completeness.

In [ ]:
```
## Confusion matrix
conf_nb = rl.confusion_matrix(ml_pairs, result_nb, len(matches))
conf_nb
```

In [ ]:
```
## Precision and Accuracy
precision = rl.precision(conf_nb)
accuracy = rl.accuracy(conf_nb)
```

In [ ]:
```
## Precision and Accuracy
print(precision)
print(accuracy)
```

In [ ]:
```
## The F-score for this classification is
rl.fscore(conf_nb)
```

References and Further Readings

Parsing

- Python online documentation: https://docs.python.org/2/library/string.html#deprecated-string-functions (https://docs.python.org/2/library/string.html#deprecated-string-functions)
- Python 2.7 Tutorial(Splitting and Joining Strings): http://www.pitt.edu/~naraehan/python2/split_join.html (http://www.pitt.edu/~naraehan/python2/split_join.html)

Regular Expression

- Python documentation: https://docs.python.org/2/library/re.html#regular-expression-syntax (https://docs.python.org/2/library/re.html#regular-expression-syntax)
- Online regular expression tester (good for learning): http://regex101.com/ (http://regex101.com/)

String Comparators

- GitHub page of jellyfish: https://github.com/jamesturk/jellyfish (https://github.com/jamesturk/jellyfish)
- Different distances that measure the differences between strings:
  - Levenshtein distance: https://en.wikipedia.org/wiki/Levenshtein_distance (https://en.wikipedia.org/wiki/Levenshtein_distance)
  - Damerau–Levenshtein distance: https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance (https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance)
  - Jaro–Winkler distance: https://en.wikipedia.org/wiki/Jaro%E2%80%93Winkler_distance (https://en.wikipedia.org/wiki/Jaro%E2%80%93Winkler_distance)
  - Hamming distance: https://en.wikipedia.org/wiki/Hamming_distance (https://en.wikipedia.org/wiki/Hamming_distance)
  - Match rating approach: https://en.wikipedia.org/wiki/Match_rating_approach (https://en.wikipedia.org/wiki/Match_rating_approach)

Fellegi-Sunter Record Linkage

- Introduction to Probabilistic Record Linkage: http://www.bristol.ac.uk/media-library/sites/cmm/migrated/documents/problinkage.pdf (http://www.bristol.ac.uk/media-library/sites/cmm/migrated/documents/problinkage.pdf)
- Paper Review: https://www.cs.umd.edu/class/spring2012/cmsc828L/Papers/HerzogEtWires10.pdf (https://www.cs.umd.edu/class/spring2012/cmsc828L/Papers/HerzogEtWires10.pdf)