MIDS W205

|  |  |  |  |
| --- | --- | --- | --- |
| **Lab #** | 10 | **Lab Title** | OpenRefine—Introduction |
| **Related Module(s)** | 10 | **Goal** | Get you started on OpenRefine and edit distance |
| **Last Updated** | 9/27/15 | **Expected duration** | 60 minutes |

# Introduction

This lab has three parts. The first two involve using OpenRefine to clean up some data files. The third one involves calculating the Levenshtein distance between two strings.

OpenRefine is an open source tool for working with bad data. In this lab we will give you a quick tour of how you can use it to clean data. This is purposely a short introduction to just get you introduced to the tool. If you want a more comprehensive tutorial you can follow any of those listed in the resources section. The second part of this lab involves understanding Levenshtein distance calculation using the dynamic programming method.

For the OpenRefine portion we will be using two datasets. One dataset has earthquake data, and one contains customer complaint data. You can access the dataset from Github (both datasets are available if you clone or pull) or use the links in the text below. The links are also in the resources sections later in this document.

The first dataset contains customer complaints; you can download that dataset [here](https://github.com/UC-Berkeley-I-School/w205-labs-exercises/blob/master/lab_10/dataset/Consumer_Complaints.csv).

The second dataset is the eq2015 dataset, which contains data about earthquakes of magnitude 3 or more during the first six months of 2015. You can download the earthquake dataset [here](https://github.com/UC-Berkeley-I-School/w205-labs-exercises/blob/master/lab_10/dataset/eq2015.csv). You can find an earthquake data attribute glossary [here](http://earthquake.usgs.gov/earthquakes/feed/v1.0/glossary.php#net).

OpenRefine is, to a large extent, menu driven, but it also allows you to use a language for doing certain types of transformations.

# OpenRefine

The basic idea in OpenRefine is that you should think of exploring your data in terms of patterns, called facets. Facets help by characterizing data and give you an overview of value ranges, missing values, and so on. There are a number of facets for different data types as well as plots such as scatter plots. Once you understand the data you can transform them using pattern matching and transformations. To support this, OpenRefine has functions for doing data transformations. These transformations can be expressed in the GREL language, although there are a few other options as well. As an example, you can decide to create a new column based on an existing column but with a transformation applied to the data. GREL allows you to match regular expressions and also perform common operations like trimming blanks, splitting strings, and so forth. In addition it has control structures such as if statements. You can even have OpenRefine call out to URLS and insert the results in a column. OpenRefine also supports fuzzy matching (clustering) of attribute values. It will suggest values to merge and will let you choose which one you want to use. You can adjust the way clustering works using parameters such as radius and character-block matching.

# Instructions, Resources, and Prerequisites

For Step 1 and Step 2, install OpenRefine from [here](http://openrefine.org/). The lab describes a number of commands to try and also poses a few questions we want you to consider and experiment with. For the submissions you should answer the SUBMISSION questions embedded through this lab.

For Step 3, make sure you have a working Python installation.

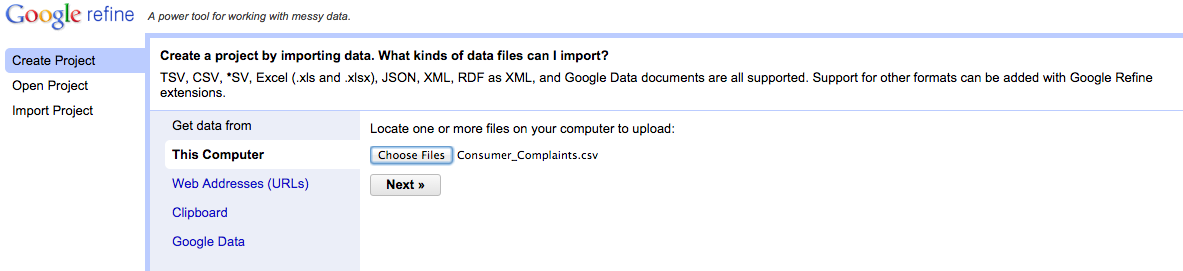
Below are a number of resources that may be of general interest to you during or after this lab.

|  |  |
| --- | --- |
| **Resource** | **What** |
| http://openrefine.org/ | This is where you download OpenRefine. |
| http://arcadiafalcone.net/GoogleRefineCheatSheets.pdf | A short description of OpenRefine commands. |
| http://enipedia.tudelft.nl/wiki/OpenRefine\_Tutorial | Another tutorial on OpenRefine. |
| http://davidhuynh.net/spaces/nicar2011/tutorial.pdf | Another tutorial on OpenRefine. |
| http://schoolofdata.org/handbook/recipes/cleaningdatawithrefine/ | Programming guide for the Spark Context object. Here you can find actions available on the Spark Contexts. |
| https://github.com/OpenRefine/OpenRefine/wiki/General-Refine-Expression-Language | GREL is the language used in OpenRefine for data refinements. This is a reference guide for the GREL language. |
| http://earthquake.usgs.gov/eart | Explanation of the earthquake data. |
| https://pypi.python.org/pypi/python-Levenshtein/0.12.0 | A Levenshtein module you can use to check your results in a Python shell. |
| https://github.com/OpenRefine/OpenRefine/wiki/Clustering-In-Depth | A good, quick read on some clustering methods. |
| Earth Quake Data set | [https://github.com/UC­Berkeley­I­School/w205­labs­exercises/blob/master/lab\_10 /dataset/eq2015.csv](https://github.com/UCBerkeleyISchool/w205labsexercises/blob/master/lab_10%20/dataset/eq2015.csv) |
| Earthquake Data Glossary. | <http://earthquake.usgs.gov/earthquakes/feed/v1.0/glossary.php#net> |
| Customer Complaints Data. | https://github.com/UC­Berkeley­I­School/w205­labs­exercises/blob/master/l ab\_10/dataset/Consumer\_Complaints.csv |

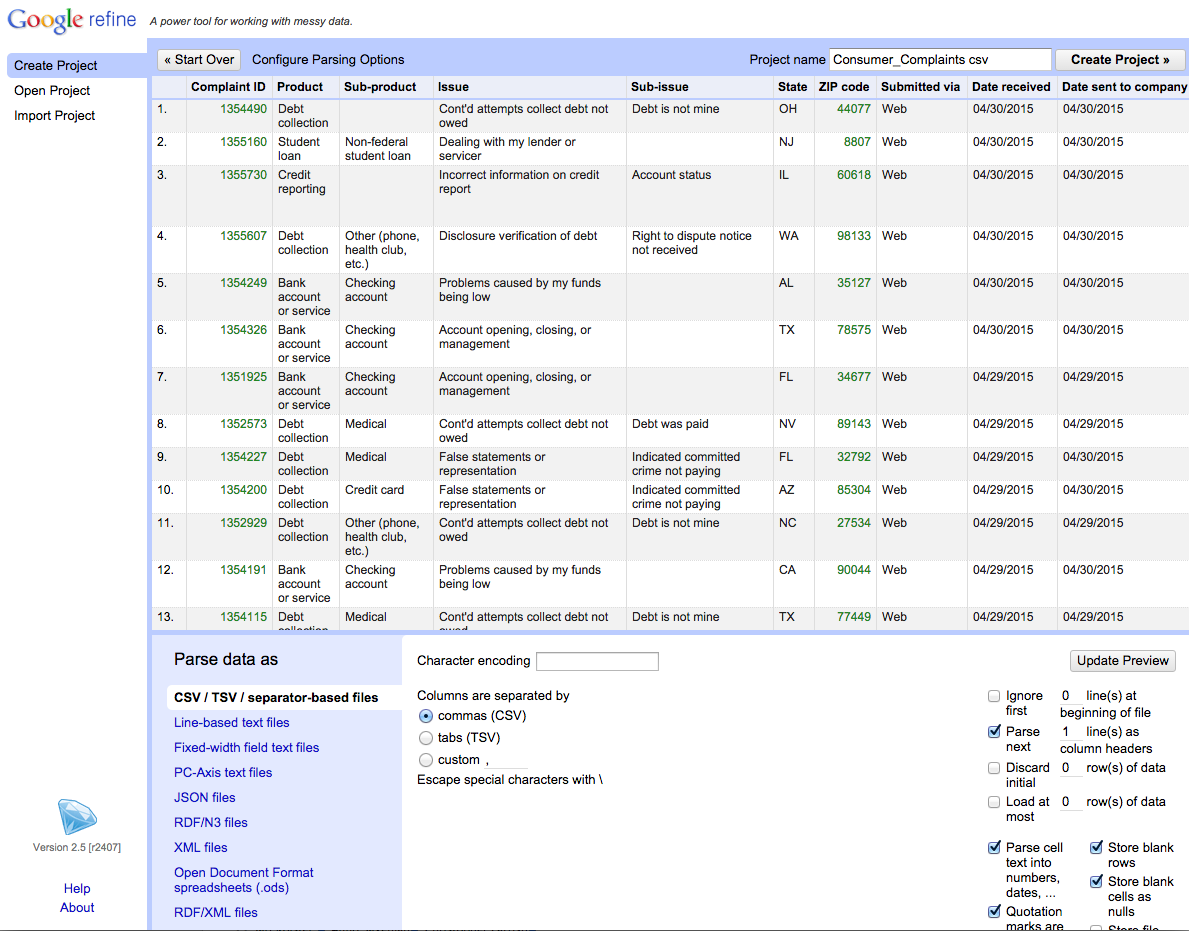
# Step 1. Wrangling the Customer Complaints Data

## Uploading data

After you start OpenRefine you can pick a dataset. For this first step choose the Customer Complaints dataset (dataset/Consumer\_Complaints.csv).



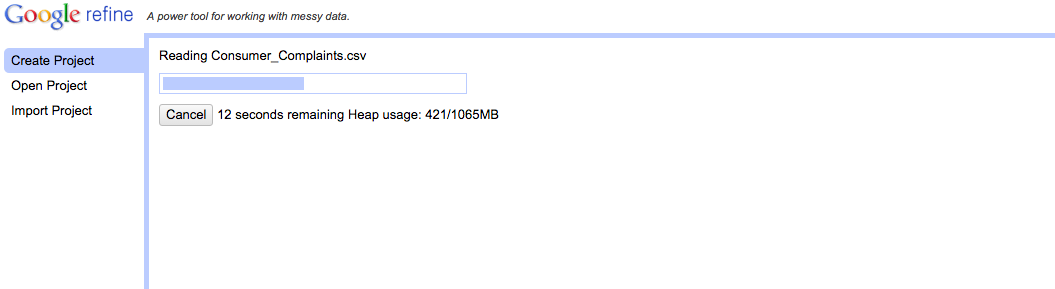
After the data are read, you can inspect them. In this case they look OK. But it they had been tab separated rather than comma separated, OpenRefine would not have identified the structure correctly. Because we think the data look good, we will now click **Create Project**.



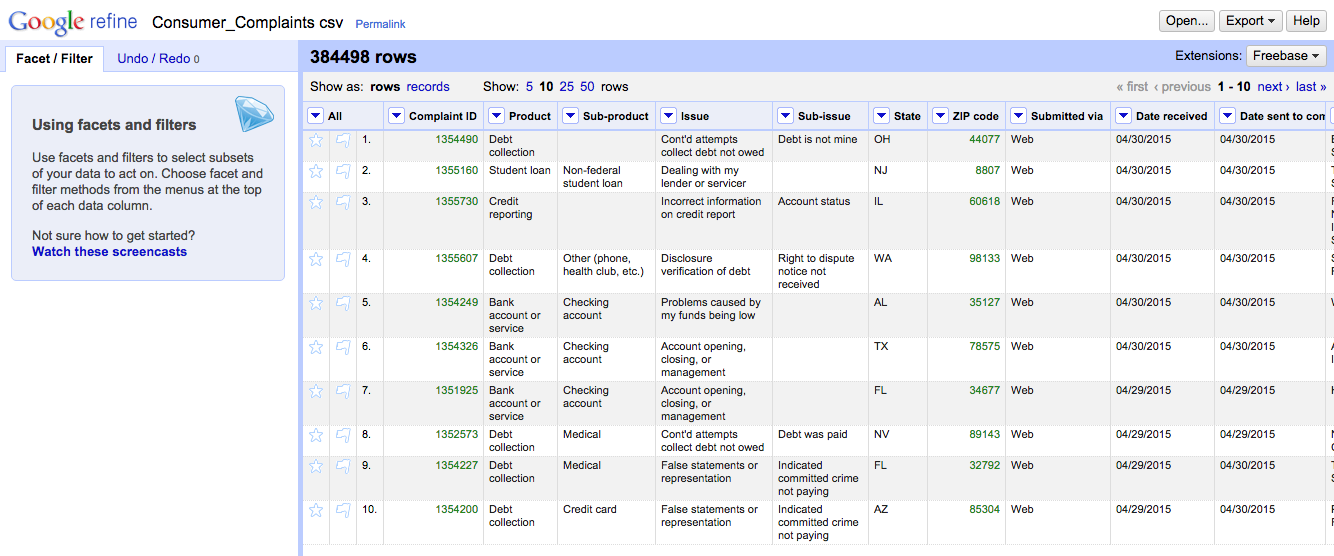
Note that we specified that the first line should be parsed as column headers.

## Creating a project

Creating the project can take a little time because there are more that 300,000 lines in this file.

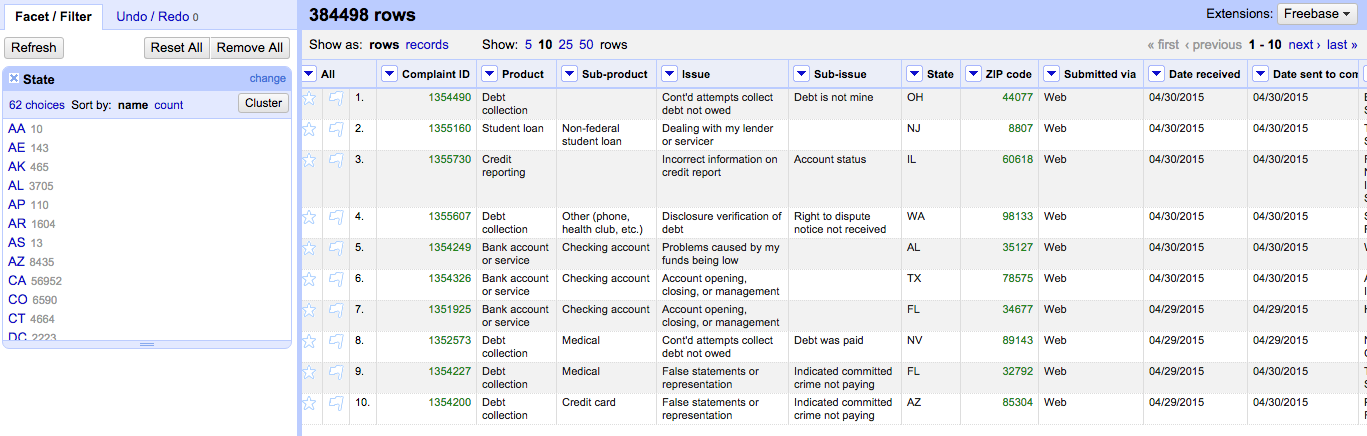


Once the project is created you can see that it has 384,498 rows.



## Check states with text facet

If you select text facet for the State attribute, you will see a summary in the left pane, indicating we have 62 different state values. Try to figure out why.



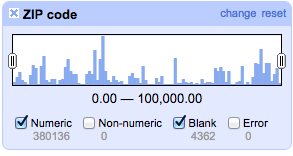
SUBMISSION 1: How many rows are missing a value in the “State” column? Explain how you came up with the number.

## Checking ZIP codes

Try the text facet on “ZIP code.” What happens? You can see that there are 24,748 different ZIP codes in this dataset. Is that reasonable? Eyeball the data—do all ZIP codes look valid? You may need to research valid ZIP codes on the Internet to determine if the values are reasonable.

Now try the numeric facet. With the numeric facet, the ZIP code attribute is treated as a numeric value. What would you say the scalar type is for ZIP codes? Can it be treated as a numeric attribute? The histogram below shows the distribution when the attribute is treated as numeric. By unchecking numeric, you can get a list of rows that are missing.

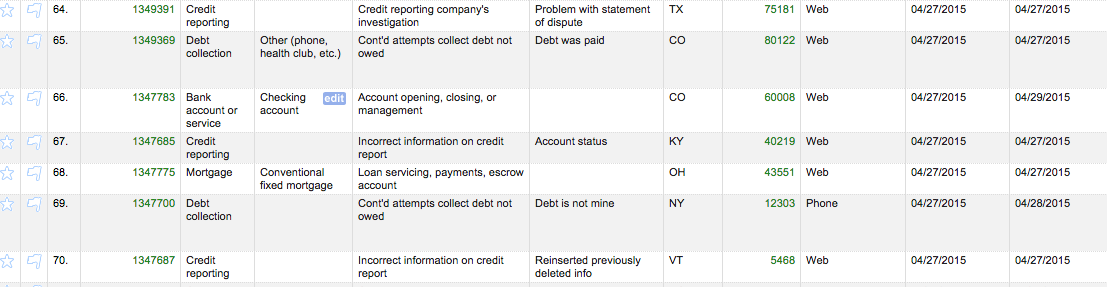
SUBMISSION 2: How many rows with missing ZIP codes do you have?



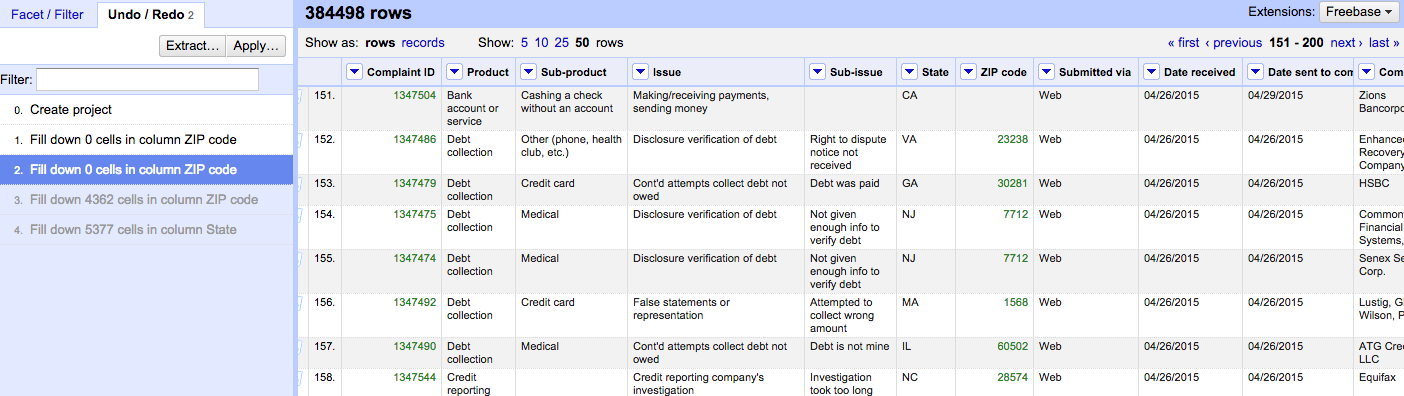
One way of filling in missing values is to take the previous value and use that to set subsequent empty cells. In OpenRefine, this is called fill down. Find a row that is blank. Apply fill down using:

Edit Cell->Fill Down

What happened to the empty cell? Is this a valid way filling in missing ZIP codes. Can you think of a better way?



If you need to undo the operation, switch to the Undo/Redo tab. Select the previous state for the data. In this example, I went back to state 2. As you can see in this screenshot, row 151 has a missing ZIP code, so presumably the fill downs for ZIP code and State have been undone. Observe that the list in Undo/Redo can look different if you have been issuing more or different commands than we have so far in this lab.



Let’s create a new column called “ZipCode5” with all ZIP codes that contain five digits preserved. All other rows should have the ZIP code 99999. (Technically speaking the four-digit ZIP codes may be valid; we do this more to illustrate transformations.)

Transformations are generally expressed in some language. OpenRefine supports a few alternative languages for transform; we will be using GREL. You can find a link to a language reference in the resources section. For this simple transformation we will be using an if-statement.



For the “ZIP code” column select:

Edit Column -> Add column based on this column.

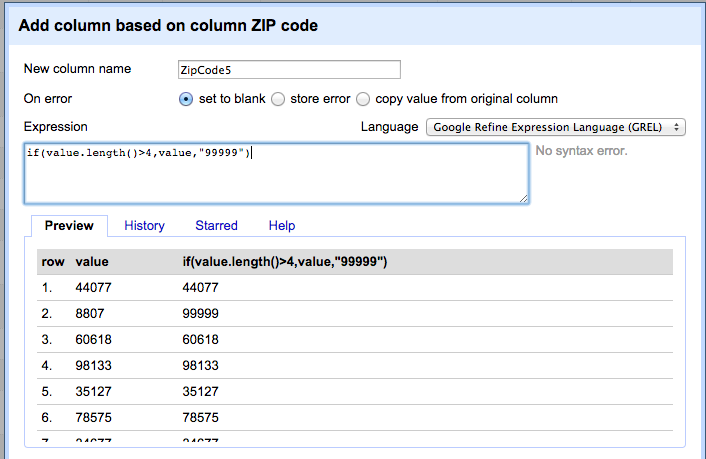
The dialogue box below will open. Insert the name of the new column and the expression:

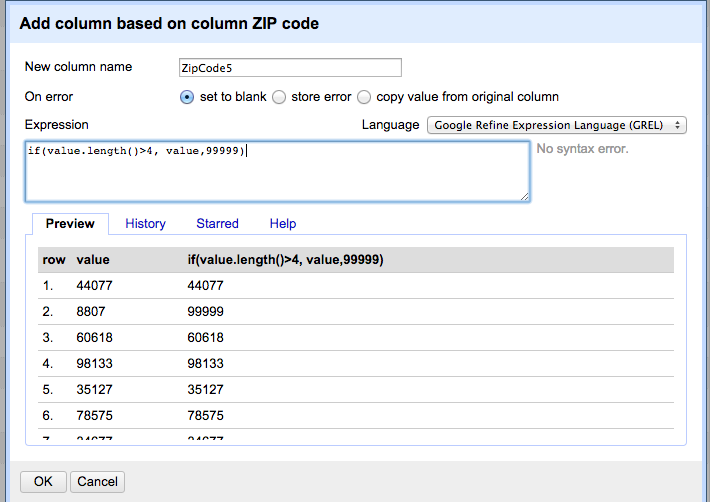
If(value.length() > 4, value, “99999”)

This expression states that if the length of value is more than four, insert the value; otherwise, insert the string “99999”.

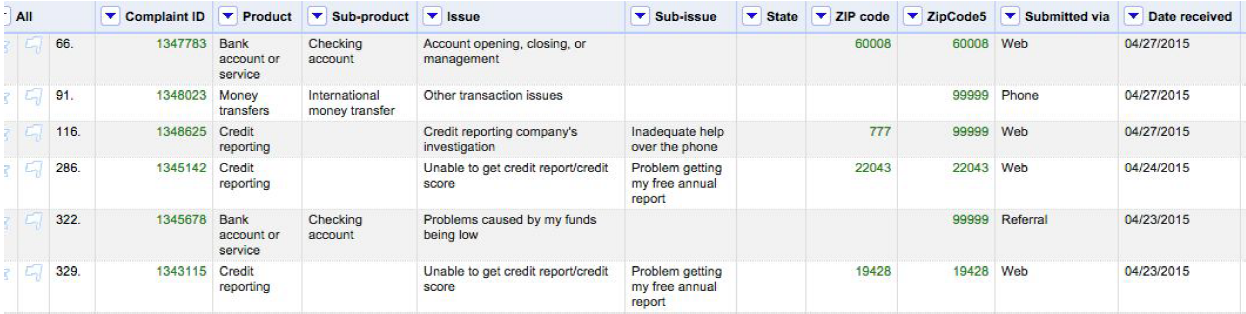
Look at the result. Did this do what you wanted? What seems to be wrong with that? What happens if you instead insert a numeric value using the following expression:

If(value.length() > 4, value, 99999)





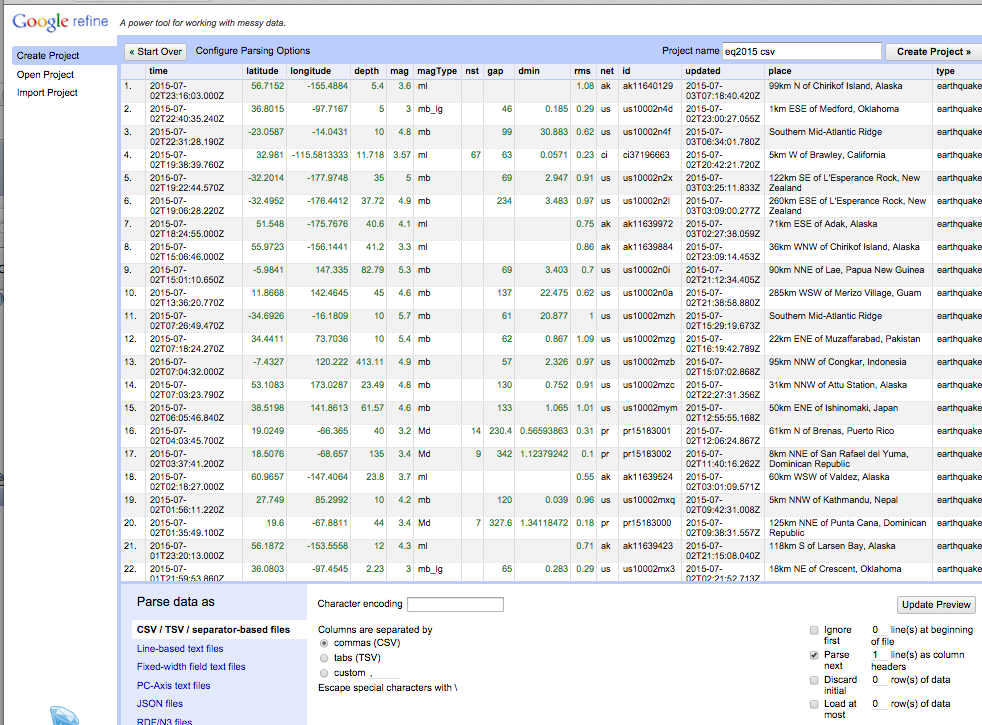
You should now have the same type for all cells in the created column. As an example, the result should look something like the following:



SUBMISSION 3: If you consider all ZIP codes less than 99999 valid ZIP codes, how many valid and invalid ZIP codes do you have, respectively?

# Step 2. Cleaning Up eq2015 Data.

Upload the dataset eq2015.csv. After you verify that the data look OK, create the project.



As you can see the “nst” column is missing quite a few values. Look up the nst attribute in the glossary. What would happen if we just ignored a row with missing values? Is there an obvious strategy for filling in the missing values? What would you suggest we do with the column?

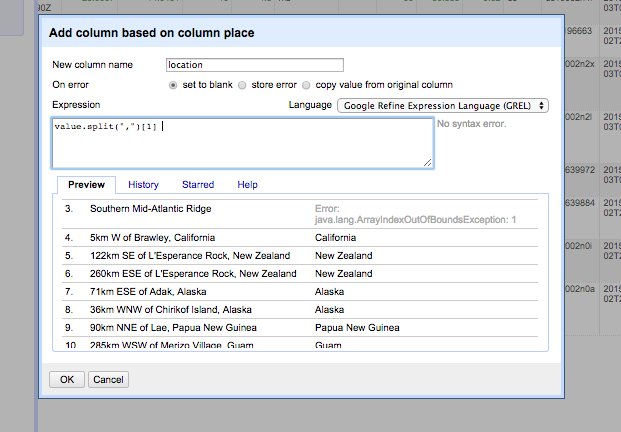
Next, we want to extract an approximate area from the “place” column. We would like to have a state or country, and we like to store that information in a separate column called “location.”

As we review the “place” column, we notice that the cell seems to consist of two comma-separated components. The components are a direction and distance and a general location.

Select the command:

Edit Column -> Add Column based on this column

You should see the following dialogue box. We typed in the column name of the new column “location.”



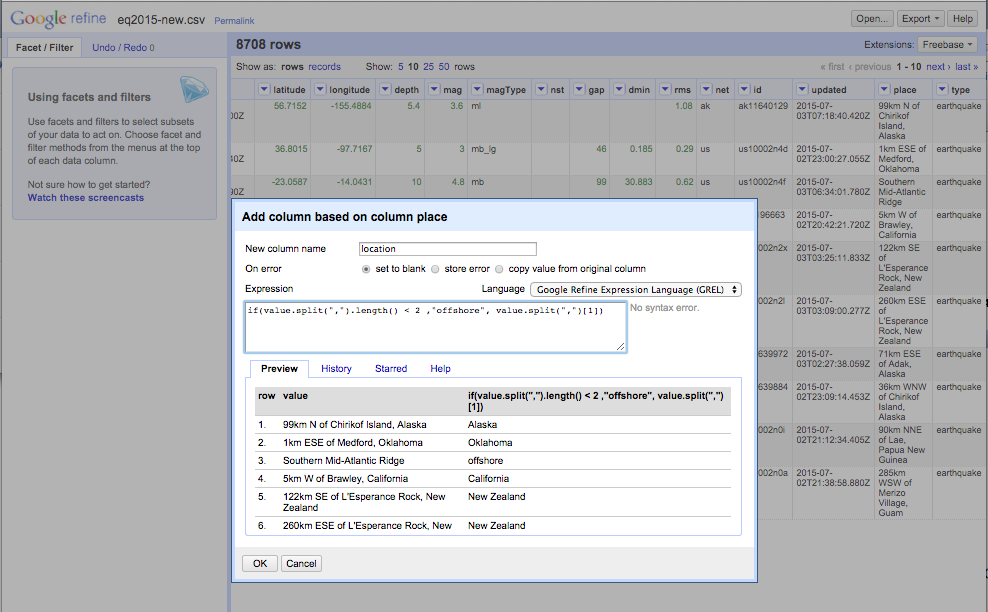
Since we noticed the cells have two comma-separated components, and the second is a location, we defined the following expression:

value.split(",")[1]

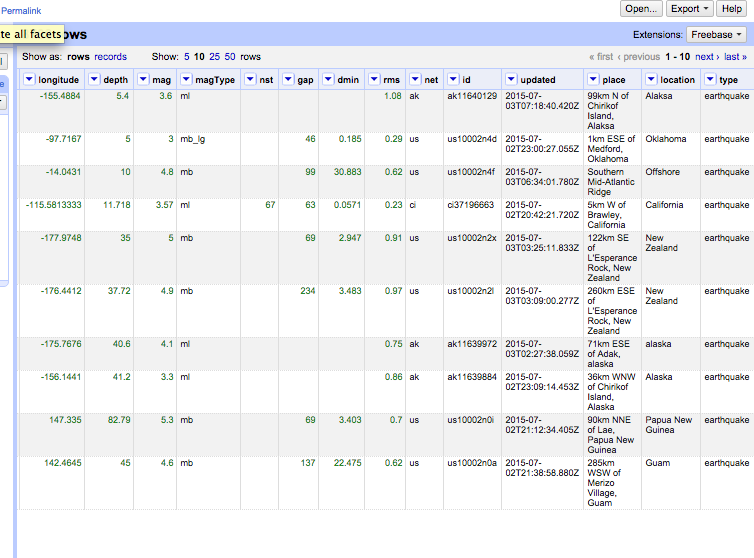
But as you probably noticed, this did not work well. In fact, not all cells have the two components. If you look at the data more closely, it appears that if a place is offshore, the location component was missing. So we modify the expression as follows:

if(value.split(",").length() < 2 ,"Offshore", value.split(",")[1])

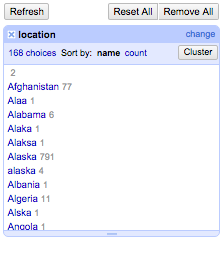
If a cell has only one component, we assume it is Offshore and put that value in the “location” column.



Check the resulting data. Do they seem reasonable, or are more adjustments needed?



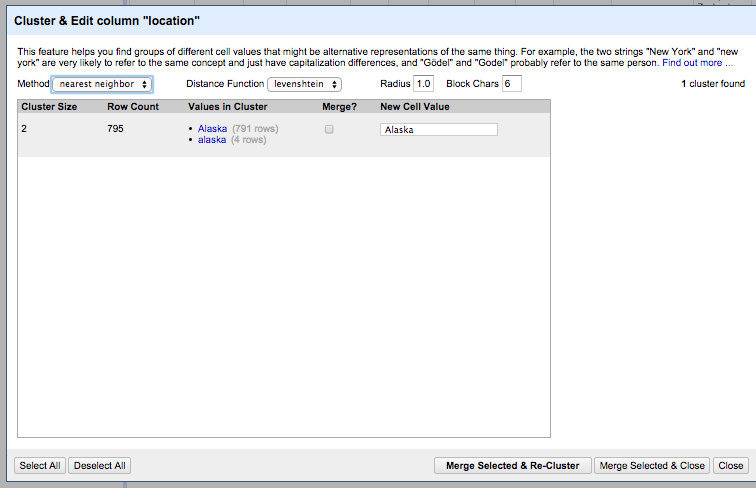
Check the value by using a text facet on the column. You may notice that there are multiple strings that look like “Alaska,” but they appear to be misspelled.



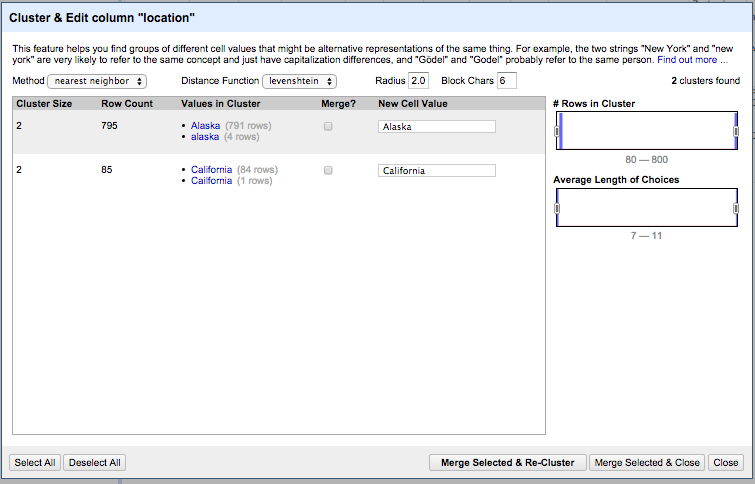
Clustering may help us detect more of these kinds of situations. Run clustering by clicking **Cluster** on the facet or using the column pop-up menu and selecting:

Edit Cell->Cluster and edit

Try key collision. What do you see? Try nearest neighbor and Levenshtein. What do you see? You can change the parameters such as Radius and Block Chars. Radius provides a threshold for how close (in terms of distance measure) the strings should be to be considered representing the same entity. The Block Char parameter may be a little counterintuitive. Blocking defines blocks within which the string distance method is applied. It helps with scalability, because we will not compare strings across the whole dataset. The OpenRefine blocking parameter defines the size of a substring S, such that all strings that share S will be in a common block. So a smaller S will likely result in bigger blocks and more computation required.



Change the radius to 2.0. What happens?

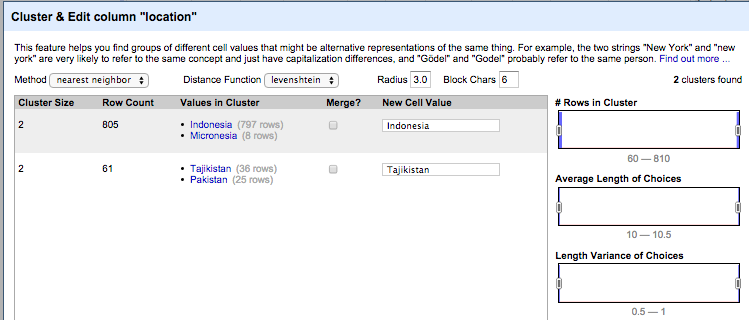


SUBMISSION 5: Change the radius to 3.0. What happens? Do you want to merge any of the resulting matches?

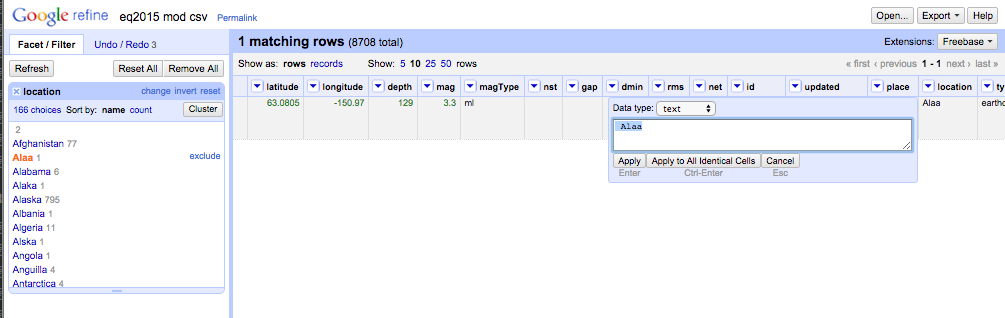
Change the block size to 2 and run the clustering.

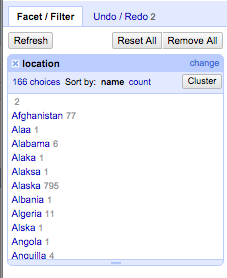
SUBMISSION 6: Change the block size to 2. Give two examples of new clusters that may be worthwhile merging.

You can try different parameters to see if you can catch the issues you see. If not, you can also note that there are a few misspellings of “Alaska” that occur only once. Hence, it is doable to go in and edit them by hand.



If you review the facet you may still see values that seem wrong but were not caught. If these are single values, the easiest fix is probably to go in and edit those cells. You can access the values by clicking the facet widget.





However, consider that if you had a very large dataset and you wanted to automate the cleaning, manual editing would not be feasible.

The “place” column strings are significantly longer than the strings for location. Try to do nearest neighbor clustering on the “place” column. What happens and why? How does the user experience compare with the clustering of the “location” column?

SUBMISSION 7: Explain in words what happens when you cluster the “place” column, and why you think that happened. What additional functionality could OpenRefine provide to possibly deal with the situation?

Hint: It takes a long time; in fact, you may want to cancel the run.

# Step 3. Levenshtein Distance

## Introduction

In this part of the lab we will go over a simple example of the Levenshtein distance calculation. We will then ask you to calculate the distance for two strings: “gumbarrel” and “gunbarell.” We will point you to a Python implementation of the Levenshtein distance that you can use to check your result.

## Installing the Levenshtein Python module

The following procedure just clones and builds a Python Levenshtein module in a directory. It does not fully install the module. But you can use it to run a distance function from your shell to check your results by running the Python shell in the Levenshtein subdirectory.

$ git clone <https://github.com/ztane/python-Levenshtein/>

$ cd python-Levenshtein/

$ python setup.py build

$ cd Levenshtein/

$ python

>>> from Levenshtein import \*

>>> distance("hej","hei")

1

>>> distance("monthgomery st","montgomery street")

5

If you want to execute this function from another location. Set you PYTHONPATH. If we assume you installed Levenshtein in the following directory: /tmp/python-Levenshtein-0.12.0/Levenshtein

Define PYTHONPATH as follows and then restart python and call the function.

export PYTHONPATH=$PYTHONPATH:"/tmp/python-Levenshtein-0.12.0/Levenshtein"

## Example: Levenshtein calculation

Let’s step through the calculation of distance between the words LOYOLA and LAJOLLA. We will denote a cell with d[i,j], where i is the row and j is the column. The dark column and row indicates the index number we will be using for the actual calculation matrix.

As a reminder the algorithms is as follows:

Denote the row by r and column by c. We have n rows and m columns.

d[i,j] denotes the value on row i and columns j.

d[i,j] is to be set to the minimum of: d[i-1,j]+1 or d[i,j-1]+1 or d[i-1, j-1]+cost[i,j]

Distance is found in the resulting value d[n,m]

We first set up the matrix. The dark row and column contain the i and j values. We then insert values 0-m in first row i==1 and 0-n in the column j==1.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|  |  |  | L | O | Y | O | L | A |
| 1 |  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | L | 1 |  |  |  |  |  |  |
| 3 | A | 2 |  |  |  |  |  |  |
| 4 | J | 3 |  |  |  |  |  |  |
| 5 | O | 4 |  |  |  |  |  |  |
| 6 | L | 5 |  |  |  |  |  |  |
| 7 | L | 6 |  |  |  |  |  |  |
| 8 | A | 7 |  |  |  |  |  |  |

Let’s calculate d[i,2], ,meaning the value for each row in column 2.

d[2,2], cost is 0, minimum is d[1,1]+0=>0

d[3,2], cost is 1, minimum is d[2,2]+1=>1

d[4,2], cost is 1, minimum is d[3,2]+1=>2

d[5,2], cost is 1, minimum is d[4,2]+1=>3

d[6,2], cost is 0, minimum is d[5,1]+0=>4, or d[5,2]+1

d[7,2], cost is 0, minimum is d[6,2]+0=>5, or d[6,2]+1

d[8,2], cost is 1, minimum is d[7,2]+1=>6

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|  |  |  | L | O | Y | O | L | A |
| 1 |  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | L | 1 | 0 |  |  |  |  |  |
| 3 | A | 2 | 1 |  |  |  |  |  |
| 4 | J | 3 | 2 |  |  |  |  |  |
| 5 | O | 4 | 3 |  |  |  |  |  |
| 6 | L | 5 | 4 |  |  |  |  |  |
| 7 | L | 6 | 5 |  |  |  |  |  |
| 8 | A | 7 | 6 |  |  |  |  |  |

Let’s calculate d[i,3], meaning the value for each row in column 3.

d[2,3], cost is 1, minimum is d[2,2]+1=>1

d[3,3], cost is 1, minimum is d[2,2]+1=>1

d[4,3], cost is 1, minimum is d[3,2]+1=>2, or d[3,3]+1

d[5,3], cost is 0, minimum is d[4,2]+0=>2

d[6,3], cost is 1, minimum is d[5,3]+1=>3

d[7,3], cost is 1, minimum is d[6,2]+1=>4, or d[6,3]+1

d[8,3], cost is 1, minimum is d[7,2]+1=>4

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|  |  |  | L | O | Y | O | L | A |
| 1 |  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | L | 1 | 0 | 1 |  |  |  |  |
| 3 | A | 2 | 1 | 1 |  |  |  |  |
| 4 | J | 3 | 2 | 2 |  |  |  |  |
| 5 | O | 4 | 3 | 2 |  |  |  |  |
| 6 | L | 5 | 4 | 3 |  |  |  |  |
| 7 | L | 6 | 5 | 4 |  |  |  |  |
| 8 | A | 7 | 6 | 5 |  |  |  |  |

If you do the same thing for the remaining columns, you will get the following matrix. You see the calculated edit distance in cell d[8,7].

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|  |  |  | L | O | Y | O | L | A |
| 1 |  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | L | 1 | 0 | 1 | 2 | 3 | 4 | 5 |
| 3 | A | 2 | 1 | 1 | 2 | 3 | 4 | 4 |
| 4 | J | 3 | 2 | 2 | 2 | 3 | 4 | 5 |
| 5 | O | 4 | 3 | 2 | 3 | 2 | 3 | 4 |
| 6 | L | 5 | 4 | 3 | 3 | 3 | 2 | 3 |
| 7 | L | 6 | 5 | 4 | 4 | 4 | 3 | 3 |
| 8 | A | 7 | 6 | 5 | 5 | 5 | 4 | **3** |

If you use the Levenshtein function to check the result, you will see the following:

>>> distance("loyola","lajolla")

3

So we are assuming we got the calculation right.

## Calculation: “gumbarrel” vs. “gunbarell”

Now calculate the edit distance between the words “gumbarrel” and “gunbarell.” After you are done, use the Python Levenshtein function to check your result.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|  |  |  | G | U | M | B | A | R | R | E | L |
| 1 |  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2 | G | 1 |  |  |  |  |  |  |  |  |  |
| 3 | U | 2 |  |  |  |  |  |  |  |  |  |
| 4 | N | 3 |  |  |  |  |  |  |  |  |  |
| 5 | B | 4 |  |  |  |  |  |  |  |  |  |
| 6 | A | 5 |  |  |  |  |  |  |  |  |  |
| 7 | R | 6 |  |  |  |  |  |  |  |  |  |
| 8 | E | 7 |  |  |  |  |  |  |  |  |  |
| 9 | L | 8 |  |  |  |  |  |  |  |  |  |
| 10 | L | 9 |  |  |  |  |  |  |  |  |  |

SUBMISSION 8: Submit a representation of the resulting matrix from the Leveshtein edit distance calculation. The resulting value should be correct.