CS 513: Theory and Practice of Data Cleaning - Final Project Phase 1

Shyam Shah - shyam3@illinois.edu

28/07/2021

1. Points received for initial Phase-1 submission: 98/100
2. Our team chooses the following Phase-1 option: (A) No change to Phase-1 report

Phase 1 Report

The purpose of this project is to take an existing dataset, apply data cleaning techniques learned through this course to it, and use it to satisfy a specified use case.

Identifying the Dataset

I have decided to use the US farmer's market dataset (D) provided to us by the course instructors.

Describing the Dataset

Let's first look at what information we have available in our dataset. This dataset consists of a single CSV file which includes information for all registered farmer's markets in the United States in 2020. It is originally provided by the U.S. Department of Agriculture and the county information was added to it before being posted as a public Kaggle dataset. Listed below is a basic schema (column name, expected column type) for this table:

```
CREATE TABLE FarmersMarket {
  FMID int,
  MarketName varchar,
  Website varchar,
  Facebook varchar,
  Twitter varchar,
  Youtube varchar,
  OtherMedia varchar,
  street varchar,
  city varchar,
  County varchar,
  State varchar,
  zip int,
  Season1Date varchar,
  Season1Time varchar,
  Season2Date varchar,
```

```
  Season2Time varchar,
  Season3Date varchar,
  Season3Time varchar,
  Season4Date varchar,
  Season4Time varchar,
  x float,
  y float,
  Location varchar,
  Credit varchar,
  WIC varchar,
  WICash varchar,
  SFMNP varchar,
  SNAP varchar,
  Organic varchar,
  Bakedgoods varchar,
  Cheese varchar,
  Crafts varchar,
  Flowers varchar,
  Eggs varchar,
  Seafood varchar,
  Herbs varchar,
  Vegetables varchar,
  Honey varchar,
  Jams varchar,
  Maple varchar,
  Meat varchar,
  Nursery varchar,
  Nuts varchar,
  Plants varchar,
  Poultry varchar,
  Prepared varchar,
  Soap varchar,
  Trees varchar,
  Wine varchar,
  Coffee varchar,
  Beans varchar,
  Fruits varchar,
  Grains varchar,
  Juices varchar,
  Mushrooms varchar,
  PetFood varchar,
  Tofu varchar,
  WildHarvest varchar,
  updateTime date
}
```

There are a lot of columns in this table but they can be broken down into the following groups

- The 'FMID' column is a unique identifier for the each farmer's market and the 'MarketName' column is the actual name of the market.
- The columns from 'Website' to 'OtherMedia' contain information regarding social media links/tags for each market.
- The columns from 'street' to 'zip' contain the various address fields for the market.

- The columns from 'Season1Date' to 'Season4Time' contain information on the market's schedule (ie. when they are open).
- The 'x' and 'y' columns are te latitude and longitude values for each market.
- The 'Location' field contains a description of the market's location
- The columns from 'Credit' to 'SNAP' contain the payment options supported by the market.
- The columns from 'Organic' to 'WildHarvest' indicate whether the market has those types of products
- The 'updateTime' column is the last time that the information for the market was updated.

Developing a Target Use Case

Let's first look at a use case (U_0) that our dataset (D) would already support without any need for data cleaning. The 'MarketName' (and 'FMID') columns do not contain any nulls or empty strings. By applying a text facet on the 'Credit' field, we can also see that values are either 'Y' or 'N'. So we can already use this data to answer a question: which markets accept credit card? I'll use a SQL query to represent this use case U_0:

```
SELECT DISTINCT(MarketName)
FROM FarmersMarket
WHERE Credit='Y'
```

Note that there may be duplicate market names, so we used the SQL DISTINCT function. Other similar queries can be used/questions can be answered using these two fields such as: how many markets accept credit card.

Next let's look at a use case that will require some data cleaning (U_1). This use case will be what the one that this project aims to answer. I'll be splitting this use case into a few different parts:

1. Create a standard for all of the product type columns to enable querying any of them with the same conditions
2. Store all of the times that they are open in a single column (or set of columns) to make finding available times to visit easier
3. Having a single column (or set of columns) to make locating markets easier

All of these parts will require some data cleaning, as there are inconsistencies in how the current columns provided are formatted. An example of a question that we would be able to answer after addressing these issues could be: which markets in New York have jam? This is a query we can use to define this use case:

```
SELECT MarketName
FROM FarmersMarket
WHERE State='New York'
  AND Jam='Y'
```

Another question could be: which markets in Washington are open on June 10th? Here is a query that attempt to answer this:

```
SELECT MarketName
FROM FarmersMarket
WHERE city='Washington'
  AND STR_TO_DATE(LEFT(SeasonDate1,10))<=STR_TO_DATE('06/10/2020')<=STR_TO_DATE(RIGHT(SeasonDate1,10))
```

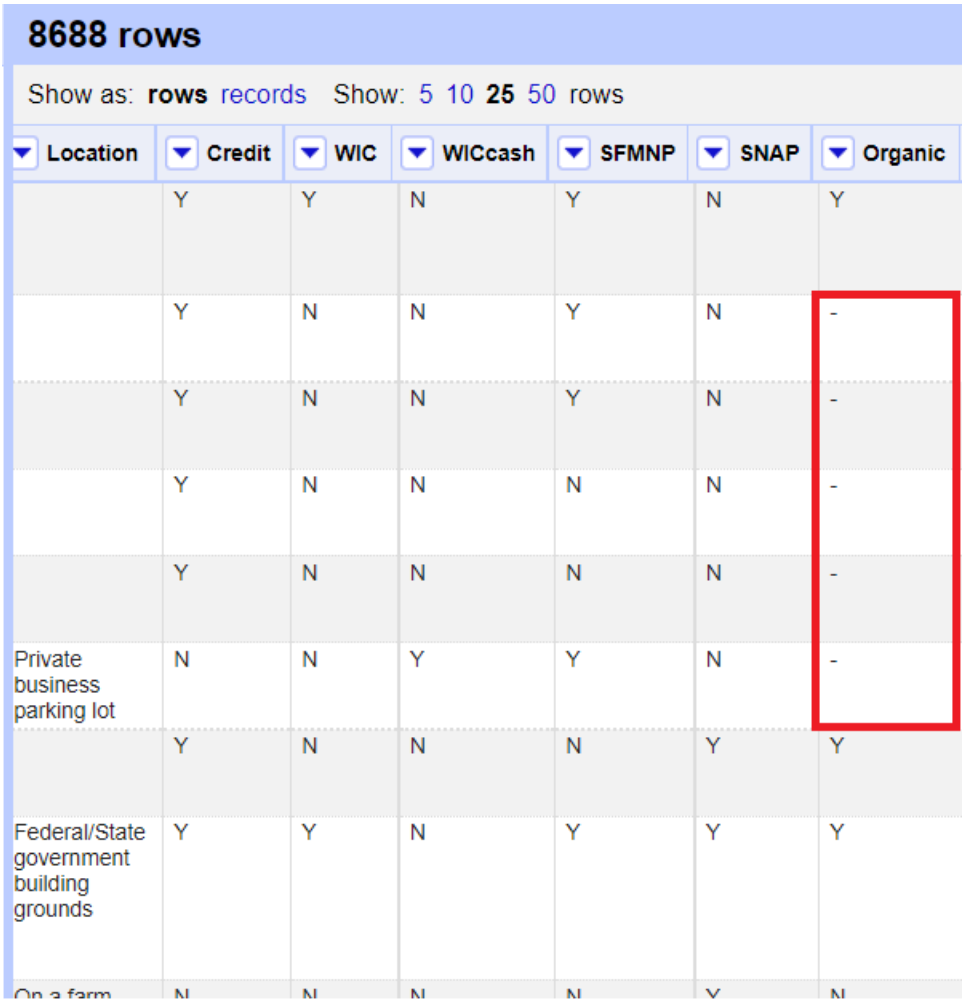
Lastly, let's look at a use case that we will not be able to achieve with any amounts of data cleaning (U_2). There are some columns that we can infer the values for based on other columns, but there are some that we can not such as the 'Website' column. So a question that we will not be able to answer would be: what are the websites for all markets in New York? Here is a query for this, which will not return the correct results:

```
SELECT Website
FROM FarmersMarket
WHERE State='New York'
```

Finding Obvious Data Quality Problems

For our use case (U_1), there are some quality issues for each of the 3 parts mentioned above.

1. Some of the columns that indicate if a product is served at a market are either blank, or use the '-' character.



8688 rows

Show as: **rows** records Show: 5 10 **25** 50 rows

Location	Credit	WIC	WICcash	SFMNP	SNAP	Organic
	Y	Y	N	Y	N	Y
	Y	N	N	Y	N	-
	Y	N	N	Y	N	-
	Y	N	N	N	N	-
	Y	N	N	N	N	-
Private business parking lot	N	N	Y	Y	N	-
	Y	N	N	N	Y	Y
Federal/State government building grounds	Y	Y	N	Y	Y	Y
On a farm	N	N	N	N	Y	N

2. For the market's schedules, some of the dates are formatted differently, and some have their schedules split into multiple columns. Although having multiple columns for each season has it's advantages, it is something we will need to change for our use case.

8688 rows

Show as: **rows** records Show: 5 10 **25** 50 rows

street	city	County	State	zip	Season1Date	Season1Time	Season2Date	Season2Time
	Danville	Caledonia	Vermont	5828	06/14/2017 to 08/30/2017	Wed: 9:00 AM-1:00 PM;	09/06/2017 to 10/18/2017	Wed: 2:00 PM-6:00 PM;
6975 Ridge Road	Parma	Cuyahoga	Ohio		06/24/2017 to 09/30/2017	Sat: 9:00 AM-1:00 PM;		
6975 Ridge Road	Parma	Cuyahoga	Ohio		06/24/2017 to 09/30/2017	Sat: 9:00 AM-1:00 PM;		
106 S. Main Street	Six Mile		South Carolina	29682				
10th Street and Poplar	Lamar	Barton	Missouri	64759	04/02/2014 to 11/30/2014	Wed: 3:00 PM-6:00 PM;Sat: 8:00 AM-1:00 PM;		
112th Madison Avenue	New York	New York	New York	10029	July to November	Tue:8:00 am - 5:00 pm;Sat:8:00 am - 8:00 pm;		

3. For the location information, some of the columns have slightly different names that would need to be clustered together. A text facet was used in OpenRefine to show this.

Devising the Initial Plan

The first two steps of the plan would be to define the dataset and use case (S_1), and profiling the data for quality problems (S_2) which has been done above.

The next step (S_3) will be to perform the data cleaning. The current plan is to use OpenRefine to do things like removing whitespaces from all text columns, clustering the location values to ensure they can be grouped appropriately, and renaming values to make the indicator columns uniform. I also intend on using Python for some extrapolation of data. For example, I will use the geopy package to pull state names from zip codes when possible, and I will explore the possibility of scraping data to populate the 'Facebook' or 'Website' columns although this part does not help address U_1.

After this, S_4 will involve writing a query on the initial dataset D to find an answer to U_1, and then writing an identical query over the cleaned dataset D' to do the same and compare the results. There are various integrity constraints I can check before making any changes to the dataset. For example, I can check that all social media links are actually links for the given social media. An integrity constraint I can check after cleaning the data could be to check that all address, city, county, state, zip code tuples are accurate (for example: a given address is in the specified county).

The last step (S_5) will be documenting the changes made to get from D to D'. This can be done by extracting the OpenRefine history, documenting any Python code used, and then creating a YesWorkflow model to tie it all together.

Since I am working on this by myself, I will be responsible for all of the steps.

Phase 2 Report

Data Cleaning Performed

OpenRefine

The FMID column is meant to be unique across all markets, so I also removed any duplicate rows based on this column. This was important for our use case because this way we can be sure that any aggregation is not skewed because of duplicate data.

The indicator columns (from 'Organic' to 'WildHarvested') were cleaned by replacing any blank or '-' characters with a 'N' to indicate that the farmer's market does not sell that product. This is directly related to our use case since now eveyr market has an indication that is either a 'Y' for yes or a 'N' for no. One thing to note is that there was a bit of inference required here, since it's likely that if a column is empty, it is because the data is not missing and not because the food product is not being sold. This is further suggested by looking at the number of rows edited in the table below, which shows the same number for many columns.

The 'city' and 'County' fields were clustered to group rows with slightly different spellings together. There were cases where one field contained the information for multiple fields, or it contained characters that weren't being handled properly, so I also decided to remove those values for now and repopulate them in the Python step. I also checked the 'State' field but it did not require clustering or filling out.

The 'Season1Date', 'Season2Date', 'Season3Date', and 'Season4Date' columns caused the first roadblock. I wanted to put these values into a nested field, so that SQL queries can be written to take advantage of a single column holding all of the information but quickly realized that not only is that not directly supported in OpenRefine, it's not supported in sqlite which is what I will be using to devise ICV queries. Instead I decided to just split each of them into two columns for the start and end dates and convert the columns that were already in a format similar to 'dd/mm/yyyy' to actual dates.This way we can still use the information to satisfy our use case.

Python

The 'city', 'County', and 'zip' fields were that were missing were populated with Python. I found that the 'x' and 'y' fields (for latitude and longitude) were most consistently populated, so I used them and the geopy Python package to fill in any missing information. This builds on the OpenRefine cleaning steps and allows queries to pick up more data.

Data Quality Changes

Here's a quantification of the data changes:

Tool	Step	Rows Modified
OpenRefine	Remove duplicates with FMID	1
OpenRefine	Edit organic column	5043
OpenRefine	Edit baked goods column	2933
OpenRefine	Edit cheese column	2933
OpenRefine	Edit crafts column	2933
OpenRefine	Edit flowers column	2933
OpenRefine	Edit eggs column	2933
OpenRefine	Edit seafood column	2933
OpenRefine	Edit herbs column	2933
OpenRefine	Edit vegetables column	2933
OpenRefine	Edit honey column	2933
OpenRefine	Edit jams column	2933
OpenRefine	Edit maple column	2933
OpenRefine	Edit meat column	2933
OpenRefine	Edit nursery column	2933
OpenRefine	Edit nuts column	2933
OpenRefine	Edit plants column	2933
OpenRefine	Edit poultry column	2933
OpenRefine	Edit prepared column	2933
OpenRefine	Edit soap column	2933
OpenRefine	Edit trees column	2933
OpenRefine	Edit wine column	2933
OpenRefine	Edit coffee column	2933
OpenRefine	Edit beans column	2933
OpenRefine	Edit fruits column	2933
OpenRefine	Edit grains column	2933
OpenRefine	Edit juices column	2933
OpenRefine	Edit mushrooms column	2933
OpenRefine	Edit pet food column	2933
OpenRefine	Edit tofu column	2933
OpenRefine	Edit wild harvested column	2933
OpenRefine	Edit crafts column	2933
OpenRefine	Cluster city column	565
OpenRefine	Cluster county column	734
OpenRefine	Split season 1 date column	5479
OpenRefine	Split season 2 date column	449
OpenRefine	Split season 3 date column	81
OpenRefine	Split season 4 date column	6
OpenRefine	Convert season 1 date columns to dates	4669
OpenRefine	Convert season 2 date columns to dates	424
OpenRefine	Convert season 3 date columns to dates	75
OpenRefine	Convert season 4 date columns to dates	6
Python	Add zip codes	889
Python	Add counties	476
Python	Add cities	59

Now lets look at some ICV queries and their results to demonstrate the improvements in data quality. The original dataset is in a table called ‘original’ and the final dataset is in a table called ‘updated’.

First let’s check if the FMID column is unique:

```
sqlite> select count(distinct(fmid)) - count(fmid) from original;
```

```
-1

sqlite> select count(distinct(fmid)) - count(fmid) from updated;
0
```

Next let's check the unique values in the indicator columns. I've randomly chosen a few to demonstrate this:

```
sqlite> select distinct(organic) from original;
Y
-
N

sqlite> select distinct(organic) from updated;
Y
N

sqlite> select distinct(maple) from original;
Y
""
N

sqlite> select distinct(maple) from updated;
Y
N
```

Lastly let's check if the Python code increased the quality of our location data:

```
sqlite> select count(zip) from original where zip!="";
7743

sqlite> select count(zip) from updated where zip!="";
8687

sqlite> select count(county) from original where county!="";
8173

sqlite> select count(county) from updated where county!="";
8687

sqlite> select count(city) from original where city!="";
8649

sqlite> select count(city) from updated where city!="";
8687
```