CS598: Theory and Practice of Data Cleaning (Summer 2017)

Final Project:
End-to-End Data Cleaning Workflow

Team Members:

Greg Embree (netid: gembre2)

Kin Keung, Wong (netid: kinwong)

Table of Content

Introduction

In this final project, we selected New York Public Library's crowd-sourced historical menus as the dataset to clean.

Repository

Our data and scripting files were verison-controlled in a git repository hosted by BitBucket: https://bitbucket.org/calvin-wong/s17-cs598-finalproject/src
There are five folders:

- **dataset_original** stores the original NYPL menu dataset files
- **dataset_post_openrefine** stores the menu dataset files after cleaned by OpenRefine
- **openrefine_change_json** stores the json files exported by OpenRefine with the history of the changes
- **Database** stores the db file used by SQLite and sql files for ICs
- **workflow** stores the workflow files to illustrate the workflow of the cleaning steps

1. Overview and initial assessment (3 days)

1.1 Original Dataset Assessment

When we started, we went through many datasets from kaggle.com. We selected "the full text of Stack Overflow Q&A about the Python programming language" , https://www.kaggle.com/stackoverflow/pythonquestions. It has 607,282 rows of questions, 987,122 rows of answers, and 1,885,078 rows of tags. The first problem we ran into was that even though OpenRefine was allocated 8GB of heap, importing data took more than ten minutes and we were never able to complete any clustering operation; we waited over 30 mins and decided to restart the application. Our solution was to use a script to split the dataset by year, from 2008 to 2016. This solved our problem of OpenRefine responsiveness. However once we started cleaning, we found the dataset didn't offer much room to clean. The following is an example row of data right after importing to OpenRefine:

4.	594	116	2008-08-03T01:15:08Z	25	cx_Oracle: How do I iterate over a result set?	There are several ways to iterate over a result set. What are the tradeoff of each?

In just one step where we remove html tags, the data is already clean enough to be loaded into database.

1.2 Current Dataset Assessment

As a result, we selected option (b) from the Final Project handout, the New York Public Library's crowd-sourced historical menus, as the dataset for this project. This dataset has four comma-delimited text files. We pushed the original dataset in our git repository (git folder link)

Dish.csv (git file link) has 9 columns and 423,397 rows. The name column had some data quality issue and the rest of the columns are fairly clean. See table below.

Column	Description	Sample Value	Quality Issue
dish_id	Unique identifier of dish	1	None
name	Name of the dish	Consomme printaniere royal	Casing, double-quoted, and ambiguity (This, side, is), comma in name
description	Description of the dish		Empty description
menus_appeared	Number of times appear in different menu	8	None
times_appeared	Number of times appear in any menu	8	
first_appeared	Year that the dish first appeared	1897	Values other than year, like 0, 1. Invalid year like 2928
last_appeared	Year that this dish last appeared	1927	Value other than year. Invalid year like 2928
lowest_price	Lowest price in US dollar	0.2	Empty price
highest_price	Highest price in US dollar	0.4	Empty price

Menu.csv (git file link) has 20 columns and 17,545 rows. In addition to columns with text as values, columns with date or numeric values also have quality issues. See table below

Column	**Description**	**Sample Value**	**Quality Issue**
menu_id	Unique Identifier of menu	12463	None
name	Menu name		Empty string, [Not Given], and etc
sponsor	Menu sponsor	HOTEL EASTMAN	Empty string, question mark as value
event	Time of the menu, like breakfast, lunch, or dinner	BREAKFAST	Empty string, question mark as value
venue	Type of the restaurant	COMMERCIAL	Empty string, question mark as value
place	City and state	HOT SPRINGS, AR	Empty string, question mark as value
physical description	Physical description	CARD; 4.75X7.5;	#N/A, empty string
occasion	Season, occasion of the menu	EASTER;	Empty string, semi-colon, and question mark as value
notes	Notes of the menu		Casing, quotes, empty string
call_number	Number used by the restaurant	1900-2822	Ambiguous value (_wotm), empty value
keywords	No description		Empty value
language	No description		Empty value
date	Date of the menu	1900-04-15	Invalid year (0001-01-01), empty value
location	Place or location of	Hotel Eastman	Question mark as

	the restaurant		value
location_type	No description		Empty value
currency	Currency		Empty value
currency_symbol	Currency symbol		Empty value
status	Status of review	complete	None
page_count	Number of page in menu	2	None
dish_count	Number of dish in menu	67	Zero value

MenuItem.csv (git file link) has 9 columns and 1,332,727 rows. Columns with coordinates and numeric values had quality issue with zero value. See table below.

Column	**Description**	**Sample Value**	**Quality Issue**
id	Unique identifier of menu item	1	None
menu_page_id	Menu id that the item is in	1389	None
price	Price of the item	0.4	Empty value
high_price	High price of the item		Empty value
dish_id	Dish id that this item maps to	1	None
created_at	Timestamp this item was created	2011-03-28 15:00:44 UTC	None
updated_at	Timestamp this item was updated	2011-04-19 04:33:15 UTC	None
xpos	X position of the item on menu	0.111429	Zero value
ypos	Y position of the item on menu	0.254735	Zero value

MenuPage.csv (git file link) has 7 columns and 66,937 rows. Various columns with empty value had data quality issue. See data below.

Column	Description	Sample Value	Quality Issue
menu_page_id	Unique identifier of the menu page	119	None
menu_id	Menu id that this menu page maps to	12460	None
page_number	Page number of the menu page	1	Empty value
image_id	Unique identifier of image for this menu page	1603595	None
full _height	Height of the menu page	7230	Empty value
full_width	Width of the menu page	5428	Empty value
uuid	Random id	510d47e4-2955-a3d9-e040-e00a18064a99	None

1.3 Use Cases

We came up with the following use cases for this NYPL menu dataset:

i. To infer the most popular types of dishes according to how many menus contain the dish.

ii. To have a good look into the type of menu that was offered (breakfast, lunch, dinner) but also what the menu was printed on (and potentially the size). For instance, folder, card, etc.

iii. To look at which parts of the world have a certain genre of food is consumed frequently.

iv. How has dining changed over the years? Are the types of food people consumed throughout the years consistent or has there been some evolution in what restaurateurs have provide on their menus.

For use case i, ii , and iii, we saw many variants of the same dish, event, occasion, and etc. So we thought clustering was really important. To make clustering even more accurate, we put in a step to remove special characters, like (,),[,],<.>,:,;, and etc. For use case iv, we would make sure to clean outlier dates, like 2928-03-26, and we further checked years were within 1851 and 2012.

2. Data Cleaning With OpenRefine (2 days)

We pushed the files cleaned by OpenRefine in our git repository (git folder link)

2.1 Increase Java heap size for OpenRefine

By default, the maximum heap allocation of OpenRefine is 1GB. But using OSX Activity Monitor, we found OpenRefine is memory bound and operations like importing could sometimes take up to 10 minutes. Our system has 16GB RAM and so we increased the maximum heap allocation to 8GB. Specifically we opened OpenRefine/Contents/Info.plist file and changed -Xms512M to -Xms2048M, and -Xmx1024M to -Xmx8196M.

```
<key>JVMOptions</key>
<array>
<string>-Xms2048M</string>
<string>-Xmx8196M</string>
```

2.2 Dish.csv (423,397 rows)

name column is the only text column in Dish.csv and we applied the following cleaning steps in OpenRefine. In our git repository, here is the link to the cleaned Menu file, Dish-csv.tsv, and the link to the OpenRefine json file with the change history, metadata_Dish.json.

2.2.1 Cleaning name column

2.2.1.1 Whitespace in name column

We first applied “Trim leading and trailing whitespace” and 9,045 cells were updated. Then we applied “Collapse consecutive whitespace” and 6,415 cells were updated.

2.2.1.2 Lowercase and special characters in name column using GREL

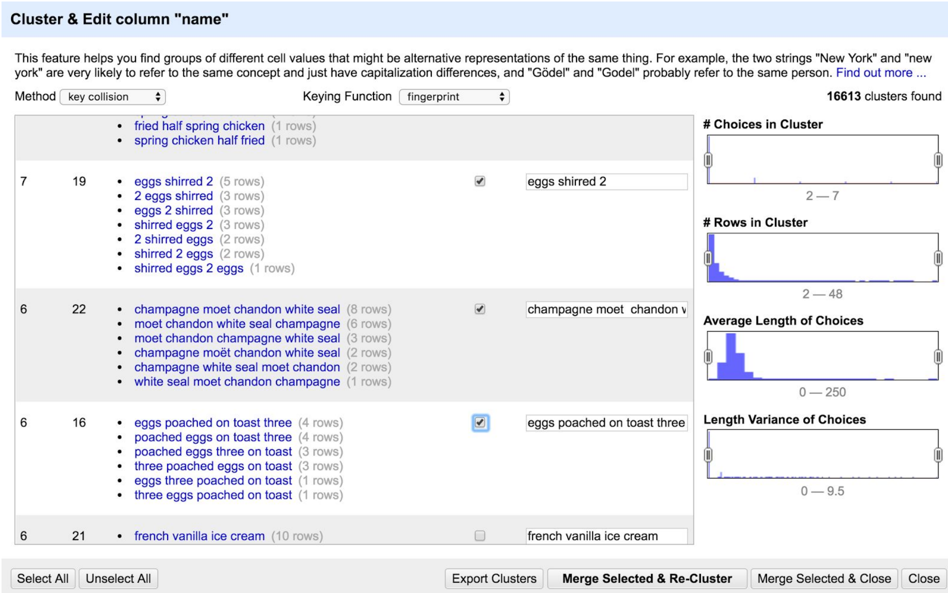
In order to reduce the number of clusters in next step, we transformed all characters in the column to lowercase, resulting in 411,704 updated cells. We also removed all special characters using GREL:

```
replace(value, /[%@#!\\\[\](),.&"':;\-_\*<>]/, "")
```

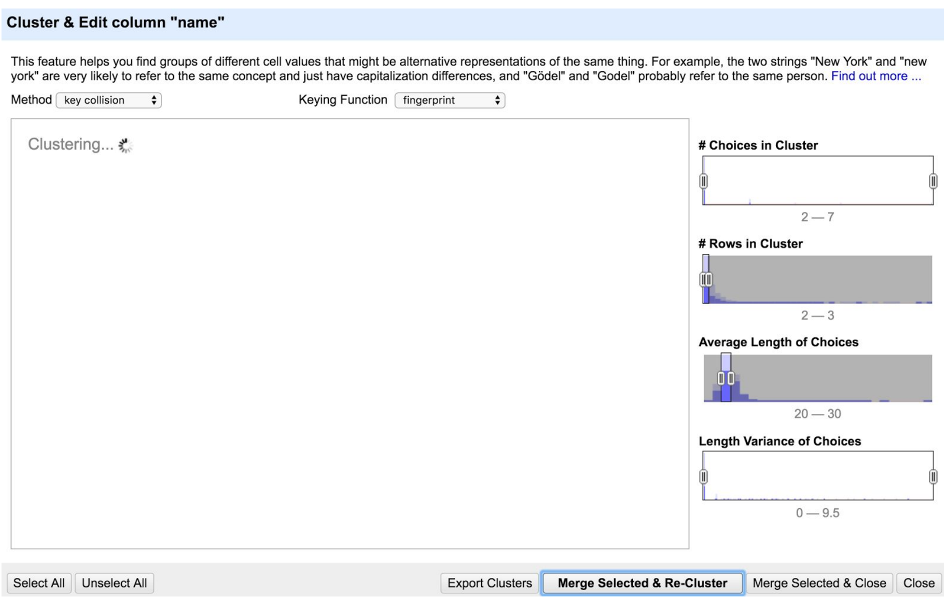
resulting in 219,603 updated cells.

2.2.1.3 Cluster and edit in name column

Selecting “Edit cells” -> “Clustering and edit”, there are 16,613 clusters.



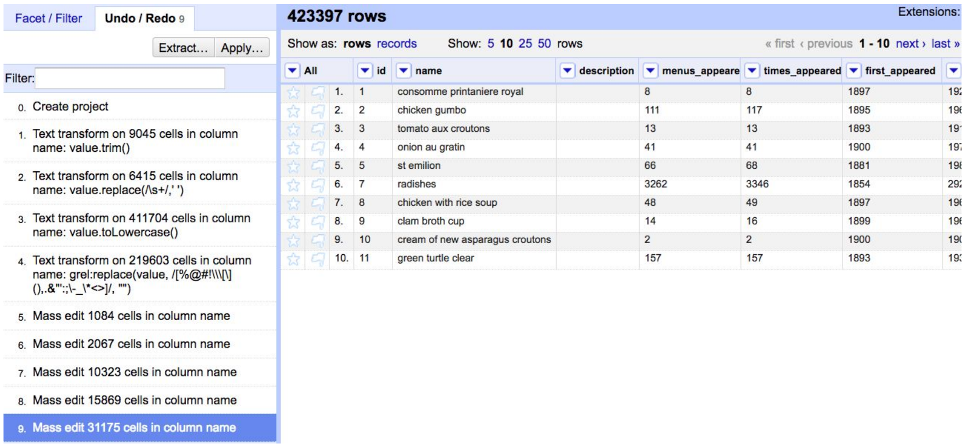
Ideally we could click “Select All” and then “Merge Selected & Re-Cluster” to merge all 16,613 clusters. However OpenRefine threw “Form too large error”. We got over by reducing the size of the clusters by narrowing the “Rows in Cluster” and “Average Length of Choices” using the sliding bars on the right pane.



This resulted in smaller cluster size and this made running “Select All” and “Merge Selected & “Re-Cluster” possible. Next iteration of clustering would reset the “Rows in Cluster” and “Average Length of Choices” to include those clusters that were previously excluded. We would again apply the same steps to reduce the cluster size, and merge clusters again. After 5 iterations, all 16,613 were merged and 60,518 cells were updated.

2.2.2 Cleaning Summary Dish.csv

The following showed the cleaning steps after cleaning is complete for Dish.csv; it is clean enough to be loaded into database.



2.3 Menu.csv (17,545 rows)

Cleaning Menu.csv requires fewer steps because clustering and merging can be done in one iteration. In our git repository, here is the link to the clean Menu file, Menu-csv.tsv, and the link to the OpenRefine json file with the change history, metadata_Menu.json.

2.3.1 Cleaning name column

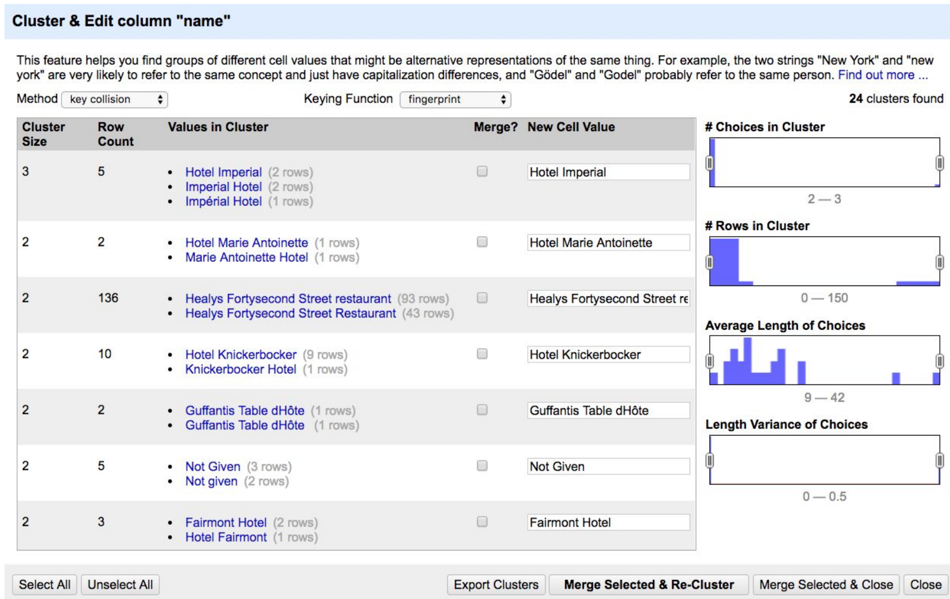
2.3.1.1 Remove special characters

We removed special characters using GREL and resulted in 1,013 changes:

```
replace(value, /[%@#!?\\\[\](),.&"':;\-_\*<>]/, "")
```

2.3.1.2 Cluster and edit in name column

There were 24 clusters found. Clicking “Select All” and then “Merge Selected & Re-Cluster” updated 530 cells.



2.3.2 Cleaning sponsor column

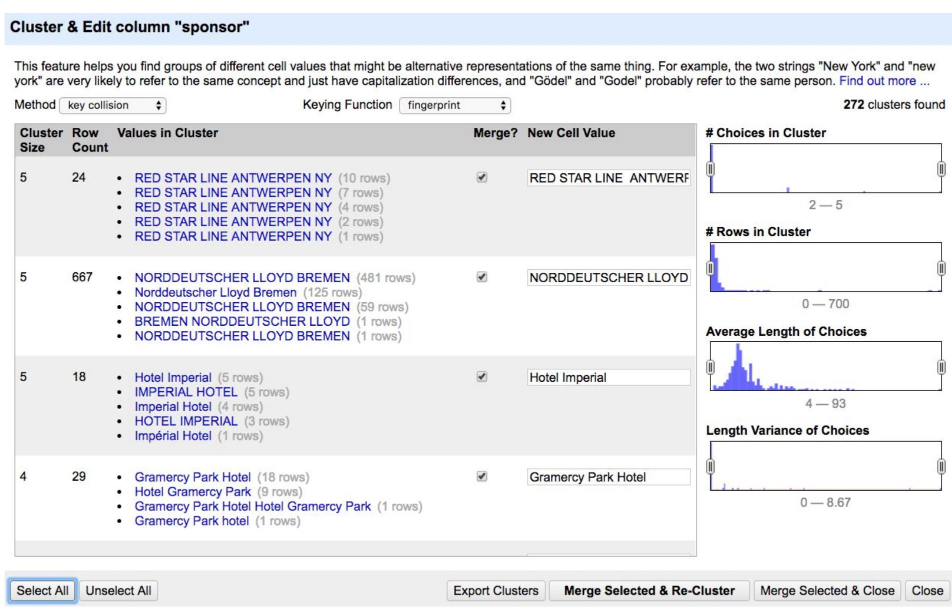
2.3.2.1 Remove special characters

We removed special characters using GREL and resulted in 4,721 changes:

```
replace(value, /[%@#!?\\\[\]( ),.&"':;\-_\*<>]/, "")
```

2.3.2.2 Cluster and edit in sponsor column

There are 272 clusters found. Clicking “Select All” and then “Merge Selected & Re-Cluster” updated 5,035 cells.



2.3.3 Cleaning event column

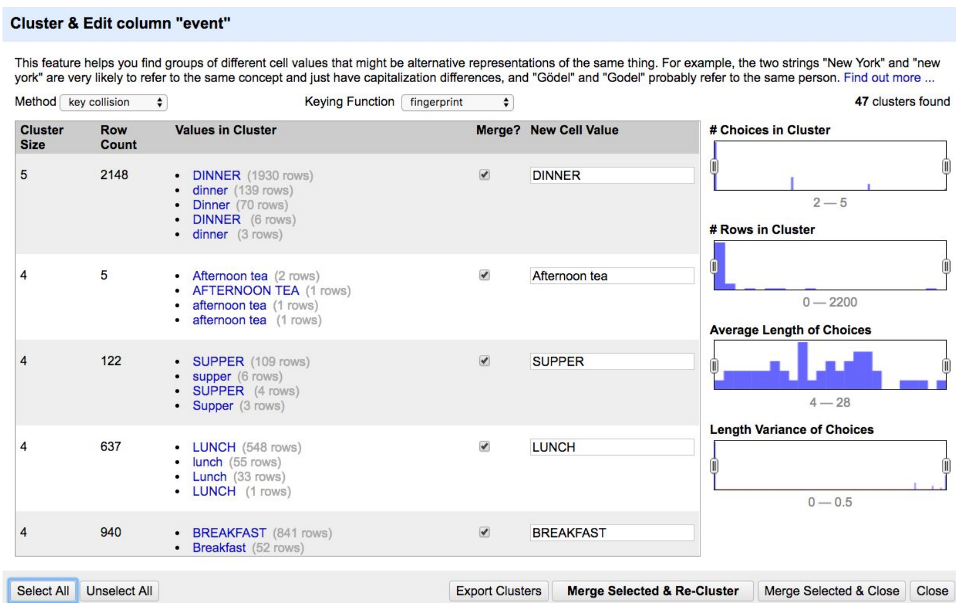
2.3.3.1 Remove special characters

We removed special characters using GREL and resulted in 1,128 changes:

```
replace(value, /[%@#!?\\\[\]().&"':;\-_\*<>]/, "")
```

2.3.3.2 Cluster and edit in event column

There are 47 clusters found. Clicking “Select All” and then “Merge Selected & Re-Cluster” updated 5,427 cells.



2.3.4 Cleaning venue column

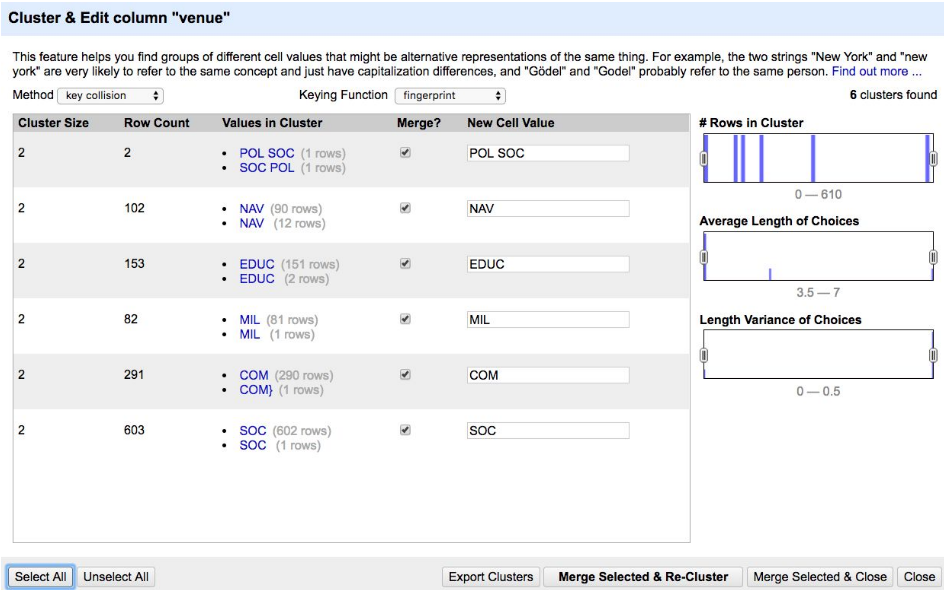
2.3.4.1 Remove special characters

We removed special characters using GREL and resulted in 2,221 changes:

```
replace(value, /[%@#!?\\\[\]().&"':;\-_\*<>]/, "")
```

2.3.4.2 Cluster and edit in venue column

There are 6 clusters found. Clicking “Select All” and then “Merge Selected & Re-Cluster” updated 1,233 cells.



2.3.5 Cleaning place column

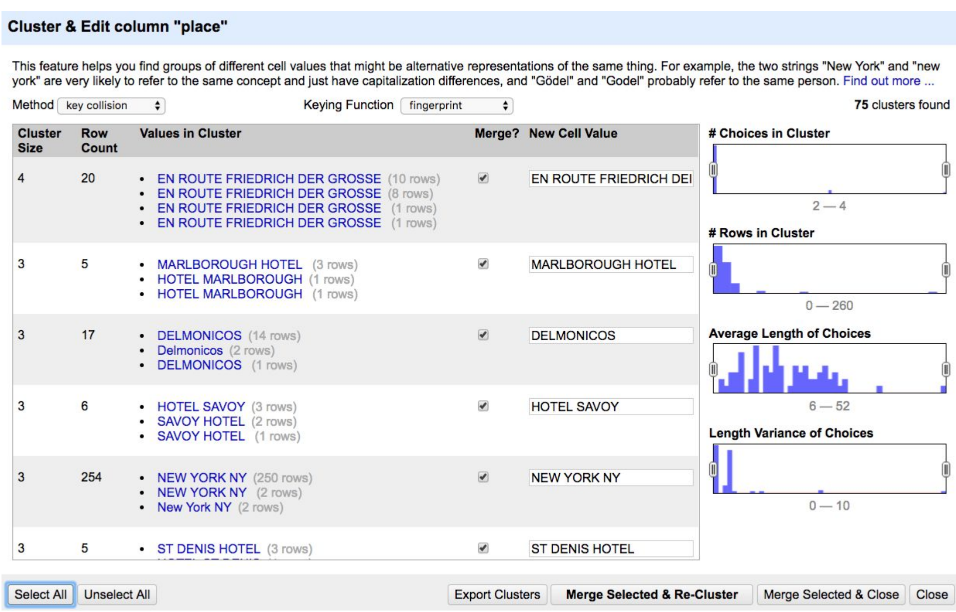
2.3.5.1 Remove special characters

We removed special characters using GREL and resulted in 5,521 changes:

```
replace(value, /[%@#!?\\\[\](),.&"':;\-_\*<>]/, "")
```

2.3.5.2 Cluster and edit in place column

There are 75 clusters found. Clicking “Select All” and then “Merge Selected & Re-Cluster” updated 897 cells.

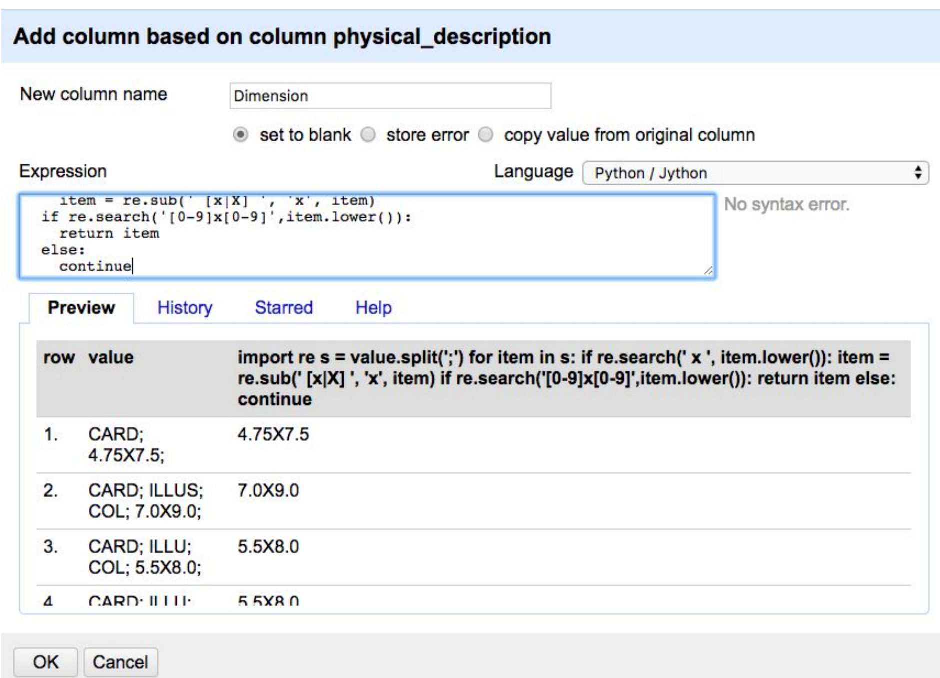


2.3.6 Cleaning physical_description column

2.3.6.1 Extract dimension into new column using Python

Unlike in homework 2, we would like to separate move dimension data (5.5X8.0) into a separate column. To do that, we made a copy of the physical_description column by selecting "Edit column" and then "Add column based on this column". The new column is called Dimension. We applied the following python code into the Expression field to get the dimension data only:

```
import re
s = value.split(';')
for item in s:
  if re.search(' x ', item.lower()):
    item = re.sub(' [x|X] ', 'x', item)
  if re.search('[0-9]x[0-9]',item.lower()):
    return item
  else:
    continue
```

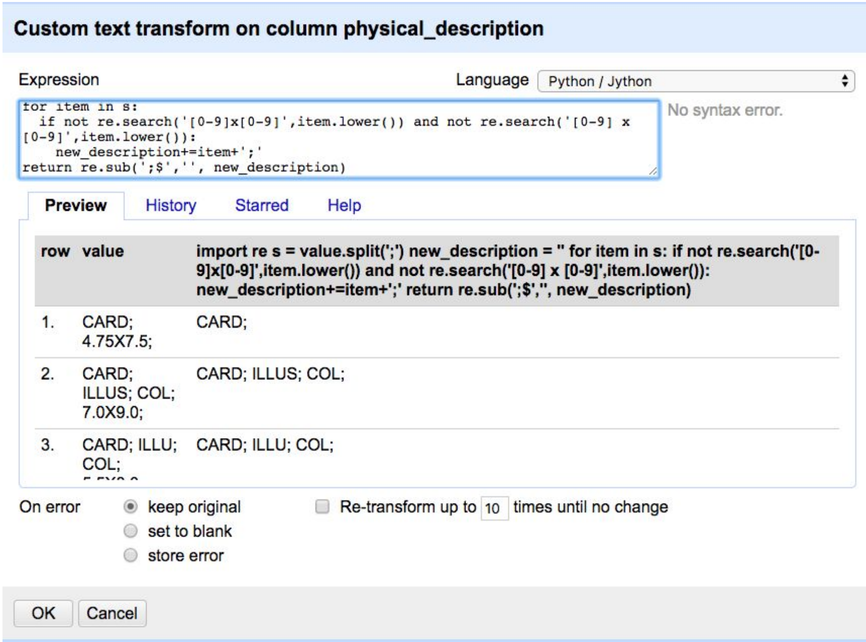


2.3.6.2 Remove dimension data from physical_description column using Python

We applied the following python code to the physical_description column to remove dimension data in 14,605 cells.

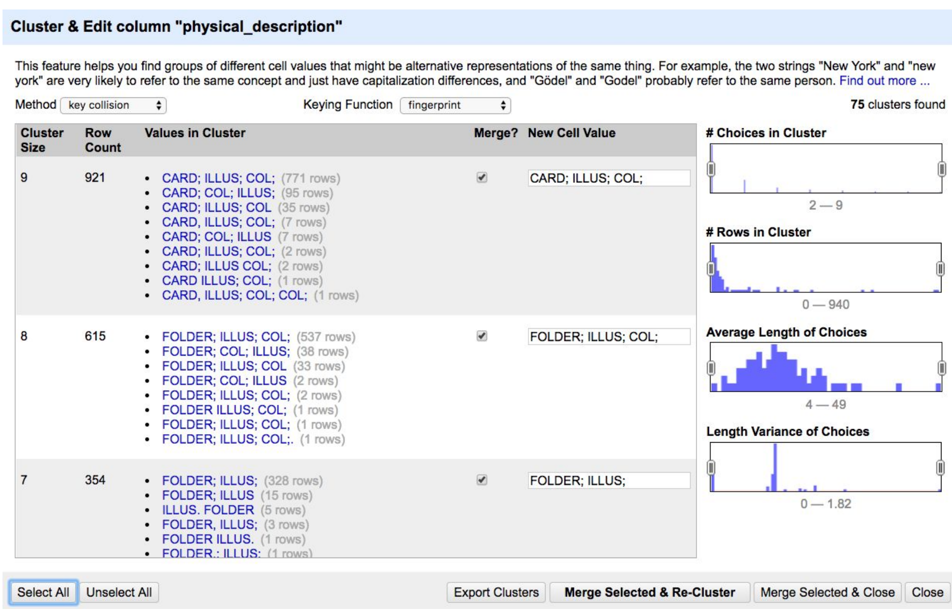
```
import re
s = value.split(';')
new_description = ''
for item in s:
  if not re.search('[0-9]x[0-9]',item.lower()) and not re.search('[0-9] x
[0-9]',item.lower()):
```

```
    new_description+=item+';'
return re.sub(';$','', new_description)
```



2.3.6.3 Cluster and edit in physical_description column

There are 75 clusters found. Clicking “Select All” and then “Merge Selected & Re-Cluster” updated 7,532 cells.



2.3.7 Cleaning occasion column

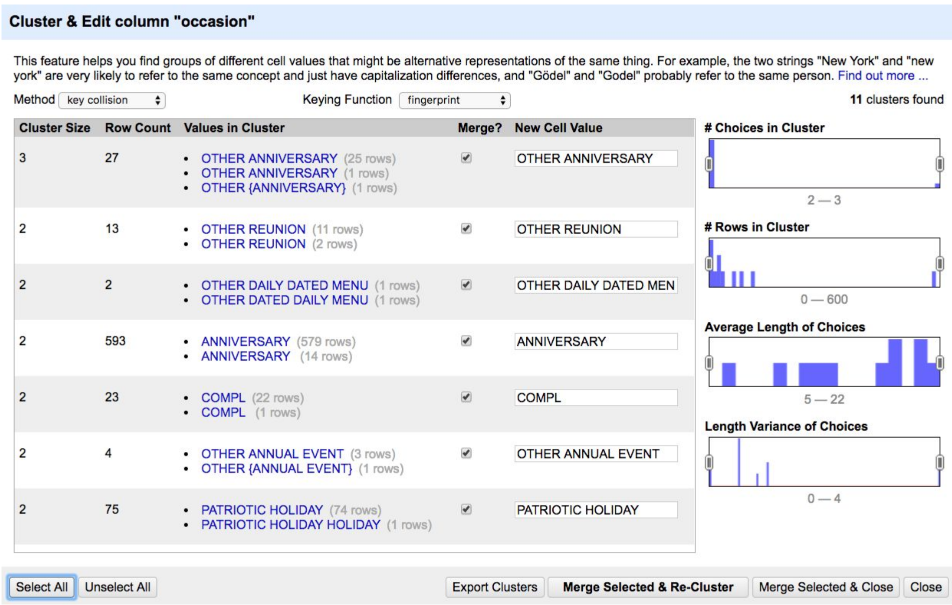
2.3.7.1 Remove special characters

We removed special characters using GREL and resulted in 2,597 changes:

```
replace(value, /[%@#!?\\\[\](),.&"':;\-_\*<>]/, "")
```

2.3.7.2 Cluster and edit in occasion column

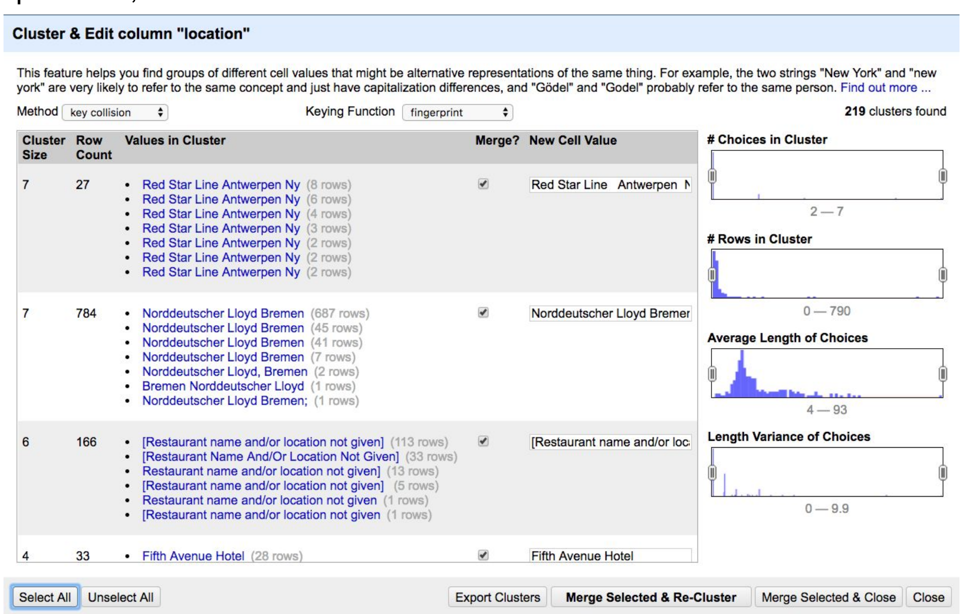
There are 11 clusters found. Clicking “Select All” and then “Merge Selected & Re-Cluster” updated 933 cells.



2.3.8 Cleaning location column

2.3.8.1 Cluster and edit in location column

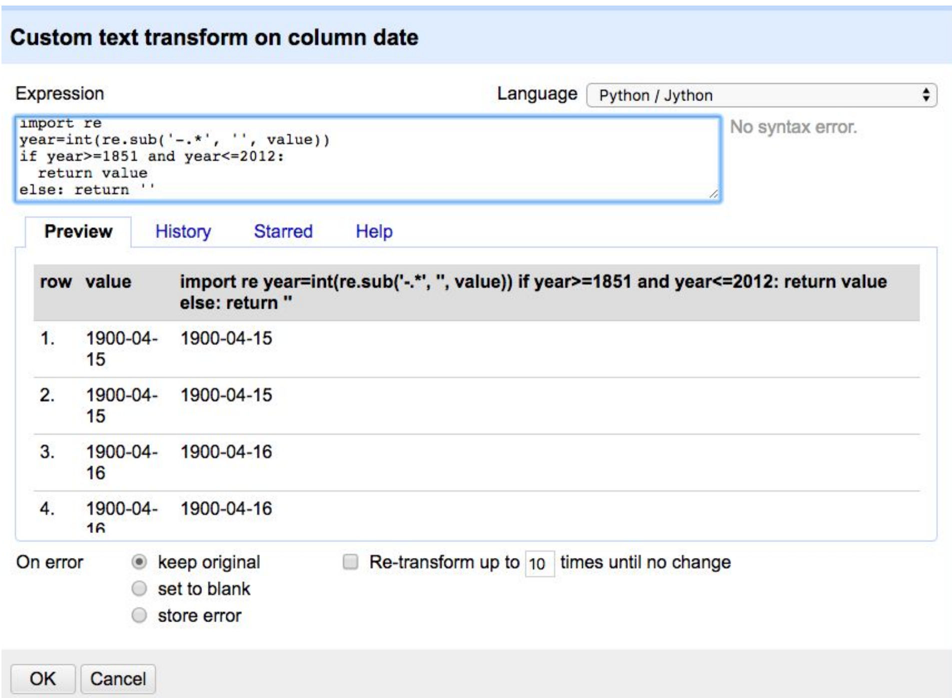
There are 219 clusters found. Clicking “Select All” and then “Merge Selected & Re-Cluster” updated 5,045 cells.



2.3.9 Cleaning date column

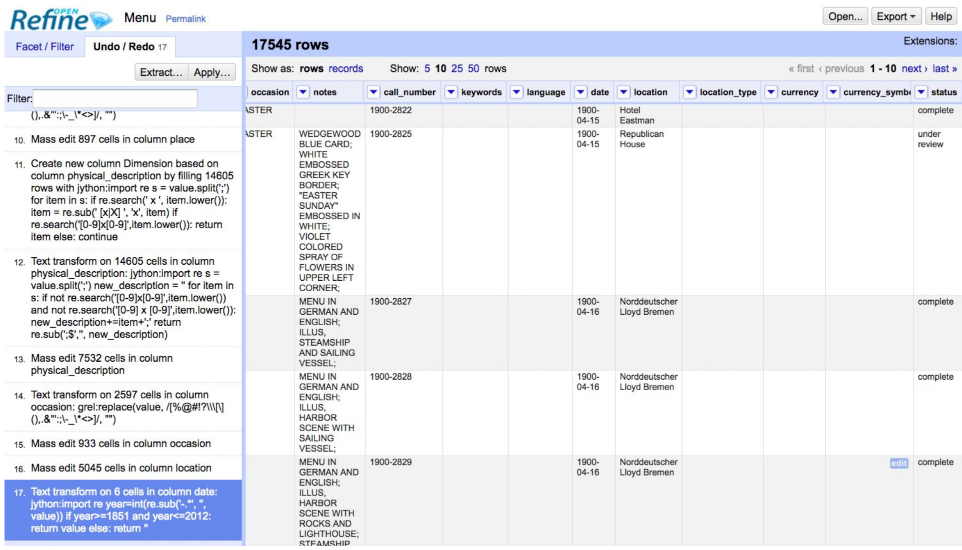
Date column has invalid years, for example 2928-03-26. We applied the following python code to only retain data within year 1851 and 2012, result changes in 6 cells:

```
import re
year=int(re.sub('-.*', '', value))
if year>=1851 and year<=2012:
  return value
else: return ''
```



2.3.10 Cleaning Summary Menu.csv

There are no many other significant clusters found in other columns. The following showed the cleaning steps after cleaning is complete for Menu.csv; it is clean enough to be loaded into database.



2.4 MenuItem.csv (1,332,726 rows) and MenuPage.csv (66,937 rows)

Both files contain all numeric values. We ran them through OpenRefine and didn't find much to clean in the two files.

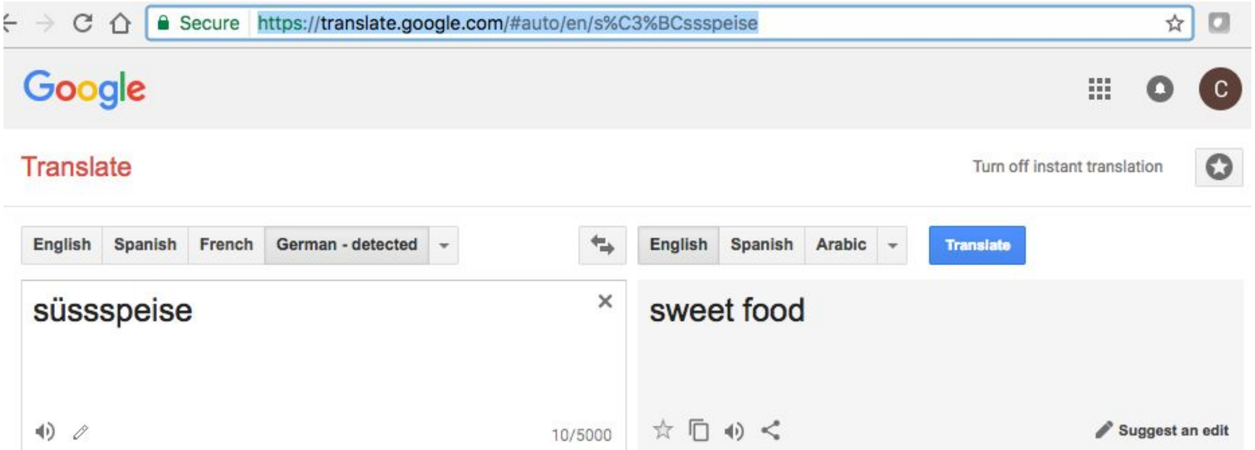
3. Alternative Refinement (3 days)

We attempted to create a new column in Dish.csv to display the English name of the dish names as many of the dishes have non-English names in them. We first wrote the following python function that tried to decode input with "ascii". Any exception in decoding would indicate input has at least one non-English name. We started passing dish names into this function.

```
def isAscii(input):
  try:
    input.decode('ascii')
    return True
  except UnicodeDecodeError:
    return False
```

Out of 516,676 dish names in Dish-csc.tsv, we found a set of 10,834 dish names that has non-English names in them. When we looked further in the set, we found many of the dish names in the set contain more than one unigram but not all of the unigrams are non-English. So we splitted the unigram using space as a delimiter and pass the unigram into function above. Out of the 10,834 dish names, we found only 3,132 unigrams that are non-English.

Our next step is to run the set non-English unigrams to Google Cloud Translate API to translate to English. To illustrate how it works, we selected one unigram from the set, süssspeise. Using Google Translate on browser, https://translate.google.com/#auto/en/s%C3%BCssspeise, we learned süssspeise is German and its English name was "sweet food".



The idea is to write a key value pair into a dictionary with süssspeise as the key and "sweet food" as the value. After we finish building this dictionary, we would start iterating through the non-English dish names and for each dish name, we would split and iterate the unigram. When a unigram is found in the dictionary as a key, we have a translation and then we would replace it with the value of the dictionary. After finishing iterating each dish name, we would write it to the new column, name_english. The python script was implemented like below and here is the link to it in git with additional YesWorflow headers, alternate_refinement_dish_workflow.py.

```
import re,sys

#Global data structures
translated_unigram_mapping = dict()
translated_dish_name_mapping = dict()
COL_NAME = 1

def isAscii(unigram):
  try:
    unigram.decode('ascii')
    return True
  except UnicodeDecodeError:
    return False

def google_translate(unigram):
  #Did not implement. See section 3 in report
  return unigram

def main(argv):
  fr = open('Dish-csv.tsv', 'r')
  for line in fr:
    s = line.split('\t')
    dish_name = s[COL_NAME]
    for unigram in dish_name.split(' '):
      if not isAscii(unigram):
        #Put unigram to be translated in dictionary
        translated_unigram_mapping[unigram] = ''
        #Put dish name to be translated in dictionary
        translated_dish_name_mapping[unigram] = ''
```

```
  fr.close()

  for unigram in translated_unigram_mapping:
    translated_unigram_mapping[unigram] = google_translate(unigram)

  for dish_name in translated_dish_name_mapping:
    eng_dish_name = ''
    for unigram in dish_name.split(' '):
      if unigram in translated_unigram_mapping:
        eng_dish_name+=' '+translated_unigram_mapping[unigram]
      else:
        eng_dish_name+=' '+unigram
    translated_dish_name_mapping[dish_name] = re.sub('^ ', '' , eng_dish_name)

  fw = open('alternate_clean_Menu.tsv', 'w')
  fr = open('Dish-csv.tsv', 'r')
  for line in fr:
    new_line = ''
    s = line.split('\t')
    for i in range(0,len(s)):
      if i==1:
        dish_name = s[i]
        if dish_name in translated_dish_name_mapping:
          new_line+='\t'+dish_name+'\t'+translated_dish_name_mapping[dish_name]
        else:
          new_line+='\t'+dish_name+'\t'+dish_name
      else:
        new_line+='\t'+s[i]
    fw.write(re.sub('^\t', '', new_line))
  fw.close()
  fr.close()

if __name__ == "__main__":
  main(sys.argv)
```

In the script above, the implementation of google_translate function was not complete. It was because while manually hitting https://translate.google.com through a browser is free, using Google Cloud Translate API programmatically is not. It is a paid web service that costs $20 per 1,000,000 characters to translate. There are python libraries, however, that worked around by attempting to send requests to translate.google.com programmatically. Googletrans, https://pypi.python.org/pypi/googletrans, is one of those libraries. But we found they are not legitimate and not allowed by Google. So we did not try them and continued to explore other options. Bing also has a translate API but it also a paid web service. In the end, there wasn't any free web service for translation.

Without a free translation web service didn't stop us, we started manually using https://translate.google.com to look up the non-English unigram set. After going through about a hundred of them, we quickly realized many non-English unigram, about 7 in 10, are places or names that have no direct English translation. For example, almadèn refers to a town in Spain; allgäuer is a town in Germany; fleischkäs is a special dish in Germany and Austria, and it is similar to but is not exactly bologna sausage. zürichoise is a special dish in Zurich and has no

English name. At this point, we found no value in having english dish names in the dataset and gave up this attempt.

4. Develop a relational database schema (3 days)

4.1 Database Language

In order to create the the relational database schema of our cleaned data we used MySQL, NYPL_Dataset.db.

4.2 Structure of the Database

The first relation is **Menu** (Menu.sql) In this relation each tuple represents a menu. Each menu in the relation is associated with some number of the MenuPage values. The attributes are as follows:

- id - is of type int and it uniquely identifies the record. This is also a primary key for the relation.
- name - is of type varchar which is seemingly a duplicate of the sponsor attribute. We witnessed very many null values for this.
- sponsor - is of type varchar which represents the place sponsoring the menu.
- event - is a varchar attribute which describes the event that the menu was created for.
- venue - is of type varchar and represents the type of location, commercial or social.
- place - is of type varchar and it represents the location of the sponsor
- physical_description - is of type varchar and it describes the physical appearance of the menu.
- Dimension - is a varchar and this attribute describes the dimensions (in inches) of the menu.
- notes - is a varchar and this provides details about the menu card.
- call_number - is of type varchar and it describes how the menus are arranged by the NYPL. This is presumably similar to the traditional call number for books at a library.
- keywords - an empty field
- language - an empty field
- date - the date of the menu in formatted as MM/DD/YY
- location - of type varchar is the location of where the menu is being served. This is a duplicate of the sponsor attribute.
- location_type - an empty field.
- currency - is a varchar type and it represents the field of currency that the food on the menu can be paid for.
- currency_symbol - is of type varchar and it represents the symbol of the currency.
- status - is of type varchar and it represents whether the menu entry is complete or not.
- page_count - is an int that represents the number of dishes of the menu.
- dish_count - is an int that represents the number of dishes in the menu.

The second relation is **Dish**, (Dish.sql). This relation covers a number of items that are listed in the MenuItems relation. Dish represents the dishes that appeared on the menu and its attributes are as follows:

- id - is an int that stores the id of the entry. This is also a primary key for the relation.
- name - of type varchar is the name of the dish.
- description - of type varchar stores a description of the dish.
- menus_appeared - is of type int that represents the number of menus the dish appears on.
- first_appeared - is an int type that represents when the dish first appeared in a menu.
- last_appeared - is an int type that represents when the dish last appeared in a menu.
- lowest_price - is a floating point value that represents the lowest offered price of the dish.
- highest_price - is a floating point value that represents the highest offered price of the dish.

The next relation is the **MenuItem** (MenuItem.sql) relation. This relation links a MenuPage to a Dish via the menu_page_id and the Dish.id. The attributes are as follows:

- id - is of type int and it uniquely identifies the record. This is also a primary key for the relation.
- menu_page_id - is of type int and it identifies the menu page of where this item appears. This can be a key to the MenuPage relation.
- price - is a floating point number that identifies the price of the item.
- high_price - is a floating point value that represents the price of the largest portion of this menu item..
- dish_id - is of type int and it is used to identify the dish in the Dish relation.
- created_at - is a varchar that shows the timestamp of when an entry is created.
- updated_at - is a varchar that shows when the database entry was last updated.
- xpos - is a floating point value that represents the x-axis position of the item on the menu.
- ypos - is a floating point value that represents the y-axis position of the item on the menu.

The next relation is the **MenuPage** (MenuPage.sql) relation. This relation represents the menu that the that the item comes from as described by the menu_id attribute. The menu_id attribute refers to the Menu.id.

- id - is of type int and it uniquely identifies the record. This is also a primary key for the relation.
- menu_id - is of type int and it represents the id of menu that this menu page belongs to.
- page_number - is of type int and it represents the page number in the menu.
- image_id - is a varchar that identifies the scanned image of this menu page.
- full_height - is of type int and represents the height of the menu.
- full_width - is of type int and represents the width of the menu.

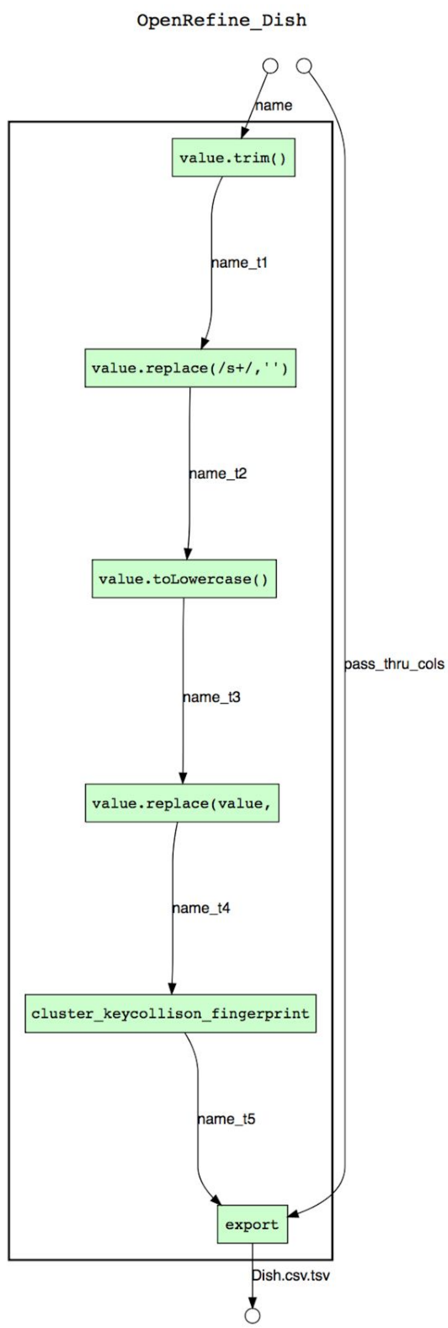
❖ uuid - is a unique identifier of type varchar and it represents 5 fields of alpha-numeric values.

5. Create workflow models (1 day)

5.1 Workflow of Using OpenRefine to clean Dish.csv

- Key input: Dish.csv
- Key output: Dish-csv.tsv
- Link to the file to generate workflow: openrefine_dish_workflow.py
- Link to the generated Workflow graph: openrefine_dish_workflow.svg
- Command to generate this graph is

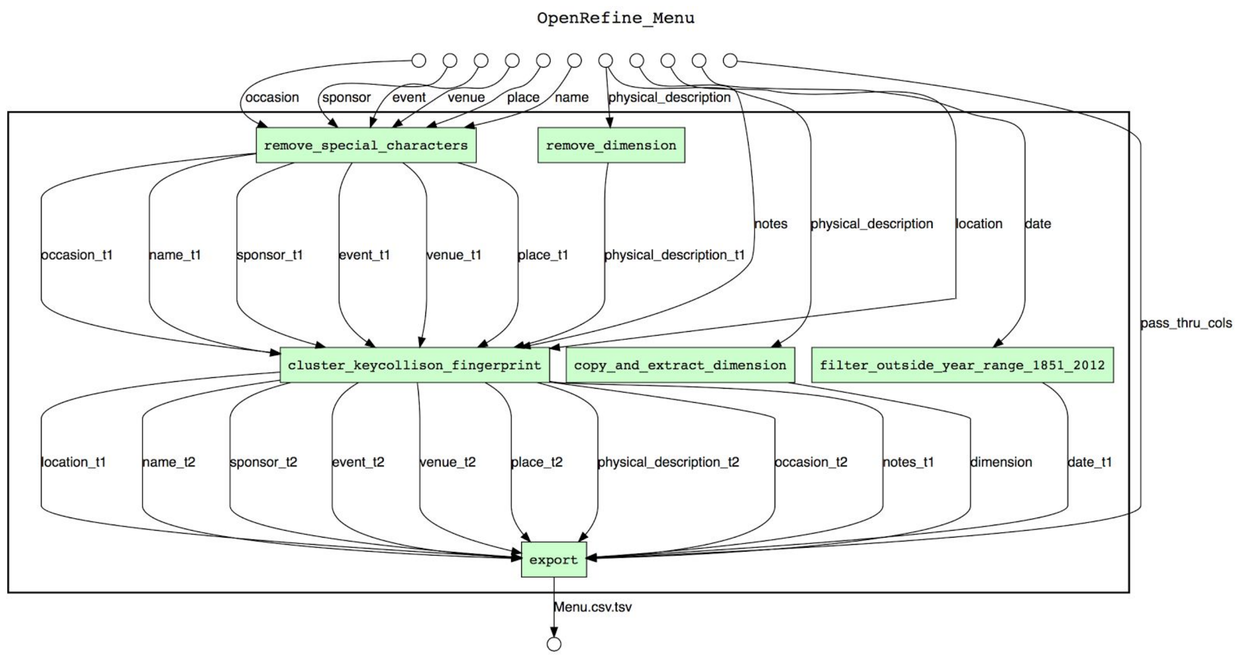
```
yw graph -c graph.layout=TB openrefine_dish_workflow.py > openrefine_dish_workflow.gv; dot
-Tsvg openrefine_dish_workflow.gv -o openrefine_dish_workflow.svg
```



5.2 Workflow of Using OpenRefine to clean Menu.csv

- Key input: Menu.csv
- Key output: Menu-csv.tsv
- Link to the file to generate workflow: [openrefine_menu_workflow.py]
- Link to the generated Workflow graph: [openrefine_menu_workflow.svg]
- Command to generate this graph is

```
yw graph -c graph.layout=TB openrefine_menu_workflow.py > openrefine_menu_workflow.gv; dot
-Tsvg openrefine_menu_workflow.gv -o openrefine_menu_workflow.svg
```

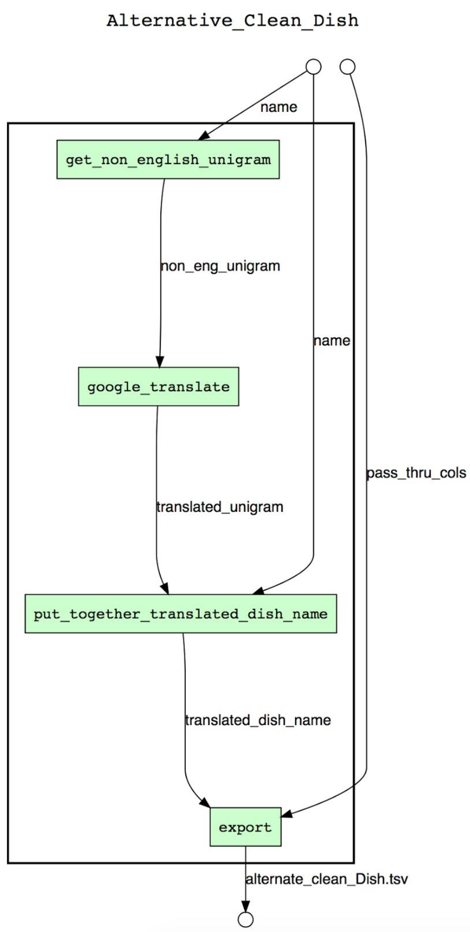


5.3 Workflow of Using Python to add column of English dish names

Although we did not succeed adding the column of English dish names, our python code was added with YesWorkFlow headers to illustrate the steps.

- Key input:Dish-csv.tsv
- Key output: alternate_clean_Dish.tsv
- Link to the file to generate workflow: alternate_refinement_dish_workflow.py
- Link to the generated Workflow graph: alternate_refinement_dish_workflow.svg
- Command to generate this graph is

```
yw graph -c graph.layout=TB alternate_refinement_dish_workflow.py >
alternate_refinement_dish_workflow.gv; dot -Tsvg alternate_refinement_dish_workflow.gv -o
alternate_refinement_dish_workflow.svg
```



6. Conclusion

The NYPL Meenu Dataset is crowd-sourced and is ideal for this data cleaning project. There are many use cases and its size is manageable by OpenRefine with increased heap size. In step 3, we attempted creating a dish name column that has the translated name for dishes that are non-English. To do that, we needed to rely on a web service. But in the end, we found that there was no translation for many non-English unigrams. The important takeaway in this step was learning how we detected and extracted non-English unigrams from dish names. This final project allowed us to clean the NYPL dataset and setup an end-to-end pipeline to integrate OpenRefine, regex, YesWorkFlow, SQLite. Since the NYPL Menu Dataset is updated bi-monthly interesting, future project might be to automatically grab these updates, clean them and add these changes into your database so as to maintain this in real time. We believe that would be a nice database to maintain.