**The cutoff point**

In this exercise, and throughout this chapter, you'll be working with the restaurants DataFrame which has data on various restaurants. Your ultimate goal is to create a restaurant recommendation engine, but you need to first clean your data.

This version of restaurants has been collected from many sources, where the cuisine\_type column is riddled with typos, and should contain only italian, american and asian cuisine types. There are so many unique categories that remapping them manually isn't scalable, and it's best to use string similarity instead.

Before doing so, you want to establish the cutoff point for the similarity score using the thefuzz's process.extract() function by finding the similarity score of the most *distant* typo of each category.

**Instructions 1/2**

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* Import process from thefuzz.
* Store the unique cuisine\_types into unique\_types.
* Calculate the similarity of 'asian', 'american', and 'italian' to all possible cuisine\_types using process.extract(), while returning all possible matches.

# Import process from thefuzz

\_\_\_\_

# Store the unique values of cuisine\_type in unique\_types

unique\_types = \_\_\_\_

# Calculate similarity of 'asian' to all values of unique\_types

print(process.\_\_\_\_('\_\_\_\_', \_\_\_\_, limit = len(\_\_\_\_)))

# Calculate similarity of 'american' to all values of unique\_types

print(\_\_\_\_('\_\_\_\_', \_\_\_\_, \_\_\_\_))

# Calculate similarity of 'italian' to all values of unique\_types

print(\_\_\_\_)

process from thefuzz

from thefuzz import process

# Store the unique values of cuisine\_type in unique\_types

unique\_types = restaurants['cuisine\_type'].unique()

# Calculate similarity of 'asian' to all values of unique\_types

print(process.extract('asian', unique\_types, limit = len(unique\_types)))

# Calculate similarity of 'american' to all values of unique\_types

print(process.extract('american', unique\_types, limit= len(unique\_types) ))

# Calculate similarity of 'italian' to all values of unique\_types

print(process.extract('italian', unique\_types, limit = len(unique\_types)))

[('asian', 100), ('asiane', 91), ('asiann', 91), ('asiian', 91), ('asiaan', 91), ('asianne', 83), ('asiat', 80), ('italiann', 72), ('italiano', 72), ('italianne', 72), ('italian', 67), ('amurican', 62), ('american', 62), ('italiaan', 62), ('italiian', 62), ('itallian', 62), ('americann', 57), ('americano', 57), ('ameerican', 57), ('aamerican', 57), ('ameriican', 57), ('amerrican', 57), ('ammericann', 54), ('ameerrican', 54), ('ammereican', 54), ('america', 50), ('merican', 50), ('murican', 50), ('italien', 50), ('americen', 46), ('americin', 46), ('amerycan', 46), ('itali', 40)]

[('american', 100), ('americann', 94), ('americano', 94), ('ameerican', 94), ('aamerican', 94), ('ameriican', 94), ('amerrican', 94), ('america', 93), ('merican', 93), ('ammericann', 89), ('ameerrican', 89), ('ammereican', 89), ('amurican', 88), ('americen', 88), ('americin', 88), ('amerycan', 88), ('murican', 80), ('asian', 62), ('asiane', 57), ('asiann', 57), ('asiian', 57), ('asiaan', 57), ('italian', 53), ('asianne', 53), ('italiann', 50), ('italiano', 50), ('italiaan', 50), ('italiian', 50), ('itallian', 50), ('italianne', 47), ('asiat', 46), ('itali', 40), ('italien', 40)]

[('italian', 100), ('italiann', 93), ('italiano', 93), ('italiaan', 93), ('italiian', 93), ('itallian', 93), ('italianne', 88), ('italien', 86), ('itali', 83), ('asian', 67), ('asiane', 62), ('asiann', 62), ('asiian', 62), ('asiaan', 62), ('asianne', 57), ('amurican', 53), ('american', 53), ('americann', 50), ('asiat', 50), ('americano', 50), ('ameerican', 50), ('aamerican', 50), ('ameriican', 50), ('amerrican', 50), ('ammericann', 47), ('ameerrican', 47), ('ammereican', 47), ('america', 43), ('merican', 43), ('murican', 43), ('americen', 40), ('americin', 40), ('amerycan', 40)]

<script.py> output:

[('asian', 100), ('asiane', 91), ('asiann', 91), ('asiian', 91), ('asiaan', 91), ('asianne', 83), ('asiat', 80), ('italiann', 72), ('italiano', 72), ('italianne', 72), ('italian', 67), ('amurican', 62), ('american', 62), ('italiaan', 62), ('italiian', 62), ('itallian', 62), ('americann', 57), ('americano', 57), ('ameerican', 57), ('aamerican', 57), ('ameriican', 57), ('amerrican', 57), ('ammericann', 54), ('ameerrican', 54), ('ammereican', 54), ('america', 50), ('merican', 50), ('murican', 50), ('italien', 50), ('americen', 46), ('americin', 46), ('amerycan', 46), ('itali', 40)]

[('american', 100), ('americann', 94), ('americano', 94), ('ameerican', 94), ('aamerican', 94), ('ameriican', 94), ('amerrican', 94), ('america', 93), ('merican', 93), ('ammericann', 89), ('ameerrican', 89), ('ammereican', 89), ('amurican', 88), ('americen', 88), ('americin', 88), ('amerycan', 88), ('murican', 80), ('asian', 62), ('asiane', 57), ('asiann', 57), ('asiian', 57), ('asiaan', 57), ('italian', 53), ('asianne', 53), ('italiann', 50), ('italiano', 50), ('italiaan', 50), ('italiian', 50), ('itallian', 50), ('italianne', 47), ('asiat', 46), ('itali', 40), ('italien', 40)]

[('italian', 100), ('italiann', 93), ('italiano', 93), ('italiaan', 93), ('italiian', 93), ('itallian', 93), ('italianne', 88), ('italien', 86), ('itali', 83), ('asian', 67), ('asiane', 62), ('asiann', 62), ('asiian', 62), ('asiaan', 62), ('asianne', 57), ('amurican', 53), ('american', 53), ('americann', 50), ('asiat', 50), ('americano', 50), ('ameerican', 50), ('aamerican', 50), ('ameriican', 50), ('amerrican', 50), ('ammericann', 47), ('ameerrican', 47), ('ammereican', 47), ('america', 43), ('merican', 43), ('murican', 43), ('americen', 40), ('americin', 40), ('amerycan', 40)]

#### Question

Take a look at the output, what do you think should be the similarity cutoff point when remapping categories?

##### Possible Answers

* 

**80**

* 

70

* 

60

Correct! 80 is that sweet spot where you convert all incorrect typos without remapping incorrect categories. Often times though, you may need to combine the techniques learned in chapter 2, especially since there could be strings that make it beyond our cutoff point, but are not actually a match!

**Remapping categories II**

In the last exercise, you determined that the distance cutoff point for remapping typos of 'american', 'asian', and 'italian' cuisine types stored in the cuisine\_type column should be 80.

In this exercise, you're going to put it all together by finding matches with similarity scores equal to or higher than 80 by using fuzywuzzy.process's extract() function, for each correct cuisine type, and replacing these matches with it. Remember, when comparing a string with an array of strings using process.extract(), the output is a list of tuples where each is formatted like:

(closest match, similarity score, index of match)

The restaurants DataFrame is in your environment, and you have access to a categories list containing the correct cuisine types ('italian', 'asian', and 'american').

**Instructions 1/4**

**25 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Return all of the unique values in the cuisine\_type column of restaurants.

# Inspect the unique values of the cuisine\_type column

print(\_\_\_\_)

# Inspect the unique values of the cuisine\_type column

print(restaurants['cuisine\_type'].unique())

# Inspect the unique values of the cuisine\_type column

print(restaurants['cuisine\_type'].unique())

['america' 'merican' 'amurican' 'americen' 'americann' 'asiane' 'itali'

'asiann' 'murican' 'italien' 'italian' 'asiat' 'american' 'americano'

'italiann' 'ameerican' 'asianne' 'italiano' 'americin' 'ammericann'

'amerycan' 'aamerican' 'ameriican' 'italiaan' 'asiian' 'asiaan'

'amerrican' 'ameerrican' 'ammereican' 'asian' 'italianne' 'italiian'

'itallian']

Okay! Looks like you will need to use some string matching to correct these misspellings!

* As a first step, create a list of all possible matches, comparing 'italian' with the restaurant types listed in the cuisine\_type column.

# Create a list of matches, comparing 'italian' with the cuisine\_type column

matches = \_\_\_\_

# Inspect the first 5 matches

print(matches[0:5])

# Create a list of matches, comparing 'italian' with the cuisine\_type column

matches = process.extract('italian', restaurants['cuisine\_type'], limit =len(restaurants['cuisine\_type']))

# Inspect the first 5 matches

print(matches[0:5])

[('italian', 100, 11), ('italian', 100, 25), ('italian', 100, 41), ('italian', 100, 47), ('italian', 100, 49)]

<script.py> output:

[('italian', 100, 11), ('italian', 100, 25), ('italian', 100, 41), ('italian', 100, 47), ('italian', 100, 49)]

# Create a list of matches, comparing 'italian' with the cuisine\_type column

matches = process.extract('italian', restaurants['cuisine\_type'], limit=len(restaurants.cuisine\_type))

# Iterate through the list of matches to italian

for match in matches:

  # Check whether the similarity score is greater than or equal to 80

  \_\_\_\_:

    # Select all rows where the cuisine\_type is spelled this way, and set them to the correct cuisine

    \_\_\_\_

Now you're getting somewhere! Now you can iterate through matches to reassign similar entries.

* Within the for loop, use an if statement to check whether the similarity score in each match is greater than or equal to 80.
* If it is, use .loc to select rows where cuisine\_type in restaurants is *equal* to the current match (which is the first element of match), and reassign them to be 'italian'.

# Create a list of matches, comparing 'italian' with the cuisine\_type column

matches = process.extract('italian', restaurants['cuisine\_type'], limit=len(restaurants.cuisine\_type))

# Iterate through the list of matches to italian

for match in matches:

  # Check whether the similarity score is greater than or equal to 80

  if match[1]>=80:

    # Select all rows where the cuisine\_type is spelled this way, and set them to the correct cuisine

    restaurants.loc[restaurants['cuisine\_type'] == match[0], 'cuisine\_type']= 'italian'

# Create a list of matches, comparing 'italian' with the cuisine\_type column matches = process.extract('italian', restaurants['cuisine\_type'], limit=len(restaurants.cuisine\_type)) # Iterate through the list of matches to italian for match in matches: # Check whether the similarity score is greater than or equal to 80 if match[1]>=80: # Select all rows where the cuisine\_type is spelled this way, and set them to the correct cuisine restaurants.loc[restaurants['cuisine\_type'] == match[0], 'cuisine\_type']= 'italian'

Finally, you'll adapt your code to work with every restaurant type in categories.

* Using the variable cuisine to iterate through categories, embed your code from the previous step in an outer for loop.
* Inspect the final result. *This has been done for you.*
* # Iterate through categories
* for cuisine in \_\_\_\_:
* # Create a list of matches, comparing cuisine with the cuisine\_type column
* matches = process.extract(\_\_\_\_, restaurants['cuisine\_type'], limit=len(restaurants.cuisine\_type))
* # Iterate through the list of matches
* for match in matches:
* # Check whether the similarity score is greater than or equal to 80
* if match[1] >= 80:
* # If it is, select all rows where the cuisine\_type is spelled this way, and set them to the correct cuisine
* restaurants.loc[restaurants['cuisine\_type'] == match[0]] = \_\_\_\_
* # Inspect the final result
* print(restaurants['cuisine\_type'].unique())

# Create a list of matches, comparing 'italian' with the cuisine\_type column matches = process.extract('italian', restaurants['cuisine\_type'], limit=len(restaurants.cuisine\_type)) # Iterate through the list of matches to italian for match in matches: # Check whether the similarity score is greater than or equal to 80 if match[1]>=80: # Select all rows where the cuisine\_type is spelled this way, and set them to the correct cuisine restaurants.loc[restaurants['cuisine\_type'] == match[0], 'cuisine\_type']= 'italian'

# Iterate through categories

for cuisine in categories:

  # Create a list of matches, comparing cuisine with the cuisine\_type column

  matches = process.extract(cuisine, restaurants['cuisine\_type'], limit=len(restaurants.cuisine\_type))

  # Iterate through the list of matches

  for match in matches:

     # Check whether the similarity score is greater than or equal to 80

    if match[1] >= 80:

      # If it is, select all rows where the cuisine\_type is spelled this way, and set them to the correct cuisine

      restaurants.loc[restaurants['cuisine\_type'] == match[0]] = cuisine

# Inspect the final result

print(restaurants['cuisine\_type'].unique())

# Iterate through categories

for cuisine in categories:

# Create a list of matches, comparing cuisine with the cuisine\_type column

matches = process.extract(cuisine, restaurants['cuisine\_type'], limit=len(restaurants.cuisine\_type))

# Iterate through the list of matches

for match in matches:

# Check whether the similarity score is greater than or equal to 80

if match[1] >= 80:

# If it is, select all rows where the cuisine\_type is spelled this way, and set them to the correct cuisine

restaurants.loc[restaurants['cuisine\_type'] == match[0]] = cuisine

# Inspect the final result

print(restaurants['cuisine\_type'].unique())

['american' 'asian' 'italian']

Tremendous work! All your cuisine types are properly mapped! Now you'll build on string similarity, by jumping into record linkage!

**Daily XP950**

# Generating pairs

**50 XP**

## 1. Generating pairs

Great work with lesson 1 - you now have a solid understanding how to calculate string similarity.

## 2. Motivation

At the end of the last video exercise, we saw how record linkage attempts to join data sources with fuzzy duplicate values. For example here are two DataFrames containing NBA games and their schedules. They've both been scraped from different sites and we would want to merge them together and have one DataFrame containing all unique games.

## 3. When joins won't work

We see that there are duplicates values in both DataFrames with different naming marked here in red, and non duplicate values, marked here in green. Since there are games happening at the same time, no common unique identifier between the DataFrames, and the events are differently named, a regular join or merge will not work. This is where record linkage comes in.

## 4. Record linkage

Record linkage is the act of linking data from different sources regarding the same entity. Generally, we clean two or more DataFrames, generate pairs of potentially matching records, score these pairs according to string similarity and other similarity metrics, and link them. All of these steps can be achieved with the recordlinkage package, let's find how!

## 5. Our DataFrames

Here we have two DataFrames, census\_A, and census\_B, containing data on individuals throughout the states. We want to merge them while avoiding duplication using record linkage, since they are collected manually and are prone to typos, there are no consistent IDs between them.

## 6. Generating pairs

We first want to generate pairs between both DataFrames. Ideally, we want to generate all possible pairs between our DataFrames.

## 7. Generating pairs

but what if we had big DataFrames and ended up having to generate millions if not billions of pairs? It wouldn't prove scalable and could seriously hamper development time.

## 8. Blocking

This is where we apply what we call blocking, which creates pairs based on a matching column, which is in this case, the state column, reducing the number of possible pairs.

## 9. Generating pairs

To do this, we first start off by importing recordlinkage. We then use the recordlinkage dot Index function, to create an indexing object. This essentially is an object we can use to generate pairs from our DataFrames. To generate pairs blocked on state, we use the block method, inputting the state column as input. Once the indexer object has been initialized, we generate our pairs using the dot index method, which takes in the two dataframes.

## 10. Generating pairs

The resulting object, is a pandas multi index object containing pairs of row indices from both DataFrames, which is a fancy way to say it is an array containing possible pairs of indices that makes it much easier to subset DataFrames on.

## 11. Comparing the DataFrames

Since we've already generated our pairs, it's time to find potential matches. We first start by creating a comparison object using the recordlinkage dot compare function. This is similar to the indexing object we created while generating pairs, but this one is responsible for assigning different comparison procedures for pairs. Let's say there are columns for which we want exact matches between the pairs. To do that, we use the exact method. It takes in the column name in question for each DataFrame, which is in this case date\_of\_birth and state, and a label argument which lets us set the column name in the resulting DataFrame. Now in order to compute string similarities between pairs of rows for columns that have fuzzy values, we use the dot string method, which also takes in the column names in question, the similarity cutoff point in the threshold argument, which takes in a value between 0 and 1, which we here set to 0.85. Finally to compute the matches, we use the compute function, which takes in the possible pairs, and the two DataFrames in question. Note that you need to always have the same order of DataFrames when inserting them as arguments when generating pairs, comparing between columns, and computing comparisons.

## 12. Finding matching pairs

The output is a multi index DataFrame, where the first index is the row index from the first DataFrame, or census A, and the second index is a list of all row indices in census B. The columns are the columns being compared, with values being 1 for a match, and 0 for not a match.

## 13. Finding the only pairs we want

To find potential matches, we just filter for rows where the sum of row values is higher than a certain threshold. Which in this case higher or equal to 2. But we'll dig deeper into these matches and see how to use them to link our census DataFrames in the next lesson.

## 14. Let's practice!

But for now, let's generate pairs.

**Daily XP1000**

##### Exercise

#### To link or not to link?

Similar to joins, record linkage is the act of linking data from different sources regarding the same entity. But unlike joins, record linkage does not require exact matches between different pairs of data, and instead can find close matches using string similarity. This is why record linkage is effective when there are no common unique keys between the data sources you can rely upon when linking data sources such as a unique identifier.

In this exercise, you will classify each card whether it is a traditional join problem, or a record linkage one.

##### Instructions

**100XP**

* Classify each card into a problem that requires record linkage or regular joins.

Fabulous! Don't make things more complicated than they need to be: record linkage is a powerful tool, but it's more complex than using a traditional join.

**Daily XP1100**

**Exercise**

**Exercise**

**Pairs of restaurants**

In the last lesson, you cleaned the restaurants dataset to make it ready for building a restaurants recommendation engine. You have a new DataFrame named restaurants\_new with new restaurants to train your model on, that's been scraped from a new data source.

You've already cleaned the cuisine\_type and city columns using the techniques learned throughout the course. However you saw duplicates with typos in restaurants names that require record linkage instead of joins with restaurants.

In this exercise, you will perform the first step in record linkage and generate possible pairs of rows between restaurants and restaurants\_new. Both DataFrames, pandas and recordlinkage are in your environment.

**Instructions 1/2**

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* Instantiate an indexing object by using the Index() function from recordlinkage.
* Block your pairing on cuisine\_type by using indexer's' .block() method.
* Generate pairs by indexing restaurants and restaurants\_new in that order.

# Create an indexer and object and find possible pairs

indexer = \_\_\_\_

# Block pairing on cuisine\_type

indexer.\_\_\_\_(\_\_\_\_)

# Generate pairs

pairs = indexer.\_\_\_\_(\_\_\_\_, \_\_\_\_)

# Create an indexer and object and find possible pairs

indexer = recordlinkage.Index()

# Block pairing on cuisine\_type

indexer.block('cuisine\_type')

# Generate pairs

pairs = indexer.index(restaurants, restaurants\_new)

# Create an indexer and object and find possible pairs indexer = recordlinkage.Index() # Block pairing on cuisine\_type indexer.block('cuisine\_type') # Generate pairs pairs = indexer.index(restaurants, restaurants\_new)

#### Question

Now that you've generated your pairs, you've achieved the first step of record linkage. What are the steps remaining to link both restaurants DataFrames, and in what order?

##### Possible Answers

* ****

**Compare between columns, score the comparison, then link the DataFrames.**

* 

Clean the data, compare between columns, link the DataFrames, then score the comparison.

* 

Clean the data, compare between columns, score the comparison, then link the DataFrames.

Correct! In the next exercise, you will compare between columns and check out the matching potentially rows between both DataFrames!

**Daily XP1200**

##### Exercise

##### Exercise

# Similar restaurants

In the last exercise, you generated pairs between restaurants and restaurants\_new in an effort to cleanly merge both DataFrames using record linkage.

When performing record linkage, there are different types of matching you can perform between different columns of your DataFrames, including exact matches, string similarities, and more.

Now that your pairs have been generated and stored in pairs, you will find exact matches in the city and cuisine\_type columns between each pair, and similar strings for each pair in the rest\_name column. Both DataFrames, pandas and recordlinkage are in your environment.

##### Instructions 1/4

**25 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Instantiate a comparison object using the recordlinkage.Compare() function.
* # Create a comparison object
* comp\_cl = \_\_\_\_

# Create a comparison object

comp\_cl = recordlinkage.Compare()

* Use the appropriate comp\_cl method to find exact matches between the city and cuisine\_type columns of both DataFrames.
* Use the appropriate comp\_cl method to find similar strings with a 0.8 similarity threshold in the rest\_name column of both DataFrames.
* # Create a comparison object
* comp\_cl = recordlinkage.Compare()
* # Find exact matches on city, cuisine\_types
* comp\_cl.\_\_\_\_('\_\_\_\_', '\_\_\_\_', label='city')
* comp\_cl.\_\_\_\_('\_\_\_\_', '\_\_\_\_', label = 'cuisine\_type')
* # Find similar matches of rest\_name
* comp\_cl.\_\_\_\_('\_\_\_\_', '\_\_\_\_', label='name', \_\_\_\_ = \_\_\_\_)

# Create a comparison object

comp\_cl = recordlinkage.Compare()

# Find exact matches on city, cuisine\_types

comp\_cl.exact('city', 'city', label='city')

comp\_cl.exact('cuisine\_type', 'cuisine\_type', label = 'cuisine\_type')

# Find similar matches of rest\_name

comp\_cl.string('rest\_name', 'rest\_name', label='name',  threshold = 0.8)

# Create an indexer and object and find possible pairs

indexer = recordlinkage.Index()

# Block pairing on cuisine\_type

indexer.block('cuisine\_type')

# Generate pairs

pairs = indexer.index(restaurants, restaurants\_new)

# Create an indexer and object and find possible pairs

indexer = recordlinkage.Index()

# Block pairing on cuisine\_type

indexer.block('cuisine\_type')

# Generate pairs

pairs = indexer.index(restaurants, restaurants\_new)

# Create a comparison object

comp\_cl = recordlinkage.Compare()

# Create a comparison object

comp\_cl = recordlinkage.Compare()

# Find exact matches on city, cuisine\_types

comp\_cl.exact('city', 'city', label='city')

comp\_cl.exact('cuisine\_type', 'cuisine\_type', label = 'cuisine\_type')

# Find similar matches of rest\_name

comp\_cl.string('rest\_name', 'rest\_name', label='name', threshold = 0.8)

<Compare>

* Compute the comparison of the pairs by using the .compute() method of comp\_cl.

# Create a comparison object

comp\_cl = recordlinkage.Compare()

# Find exact matches on city, cuisine\_types -

comp\_cl.exact('city', 'city', label='city')

comp\_cl.exact('cuisine\_type', 'cuisine\_type', label='cuisine\_type')

# Find similar matches of rest\_name

comp\_cl.string('rest\_name', 'rest\_name', label='name', threshold = 0.8)

# Get potential matches and print

potential\_matches = comp\_cl.\_\_\_\_(pairs, \_\_\_\_, \_\_\_\_)

print(potential\_matches)

# Create a comparison object

comp\_cl = recordlinkage.Compare()

# Find exact matches on city, cuisine\_types -

comp\_cl.exact('city', 'city', label='city')

comp\_cl.exact('cuisine\_type', 'cuisine\_type', label='cuisine\_type')

# Find similar matches of rest\_name

comp\_cl.string('rest\_name', 'rest\_name', label='name', threshold = 0.8)

# Get potential matches and print

potential\_matches = comp\_cl.compute(pairs, restaurants, restaurants\_new)

print(potential\_matches)

# Get potential matches and print

potential\_matches = comp\_cl.compute(pairs, restaurants, restaurants\_new)

print(potential\_matches)

city cuisine\_type name

0 0 0 1 0.0

1 0 1 0.0

7 0 1 0.0

12 0 1 0.0

13 0 1 0.0

... ... ... ...

40 18 0 1 0.0

281 18 0 1 0.0

288 18 0 1 0.0

302 18 0 1 0.0

308 18 0 1 0.0

[3631 rows x 3 columns]

#### Question

Print out potential\_matches, the columns are the columns being compared, with values being 1 for a match, and 0 for not a match for each pair of rows in your DataFrames. To find potential matches, you need to find rows with more than matching value in a column. You can find them with

potential\_matches[potential\_matches.sum(axis = 1) >= n]

Where n is the minimum number of columns you want matching to ensure a proper duplicate find, what do you think should the value of n be?

##### Possible Answers

* 

**3 because I need to have matches in all my columns.**

* 

2 because matching on any of the 2 columns or more is enough to find potential duplicates.

* 

1 because matching on just 1 column like the restaurant name is enough to find potential duplicates.

That's correct! For this example, tightening your selection criteria will ensure good duplicate finds! In the next lesson, you're gonna build on what you learned to link these two DataFrames!

**Daily XP1300**

# Linking DataFrames

**50 XP**

## 1. Linking DataFrames

Awesome work on the first 2 lessons! You've made it to the last lesson of this course!

## 2. Record linkage

At this point, you've generated your pairs, compared them, and scored them.

## 3. Record linkage

Now it's time to link your data!

## 4. Our DataFrames

Remember our census DataFrames from the video of the previous lesson?

## 5. What we've already done

We've already generated pairs between them, compared four of their columns, two for exact matches and two for string similarity alongside a 0.85 threshold, and found potential matches.

## 6. What we're doing now

Now it's time to link both census DataFrames.

## 7. Our potential matches

Let's look closely at our potential matches. It is a multi-index DataFrame, where we have two index columns, record id 1, and record id 2.

## 8. Our potential matches

The first index column, stores indices from census A.

## 9. Our potential matches

The second index column, stores all possible indices from census\_B, for each row index of census\_A.

## 10. Our potential matches

The columns of our potential matches are the columns we chose to link both DataFrames on, where the value is 1 for a match, and 0 otherwise.

## 11. Probable matches

The first step in linking DataFrames, is to isolate the potentially matching pairs to the ones we're pretty sure of. We saw how to do this in the previous lesson, by subsetting the rows where the row sum is above a certain number of columns, in this case 3. The output is row indices between census A and census B that are most likely duplicates. Our next step is to extract the one of the index columns, and subsetting its associated DataFrame to filter for duplicates.

## 12. Probable matches

Here we choose the second index column, which represents row indices of census B. We want to extract those indices, and subset census\_B on them to remove duplicates with census\_A before appending them together.

## 13. Get the indices

We can access a DataFrame's index using the index attribute. Since this is a multi index DataFrame, it returns a multi index object containing pairs of row indices from census\_A and census\_B respectively. We want to extract all census\_B indices, so we chain it with the get\_level\_values method, which takes in which column index we want to extract its values. We can either input the index column's name, or its order, which is in this case 1.

## 14. Linking DataFrames

To find the duplicates in census B, we simply subset on all indices of census\_B, with the ones found through record linkage. You can choose to examine them further for similarity with their duplicates in census\_A, but if you're sure of your analysis, you can go ahead and find the non duplicates by repeating the exact same line of code, except by adding a tilde at the beginning of your subset. Now that you have your non duplicates, all you need is a simple append using the DataFrame append method of census A, and you have your linked Data!

## 15. Linking DataFrames

To recap, what we did was build on top of our previous work in generating pairs, comparing across columns and finding potential matches. We then isolated all possible matches, where there are matches across 3 columns or more, ensuring we tightened our search for duplicates across both DataFrames before we link them. Extracted the row indices of census\_B where there are duplicates. Found rows of census\_B where they are not duplicated with census\_A by using the tilde symbol. And linked both DataFrames for full census results!

## 16. Let's practice!

Now that you know how to link DataFrames, let's put those skills to action!

# Getting the right index

Here's a DataFrame named matches containing potential matches between two DataFrames, users\_1 and users\_2. Each DataFrame's row indices is stored in uid\_1 and uid\_2 respectively.

first\_name address\_1 address\_2 marriage\_status date\_of\_birth

uid\_1 uid\_2

0 3 1 1 1 1 0

... ... ... ... ... ...

... ... ... ... ... ...

1 3 1 1 1 1 0

... ... ... ... ... ...

... ... ... ... ... ...

How do you extract all values of the uid\_1 index column?

##### Answer the question

**50XP**

#### Possible Answers

* 

**matches.index.get\_level\_values(0)**

press1

* 

matches.index.get\_level\_values(1)

press2

* 

**matches.index.get\_level\_values('uid\_1')**

press3

* 

**Both 1 and 3 are correct.**

press4

Correct! In the next exercise, you'll use these functions to subset your data and link your DataFrames!

**Daily XP1400**

**Exercise**

**Exercise**

**Linking them together!**

In the last lesson, you've finished the bulk of the work on your effort to link restaurants and restaurants\_new. You've generated the different pairs of potentially matching rows, searched for exact matches between the cuisine\_type and city columns, but compared for similar strings in the rest\_name column. You stored the DataFrame containing the scores in potential\_matches.

Now it's finally time to link both DataFrames. You will do so by first extracting all row indices of restaurants\_new that are matching across the columns mentioned above from potential\_matches. Then you will subset restaurants\_new on these indices, then append the non-duplicate values to restaurants. All DataFrames are in your environment, alongside pandas imported as pd.

**Instructions**

**100 XP**

* Isolate instances of potential\_matches where the row sum is above or equal to 3 by using the .sum() method.
* Extract the second column index from matches, which represents row indices of matching record from restaurants\_new by using the .get\_level\_values() method.
* Subset restaurants\_new for rows that are not in matching\_indices.
* Append non\_dup to restaurants.
* # Isolate potential matches with row sum >=3
* matches = \_\_\_\_[\_\_\_\_.\_\_\_(\_\_\_\_) >= \_\_\_\_]
* # Get values of second column index of matches
* matching\_indices = matches.\_\_\_\_.\_\_\_\_(\_\_\_\_)
* # Subset restaurants\_new based on non-duplicate values
* non\_dup = \_\_\_\_[~restaurants\_new.index.\_\_\_\_(\_\_\_\_)]
* # Append non\_dup to restaurants
* full\_restaurants = restaurants.\_\_\_\_(\_\_\_\_)
* print(full\_restaurants)

# Create a comparison object

comp\_cl = recordlinkage.Compare()

# Find exact matches on city, cuisine\_types -

comp\_cl.exact('city', 'city', label='city')

comp\_cl.exact('cuisine\_type', 'cuisine\_type', label='cuisine\_type')

# Find similar matches of rest\_name

comp\_cl.string('rest\_name', 'rest\_name', label='name', threshold = 0.8)

# Get potential matches and print

potential\_matches = comp\_cl.compute(pairs, restaurants, restaurants\_new)

print(potential\_matches)

city cuisine\_type name

0 0 0 1 0.0

1 0 1 0.0

7 0 1 0.0

12 0 1 0.0

13 0 1 0.0

... ... ... ...

40 18 0 1 0.0

281 18 0 1 0.0

288 18 0 1 0.0

302 18 0 1 0.0

308 18 0 1 0.0

[3631 rows x 3 columns]

<script.py> output:

city cuisine\_type name

0 0 0 1 0.0

1 0 1 0.0

7 0 1 0.0

12 0 1 0.0

13 0 1 0.0

... ... ... ...

40 18 0 1 0.0

281 18 0 1 0.0

288 18 0 1 0.0

302 18 0 1 0.0

308 18 0 1 0.0

[3631 rows x 3 columns]

# Create a comparison object

comp\_cl = recordlinkage.Compare()

# Find exact matches on city, cuisine\_types -

comp\_cl.exact('city', 'city', label='city')

comp\_cl.exact('cuisine\_type', 'cuisine\_type', label='cuisine\_type')

# Find similar matches of rest\_name

comp\_cl.string('rest\_name', 'rest\_name', label='name', threshold = 0.8)

# Get potential matches and print

potential\_matches = comp\_cl.compute(pairs, restaurants, restaurants\_new)

print(potential\_matches)

# Isolate potential matches with row sum >=3

matches = potential\_matches[potential\_matches.sum(axis=1) >= 3]

# Get values of second column index of matches

matching\_indices = matches.index.get\_level\_values(1)

# Subset restaurants\_new based on non-duplicate values

non\_dup = restaurants\_new[~restaurants\_new.index.isin(matching\_indices)]

# Append non\_dup to restaurants

full\_restaurants = restaurants.append(non\_dup)

print(full\_restaurants)

# Isolate potential matches with row sum >=3

matches = potential\_matches[potential\_matches.sum(axis=1) >= 3]

# Get values of second column index of matches

matching\_indices = matches.index.get\_level\_values(1)

# Subset restaurants\_new based on non-duplicate values

non\_dup = restaurants\_new[~restaurants\_new.index.isin(matching\_indices)]

# Append non\_dup to restaurants

full\_restaurants = restaurants.append(non\_dup)

print(full\_restaurants)

rest\_name rest\_addr city phone cuisine\_type

0 arnie morton's of chicago 435 s. la cienega blv . los angeles 3102461501 american

1 art's delicatessen 12224 ventura blvd. studio city 8187621221 american

2 campanile 624 s. la brea ave. los angeles 2139381447 american

3 fenix 8358 sunset blvd. west hollywood 2138486677 american

4 grill on the alley 9560 dayton way los angeles 3102760615 american

.. ... ... ... ... ...

76 don 1136 westwood blvd. westwood 3102091422 italian

77 feast 1949 westwood blvd. west la 3104750400 chinese

78 mulberry 17040 ventura blvd. encino 8189068881 pizza

80 jiraffe 502 santa monica blvd santa monica 3109176671 californian

81 martha's 22nd street grill 25 22nd st. hermosa beach 3103767786 american

[396 rows x 5 columns]

Awesome work! Linking the DataFrames is arguably the most straightforward step of record linkage. You are now ready to get started on that recommendation engine!