

CS513-Data Cleaning Project Phase-2

Changsoo Kim ck37@illinois.edu, Ying-Chen Lee yclee6@illinois.edu

Phase-1 Change report:

1. Points received for initial Phase-1 submission: 100 (out of 100)
2. Our team chooses to improve the Phase-1 report.
3. The changes list is as follows:

Modifications	Descriptions
In our use case (see 2.A), we replaced the year “1914” with “1897” .	We changed our use case a little bit because when we were cleaning the data, we found out that most of the menus in New York City were created before 1910. As a result, we cannot answer the question “How much has the popularity of the chicken menu changed in New York City after the outbreak of World War I?”. We changed the time interval a little bit to answer another question “How much has the popularity of the chicken menu changed in New York City after the industrialization of chicken in 1897? Did the price of the chicken menu rise or decrease?”
We added the type of each field in the ER-diagram (see 3.B).	We looked into all the fields and found the correct data types. We updated that information to easily understand the rest process.
In the MySQL query (see 2.A), we used INNER JOIN instead of LEFT JOIN.	INNER JOIN helps us keep the records that exist in both two merged tables. For example, if a record contains only menu_id but no menu_page_id, that record would be useless.

Table 1. Phase-1 changes list.

Phase-1:

1. Identify a dataset

We plan to analyze the dataset from the New York Public Library (NYPL) (<http://menus.nypl.org/>), which transcribed historical restaurant menus and dishes. However, even though it contains useful data, one needs to correct some errors and merge several tables in the dataset in order to acquire this specific use case.

2. Development of a target use case U1

A. Target use case (U₁)

- U₁: How much has the popularity of the chicken menu changed in New York City after the **industrialization of chicken in 1897?** Did the price of the chicken menu rise or decrease?

- To answer the first question, we need to get the average number of chicken dishes of each menu before and after the chicken industrialization (started around 1896), i.e. to get $\frac{\# \text{ of menu items containing "Chicken" in 1886-1896}}{\text{Total \# of menus in 1886-1896}}$ and $\frac{\# \text{ of menu items containing "Chicken" in 1897-1907}}{\text{Total \# of menus in 1897-1907}}$
- To answer the second question, we need to get the average price of chicken dishes before and after the chicken industrialization, i.e. to get $\frac{\text{Sum of price of menu items containing "Chicken" in 1886-1896}}{\# \text{ of menu items containing "Chicken" in 1886-1896}}$ and $\frac{\text{Sum of price of menu items containing "Chicken" in 1897-1907}}{\# \text{ of menu items containing "Chicken" in 1897-1907}}$.
- Assuming the tables were cleaned and some columns were added with the steps described in Section 5: Device an initial plan, we use the following SQL queries to illustrate this use case:

(1) Q₁: Create a view “nyc_chicken” by merging four tables

```
CREATE VIEW `nyc_chicken` AS
SELECT
    M.date as menu_date,
    D.name as dish_name,
    MI.price,
    M.to_dollar_rate as dollar_rate,
    M.is_in_nyc as is_nyc,
    D.is_chicken as is_chicken
FROM
    menu AS M
    INNER JOIN menu_page AS MP ON M.id = MP.menu_id
    INNER JOIN menu_item AS MI ON MP.id = MI.menu_page_id
    INNER JOIN dish AS D ON MI.dish_id = D.id
WHERE
    M.status='complete'
    AND M.is_in_nyc='Y'
    AND M.date IS NOT NULL;
```

(2) The merged view should look like this:

menu_date	dish_name	price	dollar_rate	is_nyc	is_chicken
1900-04-00	st. julien	1	1	Y	N
1901-04-00	roast chicken	1	1	Y	Y
1900-03-00	planked shad	1	1	Y	N
1905-06-00	milk fed chicken	3	1	Y	Y
1900-04-00	malaga grapes	1	1	Y	N

Table 2. Target merged table.

(3) Q₂: check chicken menu percentage & average price in 1887-1896

```
SELECT is_chicken, count(1) count, AVG(dollar_rate*price) as avg_price
FROM nyc_chicken
WHERE menu_date >= '1887-01-01' and menu_date < '1897-01-01'
GROUP BY is_chicken;
```

(4) Q3 : check chicken menu percentage & average price in 1897-1906
 SELECT is_chicken, count(1) count, AVG(dollar_rate*price) as avg_price
 FROM nyc_chicken
 WHERE menu_date >= '1897-01-01' and menu_date < '1907-01-01'
 GROUP BY is_chicken;

- All the queries uploaded in **MySQL_DDL.txt, Queries.txt**.

B. Use cases requires zero data cleaning (U_0)

The original NYPL's data could be useful without any data cleaning process as belows:

- The most popular dish throughout history. Even though the date field in the Menu.csv has 0001-01-01 and 2928-03-26 values, it doesn't matter in this limited case.
- Find "under review" menus percentage by checking Menu's status field
- Popular hotel/restaurant brands count ranking: There are a lot of dirty values so we cannot separate our analysis section, but we just make a conclusion throughout the whole dataset.

C. Use cases requires more than data cleaning (U_2)

Even if our dataset has no NaN, null, invalid values, and additional information, we still cannot solely use this dataset for the following use cases:

- When was the Chinese food first introduced in American restaurants?
 - need food category information
- Analysis of seafood restaurant location
 - need dish gradient information. We don't know whether a specific restaurant is counted as a seafood restaurant because we don't know whether the menu items it supplies contain seafood.
- Analysis of seasonal menu changes
 - need dish menu season information
- How does the economy affect the dish prices?
 - need economy conditions information year by year

3. Describe the dataset

A. Description of the dataset

We are going to use NYPL menus provided on the campuswire. It has 4 tables - Menu, MenuItem, MenuPage, Dish. They have a hierarchical structure from Menu to Dish.

	rows	columns	description
Menu.csv	17,550	20	Top level menu information
MenuPage.csv	66,938	7	Menu page info. in menu
MenuItem.csv	1,334,780	9	Menu item info. in menu page
Dish.csv	428,086	9	Dish info. in menu item

Table 3. Tables list in the dataset.

B. ER-diagram (using Chen ER-diagram notation)

The following figure is the ER-diagram of this dataset. We keep only those useful attributes for our use case to make this diagram more concise. As shown in the diagram, each menu contains several menu pages, and each menu page contains several menu items. Different menu items can have the same dish name, and therefore are mapped into the same dish.

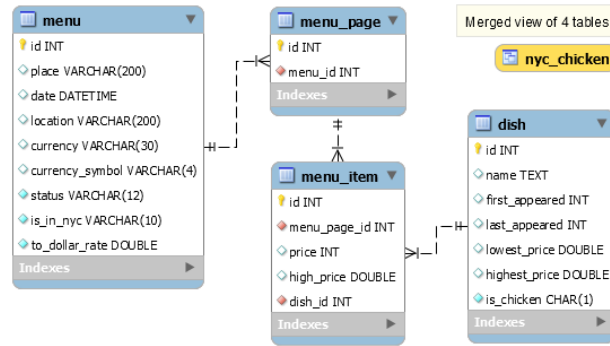


Figure 1. ER-diagram.

C. Meaning of attributes

(1) Menu:

Each tuple contains the information of the menu offered by a specific restaurant at a specific time. Menu.place is the location of that restaurant. Menu.status indicates whether the record of this menu is still under review or is complete. Menu.date is the time of the menu. Menu.currency is the currency this menu uses such as Dollars or Franc.

(2) MenuPage:

One menu might contain more than one page. This table describes which menu the specific menu page comes from.

(3) MenuItem:

Each tuple contains the information of the menu item, such as which kind of dish this item refers to and the price of this menu item.

(4) Dish:

Each tuple is a kind of dish, such as chicken soup, Strawberries, and mashed potatoes.

4. List obvious data quality problems (S2)

A. Data Cleaning is “sufficient” to implement U_1 :

1. We have the year of each menu, so we are able to compare the results before and after the outbreak of WWI.
2. We have the price of each menu item, so we can calculate the difference of average price of chicken dishes.
3. We have the place of the restaurant on each menu, so we can capture those in New York City.

B. Data Cleaning is “necessary” to implement U_1 :

1. Menu.csv

- date: it contains invalid format (e.g.: 0001-01-01, 0190-03-06, 2928-03-26)

- currency: We need to exchange different currencies into Dollars, so we need the exchange rate of different kinds of currency. We are going to add a new attribute “Menu.toDollarRate” according to different kinds of currency.
- place, location: Ideally, in our use case U_1 , place attribute should only show the city name with only one format, so that we can easily get the restaurant located in the New City with the command “WHERE place=”New York”. However, the format is quite different, such as “NEW YORK, NY”, “75 ST. & COLUMBUS AVE. NY”, and “SARATOGA SPRINGS,NY”. If we cannot find a key of NYC, we can also reference a location field that has similar information as place. However, the location field has several variations like Childs, Child’, Child’s, so we are going to cluster these facets.

2. Dish.csv

- name: need to normalize by making lower, clustering and remove unnecessary special characters (e.g: backslash)
- name: need to categorize dishes into “Chicken” and “Non-Chicken”, and add this result into a new attribute “Dish.is_chicken”.

5. Devise an initial plan

5.1(S1) ~ 5.2(S2) is already described in the above sections.

5.3 (S3) tools and specific data cleaning steps

We are going to use tools below:

- OpenRefine to:
 - collapse consecutive white spaces
 - trimming leading and trailing white spaces
 - clustering text facets to remove duplicate entries
 - keep consistent format by converting lower case or upper case
 - re-format date
- Python to:
 - convert currency
 - categorize dish
 - decide whether the restaurant is situated in NYC
- MySQL to:
 - check integrity constraints (menu > menu page > menu item > dish)
 - create tables from the cleaned dataset and then merge them based on the steps mentioned in Section 2.A: Target use case(U_1).

5.4 (S4) measure data quality / inspect ICs to document that the data cleaning steps in (S3) “worked”

Our question is “How much has the popularity of the chicken menu changed in New York City after the [industrialization of chicken in 1897?](#) Did the price of the chicken menu rise or decrease?”. We confirmed why D’ is suitable for U_1 .

- after categorizing dishes into “Chicken” and “Non-Chicken”: we can retrieve the menu items which contain chicken.
- after year correction: we can check the period
- after currency standardizing: we can check whether the price increases or decreases.
- after location checking: we can narrow down our interest into the NYC area
- after merging tables based on the MYSQL queries: we can remove menus which are still under review and remove menu items which contain NULL value.

5.5 (S5) document the types and amount of changes

Dish

- (new) category column (Chicken,Non-Chicken)

Menu

- place, location: clustering text facets, collapse/trimming white spaces, text formatting
- date: check ICs, formatting, remove records that is null
- (new) is_in_nyc: check if the restaurant is in NYC

MenuPage

- menu_id: remove records that violates the FK constraints

MenuItem

- price: remove records that has null price
- dish_id: remove records that violates the FK constraints

6. Timeline & Tentative assignments of tasks to team members

We will do the Phase-I and Phase-II summarizing tasks together and separate the others like below:

- (OpenRefine) collapse / trimming leading and trailing white spaces: CK
- (OpenRefine) clustering text facets to remove duplicate entries: CK
- (OpenRefine) keep consistent format by converting upper case: YC
- (Python) categorize dish: YC, CK
- (Python) decide whether the restaurant is situated in NYC: YC
- (Python) convert currency: CK
- (MySQL) inspect IC: YC
- (MySQL) check query result if sufficient for our use case: CK

Tasks		7.4~7.10	7.11~7.17	7.18~7.24	7.25~7.31
Phase-I	(S1) Identify a dataset	ALL			
	(S1) develop a target use case U1				
	(S1) describe the dataset D				
	(S2) list data quality problems				
	(S3) tools and specific data cleaning steps				
	(S4) measure data quality / inspect lcs				
Phase-II	(S5) document the type and amount of changes				
	Execute the data cleaning tasks		CK, YC		
	Update the Phase-I report				
	Include a narrative of all steps and methods				
	Document about the improved data quality			CK, YC	
	Summarize the data changes				
	Summarize the project results				
	Attach technical artifacts				ALL

Figure 2. Tasks and timeline.

Phase-2:

1. Data Cleaning Performed

1.1 Profile Data and Remove Unnecessary Fields (columns) with Python

As shown in the Phase-1 report, our use case ([see 2.A in Phase-1 report](#)) is to answer the question: How much has the popularity of the chicken menu changed in New York City after the industrialization of chicken in 1897? Did the price of the chicken menu rise or decrease?

To answer these two questions, we need only part of the fields in the dataset (needed fields were listed in [2.A in Phase-1 reports](#)). Even though this step is actually not necessary, it helps us focus on the question we are interested in, reduce disk storage, and make queries faster. We removed keywords, language, location_type fields in Menu.csv, and removed a description field in Dish.csv.

Menu	N/A	MenuPage	N/A	MenuItem	N/A	Dish	N/A
id	0%	id	0%	id	0%	id	0%
name	82%	menu_id	0%	menu_page_id	0%	name	0%
sponsor	9%	page_number	2%	price	33%	description	100%
event	54%	image_id	0%	high_price	93%	menus_appeared	0%
venue	54%	full_height	0%	dish_id	0.002%	times_appeared	0%
place	54%	full_width	0%	created_at	0%	first_appeared	0%
physical_description	16%	uuid	0%	updated_at	0%	last_appeared	0%
occasion	78%			xpos	0%	lowest_price	7%
notes	40%			ypos	0%	highest_price	7%
call_number	9%						
keywords	100%						
language	100%						
date	3%						
location	0%						
location_type	100%						
currency	63%						
currency_symbol	63%						
status	0%						
page_count	0%						
dish_count	0%						

1.1.1 Remove Unnecessary Fields

We used the package called “pandas” to load the csv files of the dataset and drop unnecessary fields. Remaining fields were shown in Figure 3. We used the **read_csv()** function to load the csv files and used the **drop()** function to drop unnecessary fields.

Here are the unnecessary fields:

- Menu: name, sponsor, event, venue, physical_description, occasion, notes, call_number, page_count, dish_count
- MenuPage: page_number, image_id, full_height, full_width, uuid
- MenuItem: created_at, updated_at, xpos, ypos
- Dish: menus_appeared, times_appeared

Menu	N/A	MenuPage	N/A	MenuItem	N/A	Dish	N/A
id	0%	id	0%	id	0%	id	0%
name	82%	menu_id	0%	menu_page_id	0%	name	0%
sponsor	9%	page_number	2%	price	33%	menus_appeared	0%
event	54%	image_id	0%	high_price	93%	times_appeared	0%
venue	54%	full_height	0%	dish_id	0.002%	first_appeared	0%
place	54%	full_width	0%	created_at	0%	last_appeared	0%
physical_description	16%	uuid	0%	updated_at	0%	lowest_price	7%
occasion	78%			xpos	0%	highest_price	7%
notes	40%			ypos	0%		
call_number	9%						
date	3%						
location	0%						
currency	63%						
currency_symbol	63%						
status	0%						
page_count	0%						
dish_count	0%						

1.1.2 Check Number of N/A in Each Field

We check the percentage of N/A in each field to make sure the remaining fields are large enough for us to do the analysis or do the rest of the cleaning steps. For example, if all the fields are 99% N/A, we cannot get a meaningful result. Through this step, we can get a sense of the shape of the fields. We used **isnull()** and **sum()** functions to count the number of N/A.

Even though there are more than 50% of N/A in Menu.place and Menu.currency, we think it is acceptable. For Menu.place, we can get the locations with both Menu.place and Menu.location, so even though there are many N/A in Menu.place, we could still use Menu.location to check the location of the restaurants. For Menu.currency, since the default currency is dollar, we could just replace N/A with 'dollar'.

We found out that MenuItem.dish_id has 0.002% N/A values. In order to meet the foreign key constraints in our use case (each menu item should have a dish name), we just removed those records. The result was shown in Figure 3, and the output data was in the **NYPL_only_necessary_columns.zip** file.

Menu	N/A	MenuPage	N/A	MenuItem	N/A	Dish	N/A
id	0%	id	0%	id	0%	id	0%
place	54%	menu_id	0%	menu_page_id	0%	name	0%
date	3%			price	33%		
location	0%			dish_id	0%		
currency	63%						
currency_symbol	63%						
status	0%						

Figure 3. Remaining fields in the dataset.

1.2 Perform Multiple Cleaning Steps with OpenRefine

1.2.1 Check Field Types and Length of Longest String of Each Text Field

MySQL has various data types, such as NUMBER, VARCHAR, TEXT, DOUBLE, JSON, etc. So we first need to check what the right data type is for the MySQL table. On top of that, we need to check the maximum length of the string in order to set the capacity of the VARCHAR type.

We checked the field types, and if it is a text type, check the length of the longest string by using GREL's length function (See [Appendix.1](#) for screenshots and details). Figure 4 shows a final version of field types and the length of the text fields.

Menu	Type	Len	N/A	MenuPage	Type	N/A	MenuItem	Type	N/A	Dish	Type	Len	N/A
id	Int		0%	id	Int	0%	id	Int	0%	id	Int		0%
place	Text	106	54%	menu_id	Int	0%	menu_page_id	Int	0%	name	Text	1387	0%
date	Date		3%				price	Double	33%				
location	Text	127	0%				dish_id	Int	0.002%				
currency	Text	26	63%										
currency_symbol	Text	4	63%										
status	Text	12	0%										

Figure 4. Length of longest string of each text field.

1.2.2 Remove Backslash Characters for MySQL Data Loading

While loading data into MySQL, error occurs if there are trailing backslash (\) chars. Take the following record in Menu.csv as an example:

27966,1974-11-01,"Rms \Queen Elilizabeth 2\""...complete

We got an error: Error Code: 1261. Row 10480 doesn't contain data for all columns

Therefore, we replaced every backslash in the text columns with an empty string using GREL's command — **value.replace(/\\,")**. Note that the two forward slashes indicate that the inner string is a regular expression, and the \\ means a backslash character.

1.2.3 Check Currency and Currency Symbols

Likewise the previous steps, this step is to get the sense of the currency fields. We should first check the data shapes so that we can get the right strategy. Here, we checked currency and currency symbols.

By using text facet features in OpenRefine, we can get the gist of the data properties. Menu.csv has the following currency properties.

- currency: 42 choices (All choices are attached to a “currency_types.txt” file.)
- currency_symbol: 34 choices (All choices are attached to a “currency_symbol_types.txt” file.)

We found one critical limitation of OpenRefine here. Even though the “text facet” feature is very useful for one column, it cannot mix more than one column. For example, “Francs” appears 162 times, but we cannot be sure what it really means since there are many currencies including “Franc” like below.

francs
BEF – Belgian Franc (obsolete)
LUF – Luxembourg Franc (obsolete)
MGF – Malagasy Franc (obsolete)
FRF – French Franc (obsolete)

However, every currency symbol of “Francs” is “FF”; so it would be nice if OpenRefine provides text faces features for multiple columns. Anyhow, we can get all the currency rates without big problems as we have only less than 50 currency types in our project.

1.2.4 Check Date Fields in Menu.csv

We checked the Menu.date field by using OpenRefine text facet features. Although there are several invalid dates, such as 0001-01-01, 0190-03-06, 1091-01-27, and 2928-03-26, we just left that date as it is. This is because there are only 5 cases and we cannot guess what it means. In addition, it doesn’t matter in our use case since we would automatically filter these cases out when we query specific time intervals with MySQL query commands such as **WHERE menu_date >= '1887-01-01' and menu_date < '1897-01-01'**.

1.2.5 Polish up on Every Text Fields

When comparing strings, letter case and whitespace matter. To make comparison easier, we applied several built-in operations in OpenRefine to every text field. The operations include **To lowercase** , **Trim leading and trailing whitespace**, and **Collapse consecutive whitespace**.

1.2.6 Cluster & Merge on Place and Location Columns

We can see whether the restaurants were in NewYorkCity or not by examining Menu.place and Menu.location. However, there are a lot of variants that might represent the same thing especially for NewYorkCity. Without examining them and merging them, we cannot get all records that were in NewYorkCity.

For example, “**delmonico’s ny**” and “**delmonico’s [ny]**” should be the same thing, so we have to merge them. However, there are some strings that seem to be the same thing but we cannot be sure. For example, “**jersey city, nj**” might stand for “New Jersey” and “**jersey city, ny**” might stand for “New York”. In this case, we didn’t merge them.

From the two previous examples, we found that to fix this problem, the only way is to examine them one by one manually. Fortunately, OpenRefine offers easy-to-use built-in functions to help us capture all strings which look the same. ([See Appendix.2](#) for screenshots and details)

Since different clustering methods suggest different items, we applied fingerprint, metaphone3, levenshtein, Beider-Morse in order.

1.2.7 Export Refined Data to .csv Files

After finishing OpenRefine tasks, export all the data to csv files: **After_OpenRefine.zip**

1.3 Add New Fields (Menu.is_in_nyc, Menu.to_dollar_rate, and Dish.is_chicken) with Python

1.3.1 Chicken Classification (Dish.name => Dish.is_chicken)

We searched chicken dish names and decided to use the below website as a reference:

https://en.wikipedia.org/wiki/List_of_chicken_dishes

It contains 204 chicken dish names but we don't know which part of the name means chicken. Our intuition is that all of the names must have a word that means chicken. For example, "Pollo a la Brasa" may mean "Peruvian Roast Chicken" and "Pollo" is usually used in the names of Italian, Spanish, or Mexican dishes according to google dictionary

(<https://www.google.com/search?q=pollo+meaning+in+english>). This also means that frequently used words are likely to have a meaning of chicken.

This is a pseudocode of chicken classification:

1. Filter out dishes that have a name including "chicken". Since we will use "chicken" as a keyword, we want to find out other candidates here.
2. Word count in dish names
3. Filter out words appeared only 1 time. This is because if the word occurs only one time, it should not be a popular name or it may be just an adjective or a preposition. For sure, there are some exceptional cases like coq, poulet, karaage, etc. That's why we add up popular names later.
4. Filter out known stop words (e.g.: au, wing, shish, biryani, etc.)
5. Append well-known chicken dish names

```
# List of chicken dishes: https://en.wikipedia.org/wiki/List_of_chicken_dishes
chicken_dishes = pd.read_csv("chicken_dishes.txt")
chicken_idx = chicken_dishes["name"].str.lower().str.contains("chicken")
chicken_variants = chicken_dishes[~chicken_idx]
chicken_var_words = chicken_variants.name.str.lower().str.split(expand=True).stack().value_counts()
chicken_words = chicken_var_words[chicken_var_words > 1]
chicken_words = chicken_words.drop(['à', 'au', 'wing',
                                   'shish', # skewer (a long piece of wood or metal used for holding pieces of food)
                                   'biryani', # Indian fried rice
                                   'adobo', # Philippines's cooking technique, not a dish name
                                   'nasi', # Indonesian rice
                                   'kai', # Thailand soup
                                   ], axis=0) # delete the rows with some labels
chicken_words = chicken_words.append(pd.Series({'chicken': 100, 'coq':1, 'poulet':1, 'poussin': 1, 'poulette': 1, 'karaage': 1, 'yassa': 1}))
chicken_words = chicken_words.sort_values(ascending=False)
print(chicken_words)
# print(chicken_words.index.values)
```

```
chicken    100
ayam       14
pollo       3
manok       3
galinha     2
taouk       2
yassa       1
karaage     1
poulette    1
poussin     1
poulet      1
coq         1
dtype: int64
```

```
# https://stackoverflow.com/questions/53350793/how-to-check-if-pandas-column-has-value-from-list-of-string
chicken_checks = dish.name.apply(lambda x: any([k in x for k in chicken_words.index.values]))
chicken_rows = dish.name[chicken_checks == True]
chicken_rows.describe()
```

```
count          19431
unique          18097
top    broiled spring chicken
freq              7
Name: name, dtype: object
```

```
chicken_rows.head()
```

```
1      chicken gumbo
6      chicken soup with rice
150    chicken broth, per cup
172    chicken broth
201    chicken salad
Name: name, dtype: object
```

```
dish["is_chicken"] = chicken_checks
dish.head()
```

	id	name	description	menus_appeared	times_appeared	first_appeared	last_appeared	lowest_price	highest_price	is_chicken
0	1	consomme printaniere royal	NaN	8	8	1897	1927	0.20	0.4	False
1	2	chicken gumbo	NaN	111	117	1895	1960	0.10	0.8	True
2	3	tomato aux croutons	NaN	14	14	1893	1917	0.25	0.4	False
3	4	onion au gratin	NaN	41	41	1900	1971	0.25	1.0	False
4	5	st. emilion	NaN	66	68	1881	1981	0.00	18.0	False

1.3.2 NYC detection (Menu.is_in_nyc)

In this project, our intuition is that if there is one of the words, “ny”, “nyc”, and “new york” in Menu.location or in Menu.place, that restaurant would be situated in NYC. There might be several exceptional cases, and there would be several ways to tackle this problem. However, after skimming through Menu.place and Menu.location, we think that this is the best way to our best knowledge.

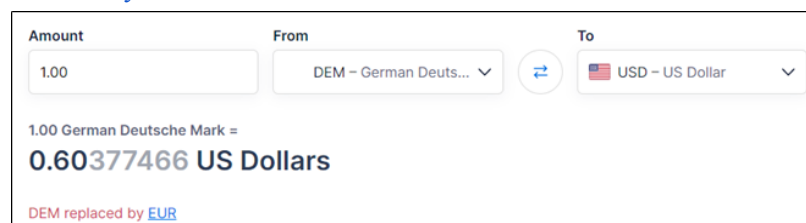
We use the following functions in Python to see whether a record in Menu contains “ny”, “nyc”, or “new york”:

```
menu_after_openrefine["is_in_nyc"] =  
menu_after_openrefine["place"].str.lower().str.contains(r'\b(ny|nyc|new york)\b', regex=True) \  
| menu_after_openrefine["location"].str.lower().str.contains(r'\b(ny|nyc|new york)\b', regex=True)
```

Note that the strings inside r” are regular expressions, \b means word boundary. Now, we have a new field Menu.is_in_nyc indicating whether that restaurant is in NYC or not.

1.3.3 Currency Standardizing (Menu.to_dollar_rate)

Basically, we will get the currency rate from a website: <https://www.xe.com/currencyconverter/>. For example, one can get a currency rate of German Deutsche Mark to US dollars with a below link: <https://www.xe.com/currencyconverter/convert/?Amount=1&From=DEM&To=USD>

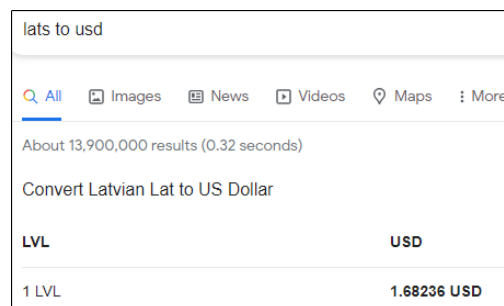


The screenshot shows the XE.com currency converter interface. The 'Amount' field is set to 1.00. The 'From' dropdown is set to 'DEM - German Deuts...'. The 'To' dropdown is set to 'USD - US Dollar'. The result displayed is '1.00 German Deutsche Mark = 0.60377466 US Dollars'. A note at the bottom states 'DEM replaced by EUR'.

This website (xe.com) also provides a Currency Data API (<https://www.xe.com/xecurrencydata/>).

However, we have only a few currencies and it costs money, so we will get the information manually.

In case of unclear currency strings, such as Lats (Ls), we choose the first match results from the Google search.



The screenshot shows a Google search result for 'lats to usd'. The search results show 'About 13,900,000 results (0.32 seconds)'. The first result is 'Convert Latvian Lat to US Dollar'. Below this, there is a table showing the conversion rate:

LVL	USD
1 LVL	1.68236 USD

As we know, all of these currency rates are the values as of now. There might be a lot of chances of dramatic changes of the value. However, since we failed to find the exchange rate at that time, and most of the currencies are dollar, we use today’s exchange rate as a compromise.

1.4 Load Data into MySQL, Check Integrity Constraints, and Merge Tables

After having all the jobs done, now we can load data into MySQL database. We are using local MySQL as a main database, and using LOAD DATA INFILE syntax to load csv file’s data. All the DDL (Data Definition Language), DML (Data Manipulation Language) queries are saved in **MySQL_DDL.txt**, **Queries.txt**, respectively.

1.4.1 Regarding N/A value handling strategy

On MySQL data loading, there is one more thing to consider. That is N/A values. We hope that after reading data with LOAD DATA, empty or missing columns will be regarded as NULL. However, if we don't handle it, it will be an empty string (") rather than NULL. Therefore, to load a NULL value into a column, we could use \N in the data file¹. For example,

Original data

12463,"HOT SPRINGS, AR",1900-04-15,Hotel Eastman,,,complete

To-Be data

12463,"HOT SPRINGS, AR",1900-04-15,Hotel Eastman,\n, \n, complete

However, there is a more intuitive way to handle this. We handled this by using Input Preprocessing, which means we don't need to replace N/A values in OpenRefine steps².

MySQL LOAD DATA Input Preprocessing example:

```
LOAD DATA INFILE 'Menu.csv'
INTO TABLE menu
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS
(id, @place, @date, @location, @currency, @currency_symbol, status, is_in_nyc, to_dollar_rate)
SET
  place = IF(@place = '', NULL, @place),
  date = IF(@date = '', NULL, @date),
  location = IF(@location = '', NULL, @location),
  currency = IF(@currency = '', NULL, @currency),
  currency_symbol = IF(@currency_symbol = '', NULL, @currency_symbol)
;
```

In this way, we could directly replace empty columns with NULL.

1.4.2 menu_page::menu_id Foreign Key Check

The foreign key constraint would be satisfied automatically after we merged Menu and MenuPage tables with the query—**INNER JOIN menu_page AS MP ON M.id = MP.menu_id** ([See Phase-1 report 2.A](#)) because only the records that have Non-NULL M.id and Non-NULL MP.menu_id would be in their merged table (NULL is not comparable). However, we want to make sure that the number of records which violate the foreign key constraint is acceptable. If there are too many records that violate it, we cannot get a meaningful result.

“menu_id” in the MenuPage table should correspond to the “id” in the Menu table. We will check this by using ADD CONSTRAINT syntax.

```
ALTER TABLE menu_page ADD CONSTRAINT FK_menu_page FOREIGN KEY (menu_id)
REFERENCES menu(id);
-- Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails
('cs513`.`#sql-1b34_25`, CONSTRAINT 'FK_menu_page' FOREIGN KEY ('menu_id')
REFERENCES `menu` ('id'))
```

¹ <https://dev.mysql.com/doc/refman/8.0/en/problems-with-null.html>

² <https://dev.mysql.com/doc/refman/8.0/en/load-data.html>

<https://stackoverflow.com/questions/2675323/mysql-load-null-values-from-csv-data>

However, it failed to add FK constraints; so we checked the reason.

```
select
  mp.id menu_page_id,
  m.id menu_id1,
  mp.menu_id menu_id2
from
  menu m
right join menu_page mp on (m.id = mp.menu_id)
where m.id is null;
-- 5799 rows returned
```

menu_page_id	menu_id1 (from Menu)	menu_id2 (from MenuPage)
119	NULL	12460
120	NULL	12460
...
5176	NULL	14136
...

There are no menu_ids (12460, 14136) in the original Menu.csv file. This means MenuPage includes invalid records that violate foreign key rules or there might be some missing records in the Menu file.

There are 5799 records that violate the foreign key constraint. However, it is only a small number compared to the total number of rows in MenuPage—66938.

By using the same method, we check the `menu_item::menu_page_id` ([see Appendix.3.1](#)) and `menu_item::dish_id` ([see Appendix.3.2](#)) foreign key constraints.

1.4.3 Merge Tables

This step has been discussed in [Phase-1 2.A](#) already. Here, we want to point some details out. The MySQL query to merge four tables is as follow:

```
CREATE VIEW `nyc_chicken` AS
SELECT
  M.date as menu_date,
  D.name as dish_name,
  MI.price,
  M.to_dollar_rate as dollar_rate,
  M.is_in_nyc as is_nyc,
  D.is_chicken as is_chicken
FROM
  menu AS M
  INNER JOIN menu_page AS MP ON M.id = MP.menu_id
  INNER JOIN menu_item AS MI ON MP.id = MI.menu_page_id
```

```

INNER JOIN dish AS D ON MI.dish_id = D.id
WHERE
    M.status='complete'
    AND M.is_in_nyc='Y'
    AND M.date IS NOT NULL;

```

The result table was shown in [Table 2](#). We merged the tables using INNER JOIN because we hope that all the records have non-NULL Menu.id, MenuPage.id, MenuItem.id, and Dish.id. We filtered out the records that are not in a complete menu, not in NewYorkCity, or do not have a date.

Now, this merge table should be clean enough for our use case.

2. Document Data Quality Changes

2.1 Summary Table of Changes³

Tasks	Number of cells per column changed
Remove unnecessary fields (Python)	All cells in unnecessary fields (listed in 1.1.1) were removed.
Remove backslash (OpenRefine)	Menu.place: 0 Menu.location: 167 Menu.currency: 0 Menu.currency_symbol: 0 Menu.date: 0 Dish.name: 76
To lowercase (OpenRefine)	Menu.place: 8,034 Menu.location: 17,499 Menu.currency: 6,456 Menu.currency_symbol: 738 Menu.status: 0 Dish.name: 416,292
Trim leading and trailing whitespace (OpenRefine)	Menu.place: 0 Menu.location: 0 Menu.currency: 0 Menu.currency_symbol: 0 Menu.status: 0 Dish.name: 1
Collapse consecutive whitespace (OpenRefine)	Menu.place: 45 Menu.location: 555

³ The total number of rows in each column was listed in [Table 3](#).

	Menu.currency: 0 Menu.currency_symbol: 0 Menu.status: 0 Dish.name: 1
Cluster & merge on place, location columns (OpenRefine)	Menu.place applied 4 methods (key collision / fingerprint): (Select All) 3184 (key collision / metaphone3): (Select All) 4396 Menu.place (key collision / levenshtein): (Partially Checked) 278 Menu.place (key collision / Beider-Morse): (Partially Checked) 27 Menu.location (key collision / fingerprint): 4209 Menu.location (key collision / metaphone3): 6069
Load data into MySQL (MySQL)	Menu.csv: 17,547 MenuPage.csv: 66,937 MenuItem.csv: 1,334,538 Dish.csv: 428,082
Remove records that violate foreign key constraints by INNER JOIN (MySQL)	MenuPage.csv: 5,799 MenuItem.csv: 3 Dish.csv: 12,752

2.2 Demonstrate the improvement of data quality

Due to the post analysis, our processes are not easy to estimate step by step. Therefore, for some steps that are hard to use queries to demonstrate the changes, we just described it. In 2.2.2, we tried to make some queries to show that some important steps truly improve the data quality.

2.2.1 Description of quality changes for each step

Tasks	Improvement of data quality
Remove unnecessary fields (Python)	By removing unnecessary fields, we can handle the tasks efficiently. In Python, we can run the script fastly. In MySQL we can save disk storage and memories, and also can get the result faster.
Remove backslash (OpenRefine)	While loading data into MySQL, the back-slash character means escape character. If we do nothing, this will result in loading failure. After removing backslash characters we can import our data into MySQL without problems.
To lowercase (OpenRefine)	In our use case, all the tasks are case-insensitive. After converting all the strings to lowercase, we now have standardized strings so we can compare directly without additional considerations.

Trim leading and trailing whitespace (OpenRefine)	In our use case, leading and trailing whitespace have no special meaning. After removing the leading and trailing whitespace, we now have standardized strings. Standardizing is a prior step for clustering tasks.
Collapse consecutive whitespace (OpenRefine)	In our use case, consecutive whitespaces have no special meaning. After removing consecutive whitespaces, we now have standardized strings. Standardizing is a prior step for clustering tasks.
Cluster & merge on place, location columns (OpenRefine)	In our data, there are many duplicated entries that have similar meaning, such as NYC, New York City, ny, etc. Having this step, we can standardize and regularize fields. Standardizing is a prior step for the NYC detection task.
Add Menu.is_in_nyc, Menu.dollar_rate, and Dish.is_chicken (Python)	We need to have augmented data in order to get the answer to our questions. This Data Integration step helps us answer our questions. We can improve the data quality by integrating necessary information.
Load data into MySQL (MySQL)	Using a database system, we can be ready for data profiling(queries), integrity constraints check, and repair.
Remove records that violate foreign key constraints by INNER JOIN (MySQL)	By removing IC(integrity constraints) violating records, now we can have the correct results for our use case.

2.2.2 Queries for Proving the Improvement of Data Quality

We could just use MySQL to load the uncleaned data and query the data we want. Here, we tried to load the uncleaned data and tried to answer the question in our use case. The DDL (Data Definition Language) queries for cleaned and uncleaned data are saved in **prove_quality_DDL.txt** and **prove_quality_DDL_uncleaned.txt** respectively. The DML (Data Manipulation Language) queries for cleaned and uncleaned data are saved in **prove_quality_DML.txt** and **prove_quality_DML_uncleaned.txt** respectively.

As for the issue mentioned in [Section 1.2.2](#) that we have to remove backslash otherwise error would occur when loaded into MySQL, we found a solution with command in MySQL. When using the LOAD DATA command, we could just add **ESCAPED BY '\b'** into it (see **MySQL_DDL_uncleaned.txt**). It will ignore the backslash character automatically.

2.2.2.1 Prove for currency steps

If we didn't get the exchange rate for different kinds of currencies, we could only use "Dollars". Here, we use the MySQL query to count the number of records that use "Dollars".

We get 1,154,265 of records that use “Dollars” or “NULL”(default is dollar), and the total number of records is 1,222,997. This means if we didn’t add the exchange rate with Python, we would lose 68,732 records.

2.2.2.2 Prove for is_nyc

If we didn’t use OpenRefine to check and cluster the place and the location fields, we need to match the string using regular expressions:

```
SELECT count(1)
FROM nyc_chicken
WHERE place REGEXP '.*(\\bny\\b|\\bnyc\\b|\\bnew york\\b).*'
OR location REGEXP '.*(\\bny\\b|\\bnyc\\b|\\bnew york\\b).*';
```

We use this command to match ny, nyc, and new york using regular expressions. Note that \\b stands for word boundary.

We get 161,826 records from this command, and we get 152,941 records from cleaned data (simply check **WHERE is_nyc='Y'**). This means if we didn’t clean the data, we would get more records. We have to admit that it violates our intuition. Further research should be done why this happened. Maybe when we were clustering manually, we changed some string and made them unrecognizable by the regular expression. For example, we might clustered “[ny]” into “[n.y.]”.

2.2.2.3 Prove for is_chicken

If we didn’t use Python to make the field “is_chicken”, we could just search “chicken” in dish.name field.

```
SELECT count(1)
FROM nyc_chicken
WHERE dish_name REGEXP 'chicken';
```

We get 38,362 dishes from uncleaned data and get 42,410 dishes by using “WHERE is_chicken='Y'”. It’s obvious that we successfully caught those dishes which don’t contain “chicken” but are actually chicken dishes.

2.2.2.4 Query for our use case

Combining the above steps, we could get the answer for our use case only by using MySQL and uncleaned data.

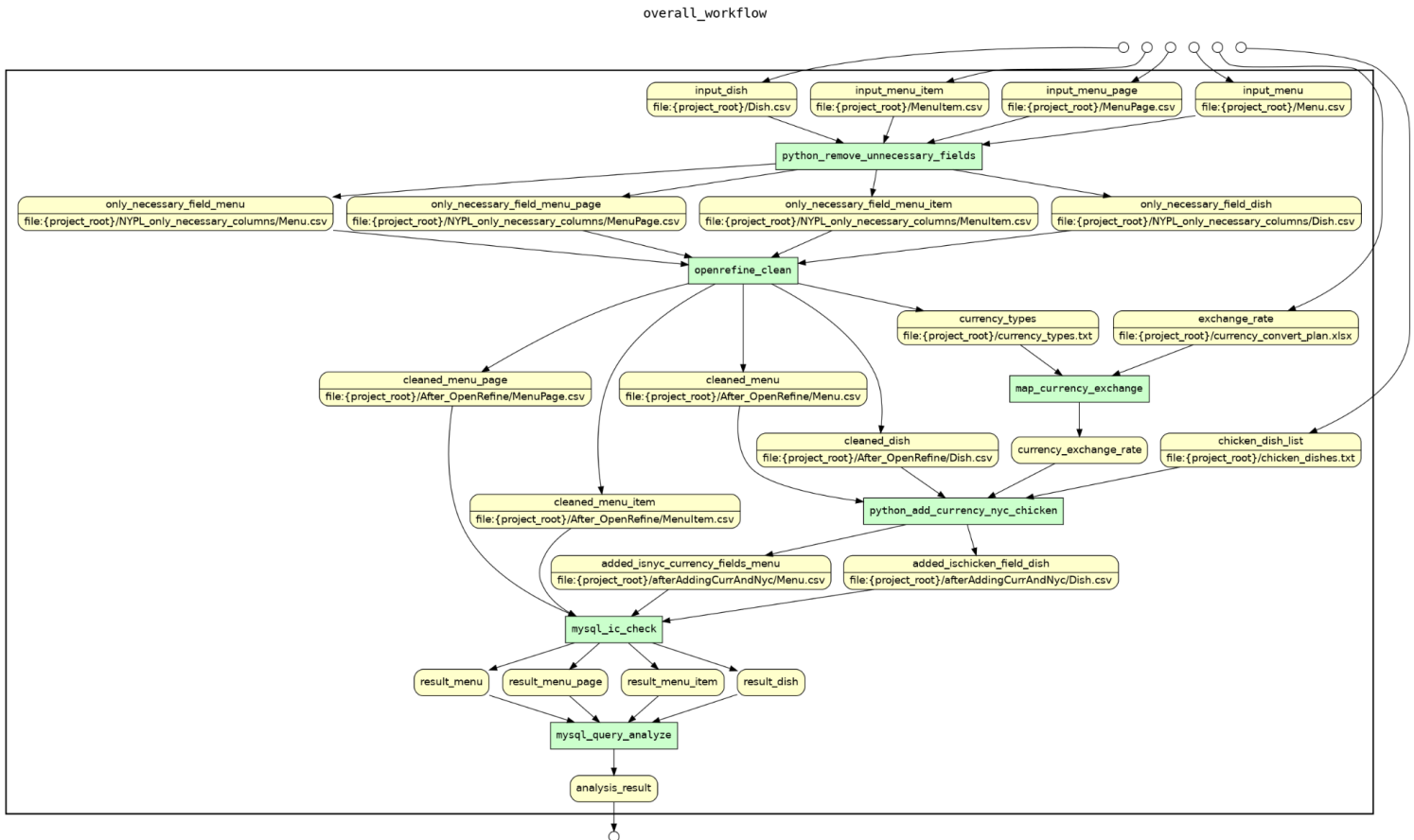
	cleaned	uncleaned
1887-1896	Is chicken: 328 (2.73%), \$0.826 Is not chicken: 11,993, \$0.713	Is chicken: 229 (1.99%), \$0.8156 Is not chicken: 11,472, \$0.6755
1897-1906	is chicken: 4310 (3.63%), \$1.084 is not chicken: 118,497, \$0.597	Is chicken: 3688 (3.06%), \$1.0549 Is not chicken: 120,179, \$0.5981

It is shown that the result truly changes after we cleaned the data. The difference seems insignificant; however, in this process, we actually use a lot of cleaning processes by using MySQL itself. If we don’t use them, such as regular expressions, ESCAPED BY ‘\b’, and Input Preprocessing for N/A values ([see](#)

[section 1.4.1](#)), we have no chance to load the data and query the data. We think these count data cleaning steps, so it proves that data cleaning processes are indispensable for our use case.

3. Create a Workflow Model

3.1 Overall Workflow W_0



First, we profiled data and removed unnecessary fields. This should be the first step because it helped us get an overview of the dataset and focus on only the fields related to our use case. We used Python to do this task because we can easily use the “pandas” package in Python to drop fields and count N/A. Moreover, since MenuItem.csv is too large to be read by OpenRefine, Python is a better choice to do this task.

Now, we could start to get our hands dirty. In this step, we aimed to combine different values which actually represent the same thing into one value. For example, “New York” is the same as “New York” (might be just a typo), “Chicken” is the same as “chicken”, “Taiwanese food” is the same as “Taiwanese food” (multiple consecutive whitespace), etc. If we didn’t do this task, we would miss a lot of data when doing queries in MySQL. We use OpenRefine because it offers many easy-to-use built-in functions and GUIs that help us in clustering, changing all letter cases into lowercase, trimming whitespace, etc.

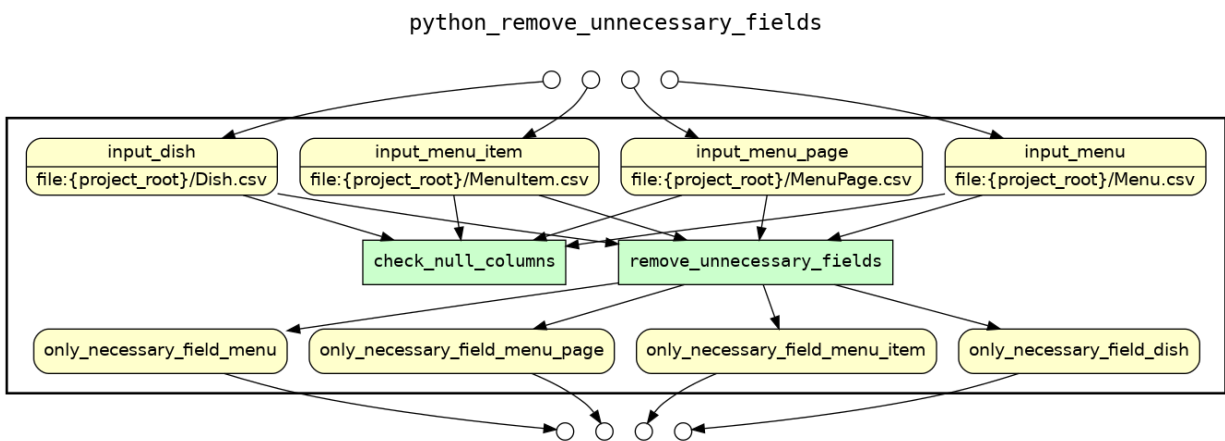
After the data was normalized, we could start to add new fields. We aimed to add Menu.is_in_nyc, Menu.dollar_rate, and Dish.is_chicken. We used Python mainly because we needed to list all the words that represent chicken dishes. We could customize our functions easily to split the chicken dishes' names into words and see whether the dishes' names contain these words.

Finally, we had to merge the four tables for our use case. MySQL, which is a relational database, is the most suitable tool to do this task. We used MySQL to check the foreign key constraint and merge four tables.

3.2 Inner Data Cleaning Workflow W_i

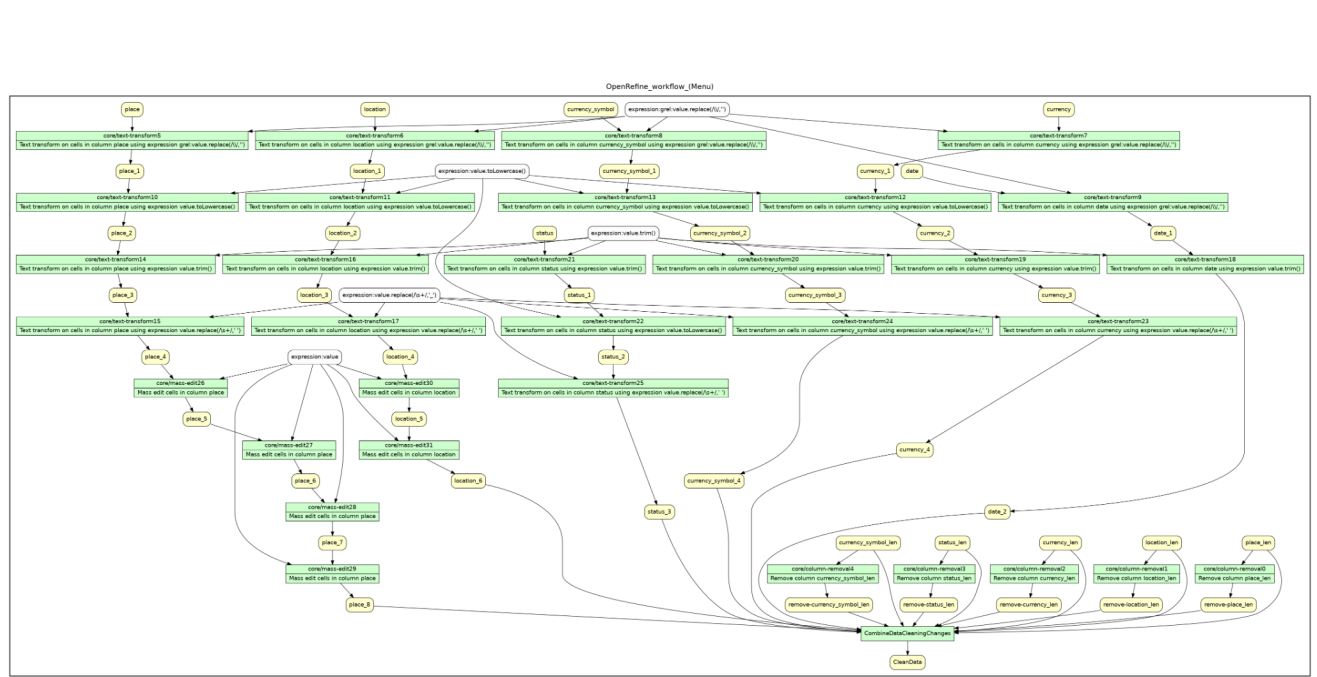
3.2.1 Python_profiling_remove_unnecessary_fields W_1

There are four csv files and we removed unnecessary fields in order to make our tasks more efficient.



3.2.2 OpenRefine_Menu W_2

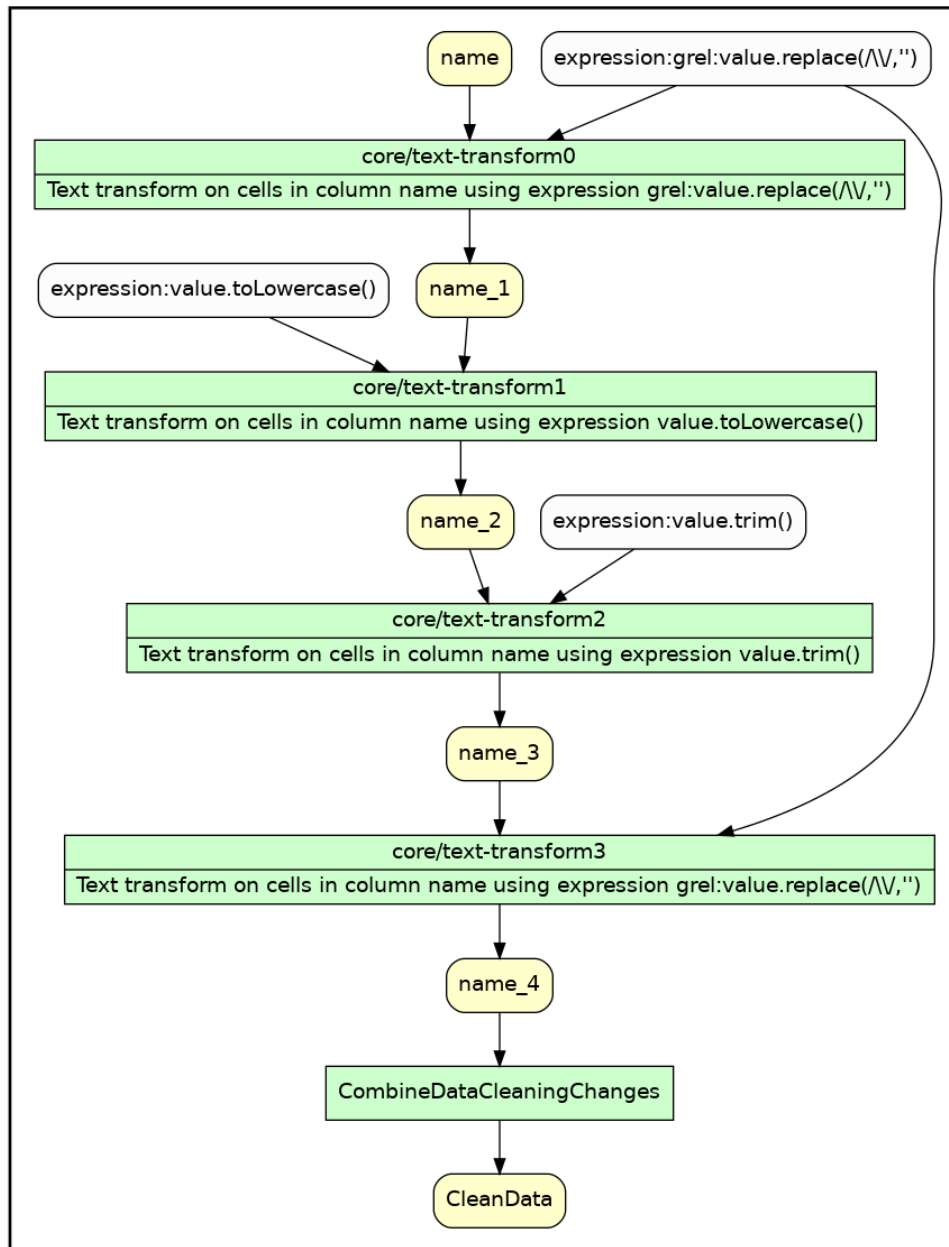
Here is an overview of OpenRefine's menu field cleaning task.



3.2.3 OpenRefine_Dish W₃

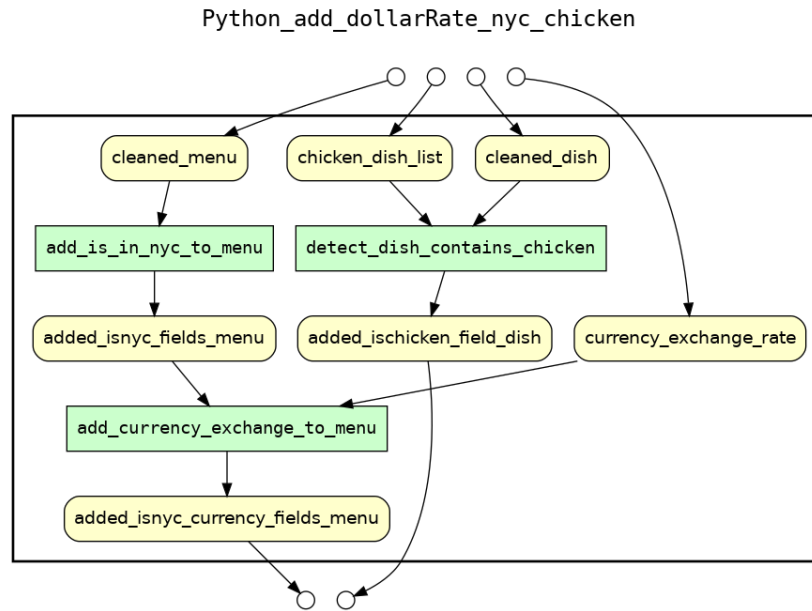
Here is an overview of OpenRefine's dish field cleaning task.

OpenRefine_workflow_(Dish)



3.2.4 Python_add_dollarRate_nyc_chicken

Here is an overview of a Python script for adding dollar_rate, is_in_nyc, is_chicken.



4. Conclusions & Summary

We tried to choose the right tools to complete the cleaning tasks. OpenRefine is good at clustering, Python enables us to customize our own functions, and MySQL helps us merge tables, check integrity constraints and answer our questions.

Although overall features are good enough for our use case, we found that there are several unscalable steps. For example, when we tried to load MenuItem.csv on the OpenRefine, the Internet browser crashed, consuming a huge amount of memory. When clustering the data, we still need to check them one by one. In this dataset, we didn't spend too much time on it. However, if the dataset is hundreds or thousands times larger than this dataset, it would be difficult to manually check them. Moreover, we filled the currency exchange rates manually, which makes our tasks not scalable.

We didn't find the exact currency exchange rate around 1897, so we had to make some compromise. Even though there are good chances of dramatic changes, most of the currencies are dollars so it would be acceptable for our use case. Also, we do not have expertise in the food industry. We are using a basic keyword matching mechanism to determine whether the dish is a chicken menu or not. We do not know how much portion of the dish has the chicken ingredients, which makes it hard to determine the chicken price changes.

Despite these limitations, we could learn how to clean data with proper tools, such as OpenRefine, RegEx, Python, Microsoft Excel, MySQL, YesWorkflow, Graphvis, or2yw. Data cleaning includes all of the steps improving data quality quantitatively and qualitatively. In this end to end final project, we have learned that Data Cleaning is an important and fundamental step to keep data fit in the user's use case.

5. Supplementary materials

Here are our supplementary materials' link. All of the files below can be accessed through the publicly opened U of I Box folder: <https://uofi.box.com/s/g89zwamb9bl1aasfuw5n19pjw5qz19jj>

1. Original Dataset
 - a. CS513-Summer2021-Dataset > Menu_2021_06_16_07_01_26_data:
<https://uofi.app.box.com/s/y7123kuldpzd4m3126sqvpa2agvv08hv/folder/139744503005>
2. Intermediate Dataset
 - a. After_OpenRefine.zip: <https://uofi.box.com/s/cgzk0logpodphiht7j5n6ynl7sgkuldc>
 - b. NYPL_only_necessary_columns.zip:
<https://uofi.box.com/s/pmbfvmlsm0gng3eqsrplp2e2u2xadh4v>
3. Final Cleaned Dataset
 - a. afterAddingCurrAndNyc.zip:
<https://uofi.box.com/s/0nxg5q61szj78we2ckwofqxsrbkkqwub>
4. Conceptual model / database schema
 - a. ERD.png: <https://uofi.box.com/s/u56qq6dkxo4o16hvzokfi33bjp4ddk93>
 - b. MySQL_DDL.txt: <https://uofi.box.com/s/kaswb3esmi9mwdepsqczgw134q7m3cr1>
5. Operation History
 - a. OpenRefineHistory_menu.json:
<https://uofi.box.com/s/24k7gs9ha4vdy0pevi8wfgsysept5xvi>
 - b. OpenRefineHistory_dish.json:
<https://uofi.box.com/s/qodpa37v3ohjdrntlnfzmp14i0jwfoa1>
 - c. Python Script (CS513_Final_Project.ipynb):
<https://uofi.box.com/s/uuc85mnzx0osl2s0oawtecmmhi3q2t3wc>
6. Queries
 - a. Queries.txt: <https://uofi.box.com/s/xdcyshqxd9kqy49ladkumfqwl0cp7z0x>
7. Outer and Inner Workflow Models
 - a. YW annotations files (these .py files has YesWorkflow annotations so we do not change the file extension as if .yw):
 - (1) Outer_workflow.py <https://uofi.box.com/s/qv9y93c5beup7fvvt9u6ukhwtw57s4h>
 - (2) python_remove_unnecessary_fields.py
<https://uofi.box.com/s/8sy4c5lsqvqvw4pouqo18pxaquhm4ryx1>
 - (3) python_add_dollarRate_nyc_chicken.py
<https://uofi.box.com/s/s23rhmtmp3v26m4phqh7m6ev3bo1p3p>
 - b. YW-generated Graphviz/DOT files (OverallWorkflow.gv):
 - (1) Outer_workflow.gv <https://uofi.box.com/s/x9s2kj6j7s9icwsnaoh93134fu0tppyw>
 - (2) python_remove_unnecessary_fields.gv
<https://uofi.box.com/s/k6ulps6azqqaligy2jnlklpk24fd9pbx>
 - (3) python_add_dollarRate_nyc_chicken.gv
<https://uofi.box.com/s/9fi59wvgvo3sekhqoyxjk79y7u91o9xe>

6. Data Analysis

6.1 Our use case U_1

How much has the popularity of the chicken menu changed in New York City after the **industrialization of chicken in 1897**? Did the price of the chicken menu rise or decrease?

To answer the first question, we need to get the average number of chicken dishes of each menu before and after the chicken industrialization (started around 1896), i.e. to get

$$\frac{\# \text{ of menu items containing "Chicken" in 1886-1896}}{\text{Total \# of menus in 1886-1896}} \text{ and } \frac{\# \text{ of menu items containing "Chicken" in 1897-1907}}{\text{Total \# of menus in 1897-1907}}$$

To answer the second question, we need to get the average price of chicken dishes before and after the chicken industrialization, i.e. to get $\frac{\text{Sum of price of menu items containing "Chicken" in 1886-1896}}{\# \text{ of menu items containing "Chicken" in 1886-1896}}$ and

$$\frac{\text{Sum of price of menu items containing "Chicken" in 1897-1907}}{\# \text{ of menu items containing "Chicken" in 1897-1907}}.$$

6.2 MySQL queries and returned values

All the queries were uploaded in **MySQL_DDL.txt**, **Queries.txt**.

Q2 : check chicken menu percentage & average price in 1887-1896

```
SELECT is_chicken, count(1) count, AVG(dollar_rate*price) as avg_price
FROM nyc_chicken
WHERE menu_date >= '1887-01-01' and menu_date < '1897-01-01'
GROUP BY is_chicken;
```

Returned values:

```
-- is_chicken Y: 328 (2.73%), $0.826
-- is_chicken N: 11,993, $0.713
```

Q3 : check chicken menu percentage & average price in 1897-1906

```
SELECT is_chicken, count(1) count, AVG(dollar_rate*price) as avg_price
FROM nyc_chicken
WHERE menu_date >= '1897-01-01' and menu_date < '1907-01-01'
GROUP BY is_chicken;
```

Returned values:

```
-- is_chicken Y: 4310 (3.63%), $1.084
-- is_chicken N: 118,497, $0.597
```

Before 1897, the percentage of chicken dishes was 2.73%, and the average price of chicken dishes was \$0.826. After 1897, the percentage of chicken dishes was 3.63%, and the average price of chicken dishes was \$1.084. Unfortunately, the percentage of chicken dishes didn't explode after the industrialization of chicken in 1897, it only rose slightly, which violates our expectation. In addition, the price of chicken dishes actually rose a lot after the industrialization of chicken.

Since most menus centered around 1900, we were struggling with finding a meaningful year that leads to a significant change. We found an interesting year from Wikipedia [1].

Soon after poultry keeping gained the attention of agricultural researchers (around **1896**), improvements in nutrition and management made poultry keeping more profitable and businesslike. Poultry shows spread interest and understanding, with 88% of all farmers having chickens by 1910.

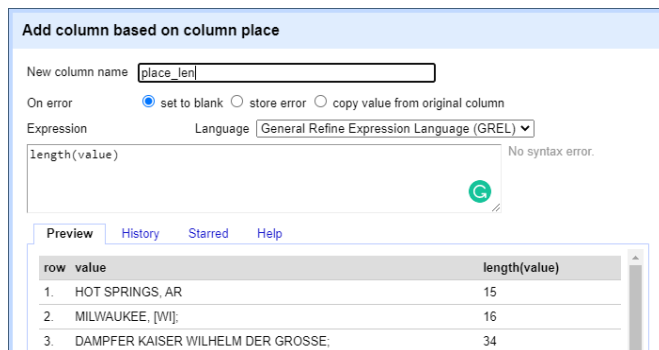
We expected that there was a substantial change of the number of chicken menus and price. As we can see, there was no big difference with regard to the number of chicken menus. The average price, however, increased a lot. From this result, we can conclude our analysis results. First, 1896 was not an eventual year of the chicken industry at least in terms of the number of chicken products. Second, however, after 1896, there was a significant increase in chicken prices. This may result from chicken industrialization or it could come from other restaurant business developments.

Due to the limited information, we cannot conduct more deep analysis. Our purpose of this project is to improve data quality and answer the U1 use case, and we would have more meaningful results if we have more information as in the U2 use case.

Appendix A. Steps of OpenRefine

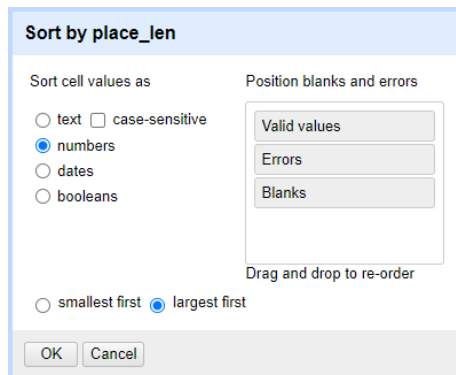
A.1 Check field types and length of longest string of each text fields

To get the max length of text fields, first, add the length column like below:
Edit Column > Add column based on this column...



row	value	length(value)
1.	HOT SPRINGS, AR	15
2.	MILWAUKEE, [WI];	16
3.	DAMPFER KAISER WILHELM DER GROSSE;	34

Then, sort the length field in descending order.



Sort cell values as

☐ text ☐ case-sensitive

☒ numbers

☐ dates

☐ booleans

Position blanks and errors

Valid values

Errors

Blanks

Drag and drop to re-order

☐ smallest first ☒ largest first

OK Cancel

Now, check the max length of the field. For example, as we sorted place_len in descending order, 106 is the max length of the place field.

place	place_len
NOS. 667 TO 677 [BROADWAY], OPPOSITE BOND STREET, MIDWAY BETWEEN BATTERY AND CENTRAL PARK, NEW YORK, [NY];	106
NOS. 667 TO 677 [BROADWAY], OPPOSITE BOND STREET, MIDWAY BETWEEN BATTERY AND CENTRAL PARK, NEW YORK [NY];	105
Restaurant (Le Pavillon du Lac) du Parc Montsouris Le Jardin de la Paresse, 20 rue Gazan, France	96

By using this method, we check the max length of every text field, and the result was shown in [Figure 4](#).

A.2 Cluster & merge on place, location columns

This feature is going to help find NewYorkCity as there are a lot of variants that might represent the same thing especially for NewYorkCity.

Method	key collision	Keying Function	fingerprint	
Cluster Size	Row Count	Values in Cluster	Merge?	New Cell Value
10	51	<ul style="list-style-type: none">delmonico's ny (26 rows)delmonico's [ny] (15 rows)delmonico's [ny?] (2 rows)delmonico's, ny (2 rows)delmonico's, [ny] (1 rows)delmonico's, n.y. (1 rows)delmonico's, ny; (1 rows)delmonicos ny (1 rows)delmonicos', ny (1 rows)delmonicos, (ny) (1 rows)	<input checked="" type="checkbox"/>	delmonico's ny
9	254	<ul style="list-style-type: none">new york, ny (198 rows)new york, [ny] (18 rows)[new york, ny] (14 rows)[new york, ny?] (7 rows)(new york, ny?) (6 rows)new york, [ny]; (5 rows)new york [ny] (3 rows)new york, ny [?] (2 rows)[new york, ny]; (1 rows)	<input checked="" type="checkbox"/>	new york, ny

Furthermore, we used several keying functions in order to find the core meaning as much as possible. For example, after merging with the fingerprint function, there are still many collisions.

Method	key collision	Keying Function	metaphone3	
8	82	<ul style="list-style-type: none">delmonico's, new york, ny (49 rows)delmonico's [new york] (16 rows)delmonico's, [new york] (5 rows)delmonico's, new york city (4 rows)delmonico's, [new york, ny?] (3 rows)[delmonico's, new york ny] (2 rows)delmonico's new york, ny (2 rows)delmonic0's, [new york] (1 rows)	<input checked="" type="checkbox"/>	delmonico's, new york, ny

However, we are going to merge only meaningful rows. For example, “jersey city, nj” and “jersey city, ny” are unchecked so it would not be merged. We are not good at American geography, so we googled “nj” and “ny”. The first result is “New Jersey” and “New York”, respectively. From these results, we guessed that “nj” is not equal to “ny”.

Method nearest neighbor levenshtein Radius 1.0 Block Chars 6

Cluster Size	Row Count	Values in Cluster	Merge?	New Cell Value
2	2	<ul style="list-style-type: none">jersey city, nj (1 rows)jersey city, ny (1 rows)	<input type="checkbox"/>	jersey city, nj

Method nearest neighbor levenshtein Radius 1.0 Block Chars 6

2	276	<ul style="list-style-type: none">new york, ny (272 rows)new yok ny (4 rows)	<input checked="" type="checkbox"/>	new york, ny
---	-----	---	-------------------------------------	--------------

2	2	<ul style="list-style-type: none"> vendome hotel,ny (1 rows) vendome hotel,nyc (1 rows) 	<input checked="" type="checkbox"/>	vendome hotel,ny
---	---	---	-------------------------------------	------------------

If they contain “new york” explicitly, it would be located in NYC. However, if they do not contain “ny” or “nyc”, we will not merge values since “murray hill hotel” can be located in another city.

Method Radius Block Chars

Cluster Size	Row Count	Values in Cluster	Merge?	New Cell Value
3	5	<ul style="list-style-type: none"> murray hill hotel,ny (3 rows) murray hill hotel (1 rows) murray hill hotel [nyc] (1 rows) 	<input type="checkbox"/>	murray hill hotel,ny
3	3	<ul style="list-style-type: none"> hoffman cafe, new york (1 rows) hoffman cafes, new york, ny (1 rows) hoffman house cafe, new york, ny (1 rows) 	<input checked="" type="checkbox"/>	hoffman cafe, new york

Since different clustering methods suggest different items, we applied fingerprint, metaphone3, levenshtein, Beider-Morse in order.<https://docs.openrefine.org/manual/cellediting>

A.3.1 menu_item::menu_page_id Foreign Key Check

“menu_page_id” in the MenuItem table should correspond to the “id” in the MenuPage table. We will check this by using ADD CONSTRAINT syntax.

```
ALTER TABLE menu_item ADD CONSTRAINT FK_menu_item FOREIGN KEY (menu_page_id)
REFERENCES menu_page(id);
-- 1334538 row(s) affected Records: 1334538 Duplicates: 0 Warnings: 0          93.984 sec
```

This query successfully completed, which means there are no foreign key violations.

A.3.2 menu_item::dish_id Foreign Key Check

“dish_id” in the MenuItem table should correspond to the “id” in the Dish table. We will check this by using ADD CONSTRAINT syntax.

```
ALTER TABLE menu_item ADD CONSTRAINT FK_menu_item2 FOREIGN KEY (dish_id)
REFERENCES dish(id);
-- Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails
('cs513`.`#sql-1b34_28`, CONSTRAINT `FK_menu_item2` FOREIGN KEY (`dish_id`)
REFERENCES `dish` (`id`))
```

Let us check what violates the constraints.

```
select
  mi.id menu_item_id,
  mi.dish_id dish_id1,
  d.id dish_id2
from
  menu_item mi
left join dish d on (mi.dish_id = d.id)
```

```
where d.id is null;  
-- 3 rows returned
```

There are no dish_ids (220797, 329183, 395403) in the original Dish.csv file. This means MenuItem includes invalid records that violate foreign key rules or there might be some missing records in the Dish file.

menu_item_id	dish_id1 (from MenuItem)	dish_id2 (from Dish)
619133	220797	NULL
837354	329183	NULL
1047160	395403	NULL

Also, we can check the other side.

```
select  
  mi.id menu_item_id,  
  mi.dish_id dish_id1,  
  d.id dish_id2  
from  
  menu_item mi  
right join dish d on (mi.dish_id = d.id)  
where mi.dish_id is null;  
-- 12752 rows returned
```

menu_item_id	dish_id1 (from MenuItem)	dish_id2 (from Dish)
NULL	NULL	3031
NULL	NULL	3032
NULL	NULL	3033
...

There are no dish_ids (3031, 3032, 3033) in the original MenuItem.csv file. This means Dish includes invalid records that violate foreign key rules or there might be some missing records in the MenuItem file.

Appendix B. References

[1] Poultry farming in the United States:

https://en.wikipedia.org/wiki/Poultry_farming_in_the_United_States

[2] MySQL load NULL values from CSV data:

<https://stackoverflow.com/questions/2675323/mysql-load-null-values-from-csv-data>