CS 838 (Spring 2017): Data Science
Project Report - Stage 4 (Group 12)

Deepanker Aggarwal
deepanker@cs.wisc.edu

Saket Saurabh
ssaurabh@cs.wisc.edu

Vishnu Lokhande
lokhande@cs.wisc.edu

1 Objective

In the previous stage, a matcher was built which can determine whether two tuples, one from table A and the other from table B, denote the same entity or not. The matcher was trained on small sample datasets from table A and table B.
In this stage, we apply the trained matcher on tables A and B to determine the set of all matching tuple pairs and then later merge these tuple pairs to form a single table.

2 Choice of the matcher

- From the previous stage, upon training and testing several matchers on a sample of dataset, we found that Naive Bayes performed better than other matchers. This is based on the test-set accuracies.
- Next, we train the selected matcher on the combined set of training and testing data. This is to improve the performance of the matcher before taking it to the production stage.

3 Deploying the matcher and creating Table E

- We first generate the set of all tuple pairs of tables A and B. We do blocking using attribute equivalence blocker on 'zip-code' and overlap blocker on 'address'.
- The table A is the restaurants data from Yelp and it has following schema (name, address, zipcode, cuisine, price). Table B is the NYC Restaurant Inspection dataset and it has the following schema (name, address, zipcode, cuisine, violation_code, critical_flag, grade).

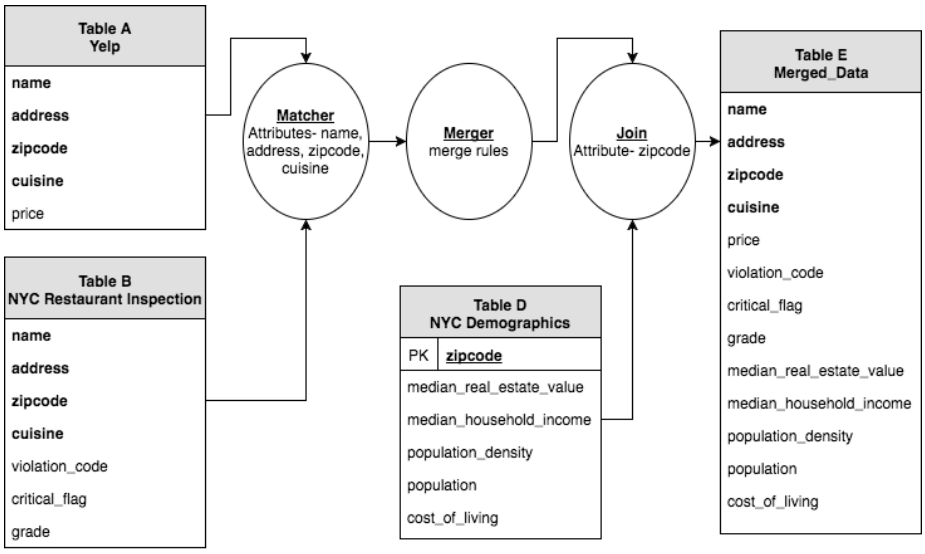

Figure 1: Schema Merging

- We apply the matcher chosen in the previous section to generate the set of all matching tuple pairs. The matcher works on the common schema between the two tables A and B which is (name, address, zipcode, cuisine)
- The positively matched tuple pairs are merged into a single tuple. Merging is done using some rules which is explained in the next section.
- At this point, we join the output of the merger with a Table D which is NYC demographic dataset. Please refer to stage 1 for an elaborate description of this table. The join is done over the "zip-code" attribute.
- The figure(3) describes this entire process as a flow-chart.

4 Merging the Tables

We do not have any missing values in our data, hence, the case is not handled. The common attributes between the tables A and tables B are (name, address, zipcode, cuisine). Rules for each of these attributes is described below.

- **Name**
  Among the two names, we choose the longer name. Upon observation of the dataset we came to know that longer names are more informative than shorter ones.
- **Address**
  We choose the address from the Yelp dataset over the Inspection dataset.

We validated the addresses by querying them using Google Maps API and determined that the addresses from the Yelp dataset were more reliable than the ones from Inspection dataset. Also, addresses from the Yelp dataset are provided by the restaurants themselves (as per the dataset description).

- **Zipcode**
  We choose zipcodes from the Yelp dataset. The zipcode values in the Yelp dataset are coming from the relational backend of Yelp making these values cleaner than the ones from the Inspection dataset.

- **Cuisine**
  We follow a more complex rule to merge cuisine data from table A and table B. Since we want the cuisine of restaurant to be as descriptive as possible, we take the union of all the cuisine type mentioned in the table A and table B.

  However, it is to be mentioned that the cuisine data is not so clean. Therefore, first we create a dictionary of all well-known cuisines that maps variations of the same cuisine to a fixed cuisine keyword. We concatenate the cuisine strings from table A and table B for each restaurant, which is then tokenized. We find the associated cuisine keyword for each token. Finally, we construct the most descriptive cuisine value for the restaurant by taking the union of all the cuisine keywords.

5 Statistics of Table E

- The schema of table is as follows (name, address, zipcode, cuisine, price, violation_code, critical_flag, grade, median_household_income, median_real_estate_value, population_density, cost_of_living, population). This can also be observed from the Figure(3).

- Table A has $16,149$ tuples and table B has $20,201$ tuples. After blocking, the candidate set has $216,213$ tuples. The number of tuples in the Table E are $5,561$.

- Some sample tuples from table E are as follows:

  1. (juniper,"237 w 35th st,",10001,bars american,$$,06C,Critical,A, 81671,650200,35350,157.4,21966)
  2. (mission chinese food,"171 e broadway, ,",10002,desserts bars chinese,$$,10F,Not Critical,A,33218,535600,93461,168.9,82191)
  3. (okinii,"216 thompson st, ,",10012,japanese bars,$$,10B,Not Critical,A, 86594,1000001,80873,164.7,26145)
  4. (the fitz,"fitzpatrick manhattan, 687 lexington ave,",10022,bars american irish,$$,10B,Not Critical,A,109019,866100,67873,158.2,29618)
  5. (ny sweet spot cafe,"2376 coney island ave,",11223,middle eastern european,$$,06C,Critical,A,41328,613000,35968,167.4,74606)

6 Code

```
########################
#--- data_merger.py ---#
########################
# The main python file that does data merging for matches between
# tables A and B. The file expects two command-line arguments
# and can be run as:
# ‘python3 data_merger.py <input_file> <output_file>‘
# It uses two helper files merge_rules.py and nyc_demographics.py
# that are described later in the document.

import csv
import sys

# Import our python classes that we wrote for merging. (Included below)
import merge_rules
import nyc_demographics

nyc_demographics_dataset = nyc_demographics.NYCDemographicsDataset()

def merge_attributes(row):
    merge_rules_dict = {
        'NameMergeRule': merge_rules.NameMergeRule(),
        'AddressMergeRule': merge_rules.AddressMergeRule(),
        'ZipcodeMergeRule': merge_rules.ZipcodeMergeRule(),
        'CuisineMergeRule': merge_rules.CuisineMergeRule()
    }
    merged_name =
        merge_rules_dict['NameMergeRule'].process(row["ltable_name"],
        row["rtable_name"])
    merged_address =
        merge_rules_dict['AddressMergeRule'].process(row["ltable_address"],
        row["rtable_address"])
    merged_zipcode =
        merge_rules_dict['ZipcodeMergeRule'].process(row["ltable_zipcode"],
        row["rtable_zipcode"])
    merged_cuisine =
        merge_rules_dict['CuisineMergeRule'].process(row["ltable_cuisine"],
        row["rtable_cuisine"])
    extracted_price = row["ltable_price"]
    extracted_violation_code = row["rtable_violation_code"]
    extracted_critical_flag = row["rtable_critical_flag"]
    extracted_grade = row["rtable_grade"]
    merged_row = [merged_name, merged_address, merged_zipcode,
        merged_cuisine,
               extracted_price, extracted_violation_code,
                   extracted_critical_flag, extracted_grade]
```

```
    # Add the attributes from Table D: NYC Demographics Dataset
    merged_row.extend(nyc_demographics_dataset.find(merged_zipcode))
    return merged_row

def write_header(f):
    # Write CSV Header
    f.writerow(["name", "address", "zipcode", "cuisine", "price",
                "violation_code", "critical_flag", "grade",
                "median_household_income", "median_real_estate_value",
                "population_density", "cost_of_living", "population"])

def write_row(f, merged_row):
    f.writerow(merged_row)

def main(argv):
    if len(argv) < 2:
        sys.stderr.write("Incorrect arguments. Expecting two arguments:
            <input_file> <output_file>\n")
        sys.exit(-1)

    input_file = csv.DictReader(open(argv[0]))
    output_file = csv.writer(open(argv[1], "w"))
    write_header(output_file)

    for row in input_file:
        merged_row = merge_attributes(row)
        write_row(output_file, merged_row)

if __name__ == "__main__":
    main(sys.argv[1:])


########################
#--- merge_rules.py ---#
########################
# This is a helper python file that contains classes that define
# rules for merging columns of A and B.
# Specifically, it defines following classes:
# class NameMergeRule- to merge restaurant names,
# class AddressMergeRule- to merge restaurant addresses,
# class ZipcodeMergeRule- to merge restaurant zipcodes,
# class CuisineMergeRule- to merge restaurant cuisines.


import json
import re

class NameMergeRule:
    @staticmethod
    def process(value_l, value_r):
```

```
        # After inspection of the dataset, we find that the longer names
        # are more descriptive and hence better.
        if len(value_l) > len(value_r):
            return value_l
        else:
            return value_r


class AddressMergeRule:
    @staticmethod
    def process(value_l, value_r):
        if value_l:
            # Trust address from Yelp(value_l) over
            # Inspection dataset(value_r) because it is more reliable,
            # given that it is provided by the restaurant itself
            # and is validated by calling the Google Map API.
            return value_l
        else:
            return value_r


class ZipcodeMergeRule:
    @staticmethod
    def process(value_l, value_r):
        if value_l:
            # Trust zipcode from Yelp(value_l) over
            # Inspection dataset(value_r) because
            # it is coming from a relational database backend of Yelp,
            # extracted using Yelp API.
            return value_l
        else:
            return value_r


class CuisineMergeRule:
    valid_cuisine_map = {}

    def __init__(self):
        with open('cuisines_dictionary.json') as json_data:
            self.valid_cuisine_map = json.load(json_data)

    def process(self, value_l, value_r):
        # For cusinies, we merge all the cusinie names mentioned in
        # both value_l and value_r. However, we pick only those names
        # that exist in the valid dictionary of cuisines,
        # which we have created.
        merged_cuisine = set()
        cuisines = re.split('\W', value_l + " " + value_r)
        for cuisine in cuisines:
            if cuisine in self.valid_cuisine_map:
```

```
                # add the cusinie associated with this key to the set
                merged_cuisine.add(self.valid_cuisine_map[cuisine])

        return " ".join(merged_cuisine)


##############################
#--- nyc_demographics.py ---#
##############################
# This is a helper python file that loads
# the NYC Demographics dataset (Table D).

# Given a zipcode, the find() function of
# the class NYCDemographicsDataset
# returns the corresponding demogrpahic data for that zipcode.

import csv

class NYCDemographicsDataset:
    zipcode_data_map = {}

    def __init__(self):
        input_file = csv.DictReader(open("../data/Table_D.csv"))
        for row in input_file:
            data = [
                        row["median_household_income"],
                        row["median_real_estate_value"],
                        row["population_density"],
                        row["cost_of_living"],
                        row["population"]
                    ]
            self.zipcode_data_map[row["zipcode"]] = data


    def find(self, zipcode):
        if zipcode in self.zipcode_data_map:
            return self.zipcode_data_map[zipcode]
        else:
            return []


##################################
#--- cuisines_dictionary.json ---#
##################################
# This is the dictionary of well-known cuisines that we use
# when merging cuisines from Table A and Table B.
# The key describes the possible derivatives/variations associated
# with a cuisine value.

{
```

```
"bars": "bars",
"cocktail": "bars",
"beer": "bars",
"wine": "bars",
"pubs": "bars",
"gastropubs": "bars",
"american": "american",
"burgers": "burgers",
"burger": "american",
"hot": "american",
"dogs": "american",
"hamburgers": "american",
"steak": "american",
"pancakes": "american",
"waffles": "american",
"japanese": "japanese",
"sushi": "japanese",
"korean": "korean",
"mexican": "mexican",
"tacos": "mexican",
"barbeque": "barbeque",
"barbecue": "barbeque",
"french": "french",
"brasseries": "french",
"creperies": "french",
"greek": "greek",
"desserts": "desserts",
"ice": "desserts",
"cream": "desserts",
"yogurt": "desserts",
"gelato": "desserts",
"italian": "italian",
"pizza": "italian",
"latin": "latin",
"cuban": "cuban",
"dominican": "dominican",
"puerto rican": "puerto rican",
"puerto": "puerto rican",
"rican": "puerto rican",
"chinese": "chinese",
"noodles": "chinese",
"ramen": "chinese",
"asian": "asian",
"jewish": "jewish",
"kosher": "jewish",
"cambodian": "cambodian",
"thai": "thai",
"german": "german",
"taiwanese": "taiwanese",
"vietnamese": "vietnamese",
```

```
"cafe": "cafe",
"donuts": "cafe",
"coffee": "cafe",
"tea": "cafe",
"juice": "cafe",
"smoothies": "cafe",
"cafe": "cafe",
"bagels": "cafe",
"pretzels": "cafe",
"cafeteria": "cafe",
"vegan": "vegan",
"vegetarian": "vegan",
"filipino": "filipino",
"mediterranean": "mediterranean",
"indian": "indian",
"hawaiian": "hawaiian",
"caribbean": "caribbean",
"turkish": "turkish",
"middle eastern": "middle eastern",
"middle": "middle eastern",
"eastern": "middle eastern",
"falafel": "middle eastern",
"seafood": "seafood",
"venezuelan": "venezuelan",
"spanish": "spanish",
"malaysian": "malaysian",
"bakery": "bakery",
"bakeries": "bakery",
"british": "british",
"european": "european",
"irish": "irish",
"pakistani": "pakistani",
"halal": "pakistani",
"scandinavian": "scandinavian",
"brazilian": "brazilian",
"senegalese": "senegalese",
"african": "african",
"cantonese": "cantonese",
"portuguese": "portuguese",
"malaysia": "malaysia",
"peruvian": "peruvian",
"colombian": "colombian",
"bangladeshi": "bangladeshi",
"hookah": "bangladeshi",
"salvadoran": "salvadoran",
"chilean": "chilean",
"australian": "australian",
"russian": "russian",
"ukrainian": "ukrainian",
"uzbek": "uzbek",
```

```
  "armenian": "armenian",
  "mongolian": "mongolian",
  "afghan": "afghan",
  "haitian": "haitian",
  "trinidadian": "trinidadian"
}
```