**Big Data Computer Systems**

**Instructor: Prof. JIANGJIANG (JANE) LIU**

**Project**

**Monday, May 1, 2017**

**Bhavya Ashokkumar Dalwadi**

**L20393359**

**Implementation of association rule mining Apriority algorithm.**

**Introduction**

Affiliation manage mining tries to "find visit designs, affiliations, connections, or easygoing structures sets of things or protests in exchange database, social database, and so on i.e. to discover the connection or reliance of event of one of one thing in view of event of different things. Apriority calculation is an essential calculation for affiliation administer mining. In this venture, I’ve will execute Apriority calculation. In software engineering and information mining, Apriority is an exemplary calculation for learning partner rules, Apriority is intended to work on database containing exchanges (for instance, accumulations of things purchased by clients, or points of interest of a site frequentation). Different calculations are intended for discovering affiliation controls in information having no exchanges, or having no timestamps (DNA sequencing).

**Implementation**

**Step 1. Pivoting the data and sampling it**

Since the entire 311 data is over a million rows, I’ve decided to sample the data within.

Of all the attributes, I’ve decided that pivoting by the month will keep the sampling fair and

unbiased. Our idea was to sort the dataset by date (done before downloading on the site itself)

and simply pick, for each month, 20% of all the rows having that month. To do the 20% sampling, I’ve decided to use an algorithm called reservoir sampling, which lets us pick randomly and uniformly the 20% of rows in each month from the dataset efficiently in a single pass. Since, it is not the scope of this project to describe it thoroughly, I’ve will leave it at that. The code, although not necessary, is included in sac/generate\_CSV.py.

**Step 2. Choosing attributes (items)**

All the attributes in the 311 are not interesting. Some of them are mostly empty, such as school

information. Some attributes are almost unique for each row, such as latitude and longitude. We

therefore, chose a set of attributes that I’ve focused our attention on. This is the same set included

in our optionally submitted file in data/attr\_list.txt, which is used in generate\_CSV.py. I’ve parse

our main CSV file and do the sampling and only project on these attributes. One additional change we

did was change the Created Date attribute into a Month (i.e. 01/01/2009 12:00 AM -> January).

* Created Date
* Agency
* Complaint Type
* Location Type
* Incident Zip
* Street Name
* Borough

**Clear description of how to run program**

Run the following from the directory where you put run.sh (NOTE: you must cd to that directory before running this command):

she run.sh <INTEGRATED-DATASET-FILE> <min\_sup> <mincing>

, where:

<INTEGRATED-DATASET-FILE> is the path to the CSV file

<min\_sup> is the value of minimun support

<mincing> is the value of minimun confidence

This will produce a file called output.txt, containing the requisite output for our project.

You can run also run our scripts directly by calling

python extract\_Rule.py <INTEGRATED-DATASET> <min\_sup> <mincing> <OUTPUT-FILE>

**A clear description of the internal design of project**

I’vehave only one python script: extract\_Rule.py, with two main functions for large itemsets and

association rules respectively.

Part 1. Large Item sets

This part will extract large item sets above the given minimum support. The main functions include:

\* extractives: A priori algorithm to extract the large item sets

\* compute\_L1: step 1 for A priori algorithm to compute L\_1

\* get Candidate: candidate generation for A Priori Algorithm

Detailed description is as follows:

1. extractives: the main function for A priori algorithm. Here I’ve used the same A priori algorithm as

described in Section 2.1 of the Agrawal and Srikanth paper.

Our A priori algorithm is:

Step 1. Generate large 1-itemsets L\_1 by calling function compute\_L1.

Step 2. If the previous L\_{k-1} is non-empty, I’ve compute the candidate Cake by calling function

get Candidate. Then keep the candidates whose support is above min-sup as Lok.

Step 3. Store all the large item sets {Lok} and return

2. compute\_L1: compute the first step of A priori algorithm to get L\_1. To avoid reading the input file

multiple times, I’ve also store all the baskets in memory to speed up. After the item sets have been generated,

I’ve delete this storage to potentially reclaim memory.

3. get Candidate: function to generate the candidates Cake for A priori algorithm. Here I’ve used the similar

Apriority Candidate Generation method as described in Section 2.1.1 of the Agrawal and Srikanth paper.

The slight difference is that I’ve will never keep two items from the same column in one basket.

For example, considering the column 'Month', no basket will have two months September and

January. So, I’ve changed the join condition as "only the last items in the basket are from different columns".

This is because, unlike the transaction example, our items have different domains (in the transaction, they

were all purchased items). Therefore, it makes no sense to have an item set like {September, April}. Such

an item set will not have a non-zero support anyway but I’ve never create such an item set in the first place.

**Apriority Candidate Generation method is:**

**Association Rules**

Once I’ve have the large item sets, to get the rules, I’ve simply iterate through our k sized large

item sets (from k = 2 to the largest one). There are two optimizations I’ve can make.

First is the rule generation itself. The number of rules for all item sets is very large if I’ve do

it naively and generate all possible rules with >= 1 items in the LHS and =1 items in the RHS

for all k-items set. For instance, in a large item set of size k, the only rules I’ve need to generate

are the rules with k-1 items on the LHS and 1 item on the RHS? This follows because

all rules with < k-1 items on the LHS must have been generated for a smaller k item set.

For instance, for an item set {x,y,z}, it must have been created from {x,y} and {x,z}.

It follows that there must be an item set {y z} also since {x y z} is large. Therefore,

all rules [x] => [z] and [x] => [y] and [y] => [z] must have been generated in a smaller

sized k. This optimization is extremely useful as the number of

rules generated per k-sized item set is exactly k. Another benefit for this method is that

all rules generated are DISTINCT because the item sets themselves are distinct (by definition).

As you can see, this optimization is incredibly beneficial and makes the solution elegant.

The second optimization is as discussed in class. I’venever need to go to the data file

again to calculate the confidence for these rules. Given a rule: [LHS] => [RHS], the

confidence for this rule is Support(LHS U RHS)/Support(LHS). But for a given rule, we

have identified that Support of (LHS U RHS) is the support of the kth-item set that

generated this rule (because of optimization 1, all k-item sets only generate k-sized rules).

The Support(LHS) is easily found by looking at the k-1 item sets and finding the one that

is equal to LHS. In this way, I’ve can calculate the confidence of a rule in near constant time, which

is incredibly useful. I’ve can make this amortized constant by hashing the k-item sets so that

I’ve can find the support(LHS) in amortized constant time.

Given these optimizations, the functions for generating rules are:

\* extractRules

\* get Rules

\* get Support

1. extract Rules

This function starts at item sets of sizes 2 and proceeds till the largest size

item set, generating rules for each item set using get Rules below. It then computes

the confidence for each rule and discards it if it is lower than our minimum

confidence. The support for this rule is exactly the support for the large

item set. This is also stored in the rule. In other words, a rule is stored in the

format <LHS, RHS, cone, sup>. It stores the rule in a list for later printing.

2. get Rules

This function is extremely simple due to our optimization. It simply returns

for each element A in the set, a rule [item set-A] => [A].

3. get Support

This function simply finds the item set that matches the input item set and retrieves

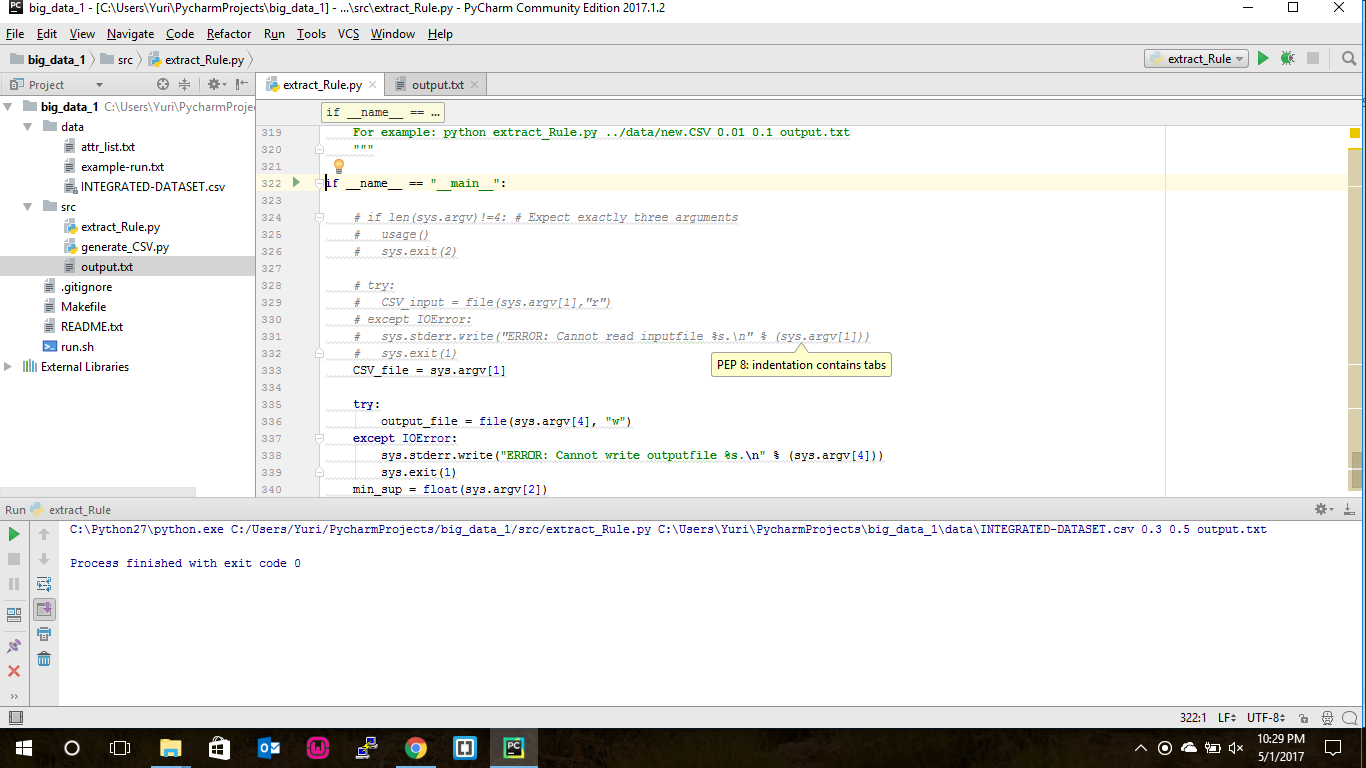
its stored (from generating large item sets) support value.

Once both the item set generation and the rule generation is done, I’ve simply

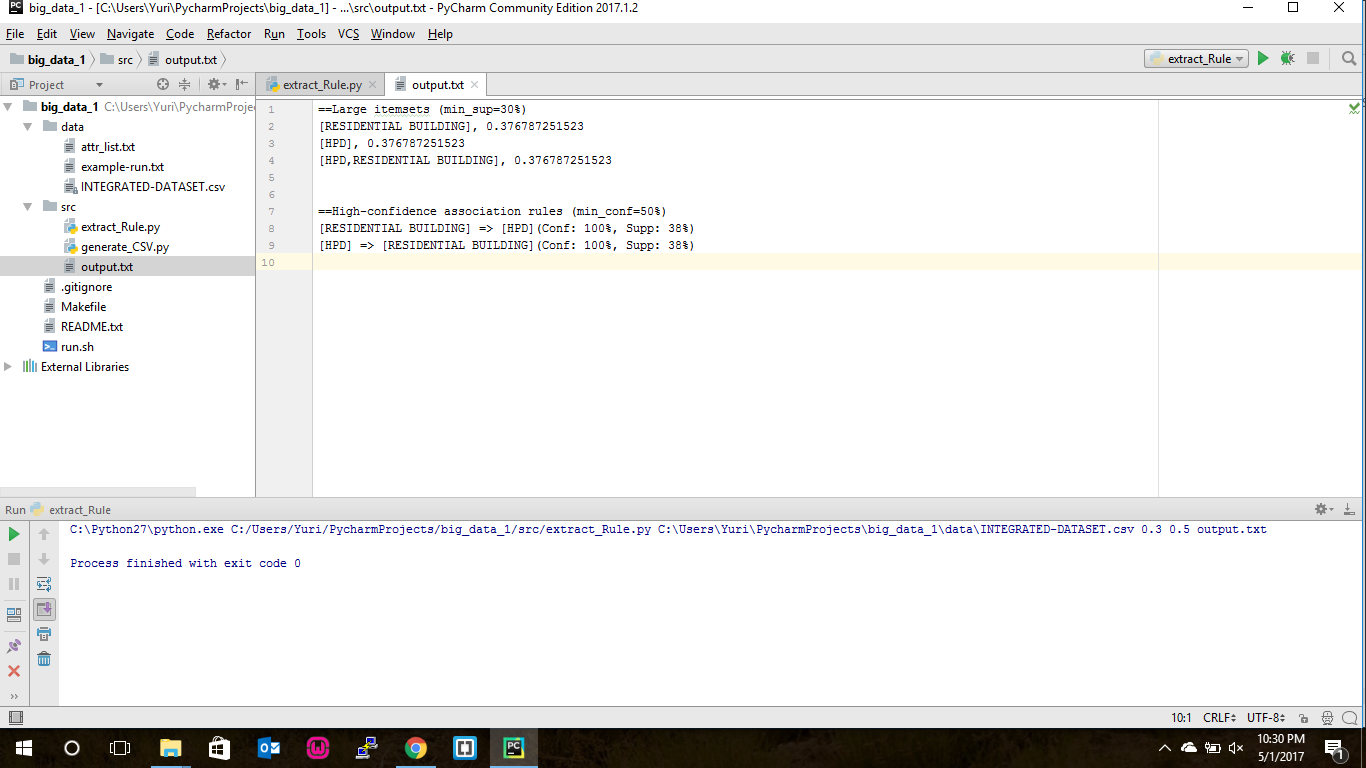
call our write File function that writes out the calculated item sets and rules

to the output file in the correct format.

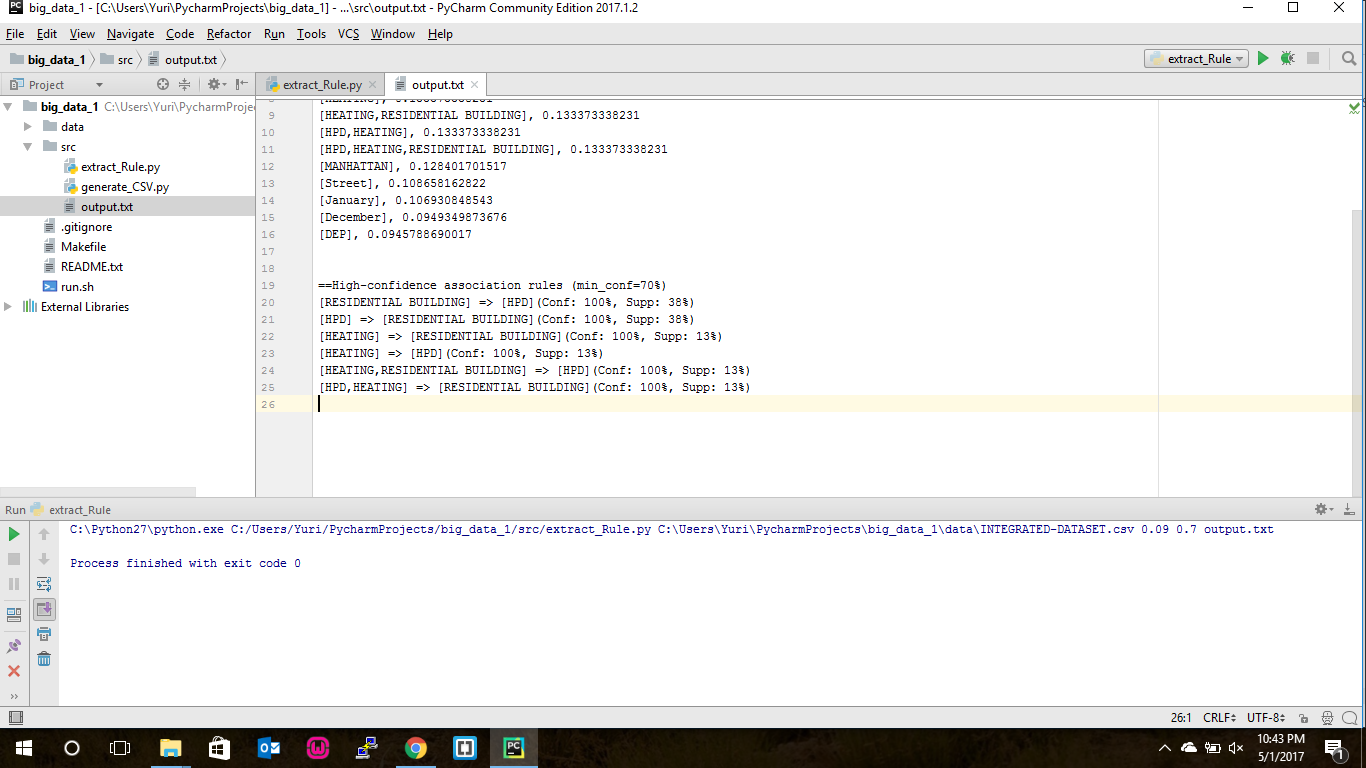
**Testing Result**



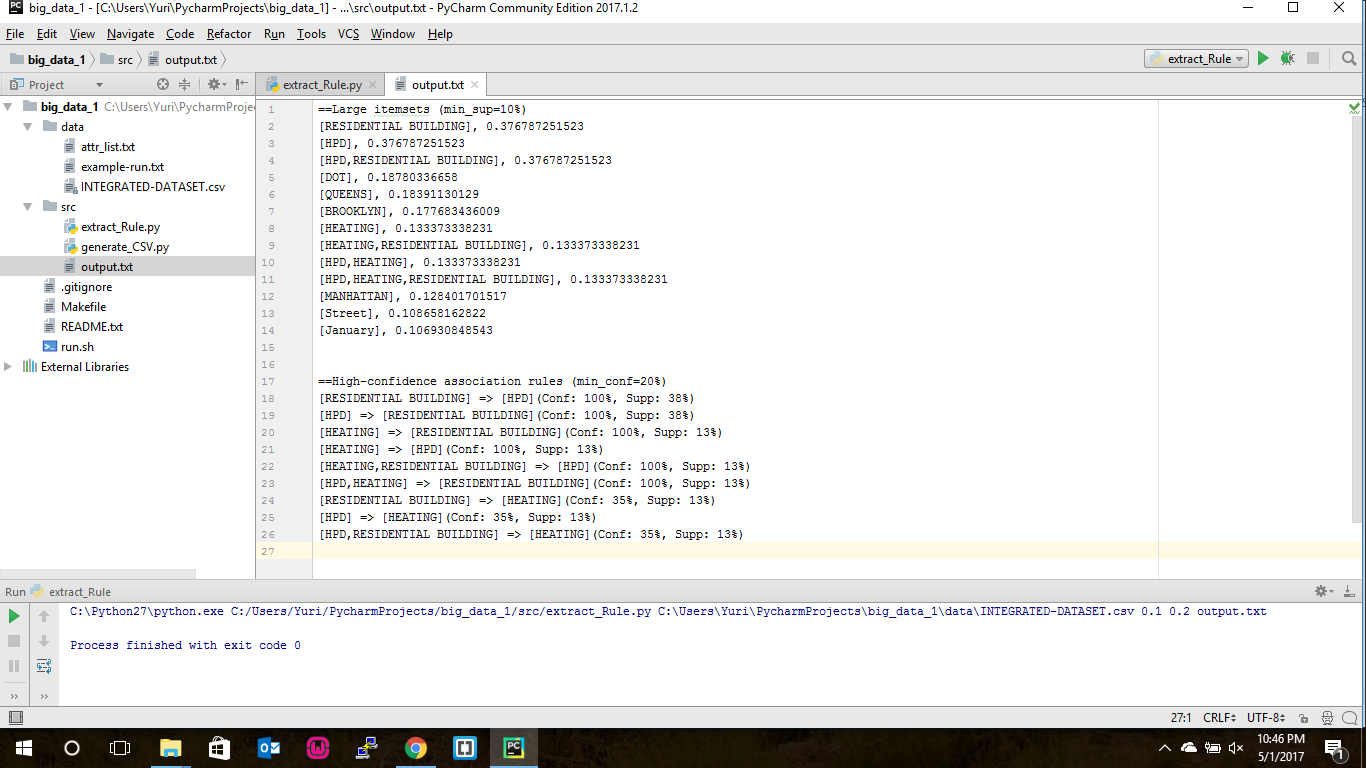
**INTEGRATED-DATASET 0.3 0.5 .png**

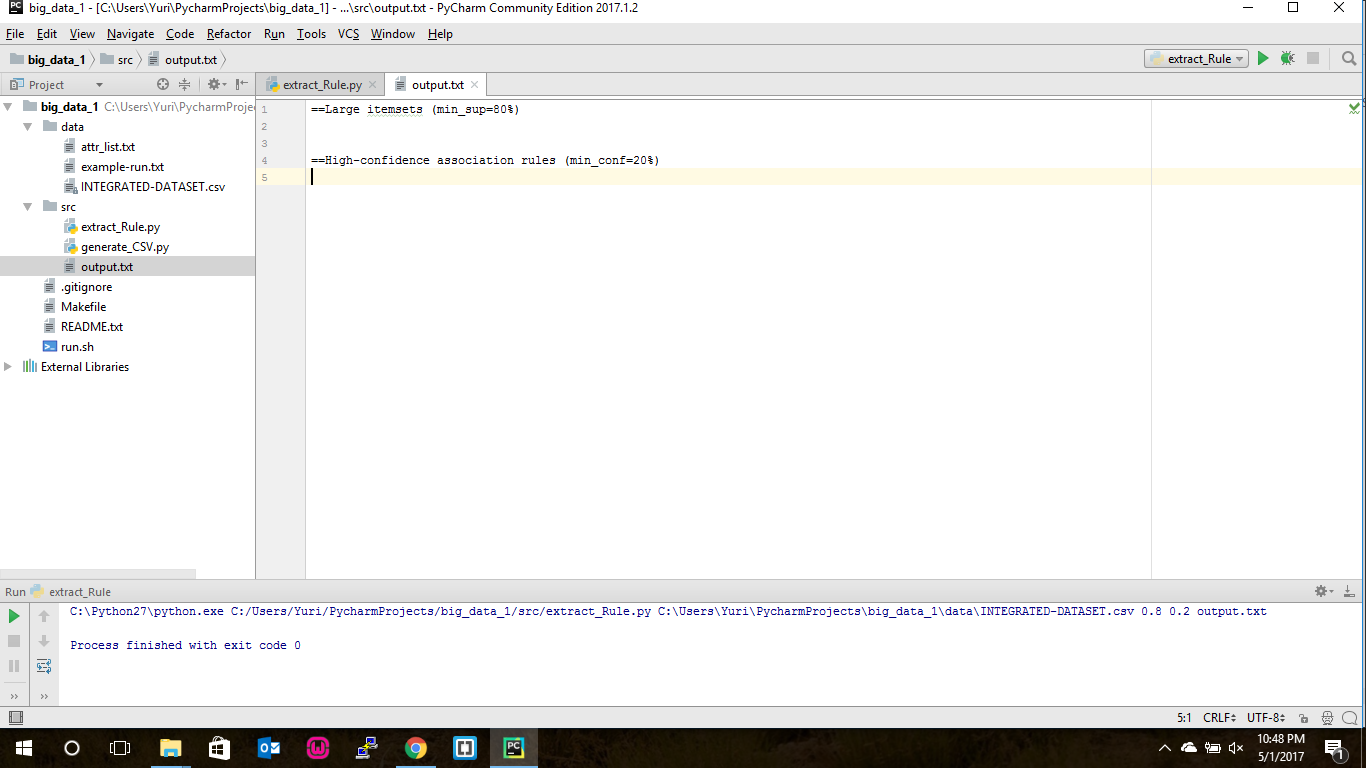
****

**OUTPUT INTEGRATED-DATASET 0.3 0.5 .png**

****

**Output for Data set : Sat Results Dataset**

****

**Output for Parking-Violation-Codes Dataset **

**Output for New-Driver-Application Dataset**