CSCI 4030 - Project I Report

Luisa Rojas-Garcia (100518772)

Introduction

One of the hardest problems when working with big data mining is finding the frequent *pairs* of items in the data set being evaluated.

The two algorithms implemented, a-priori and pcy, are some of the most popular and effective methods used for this purpose.

The implementation programs for these algorithms was written using the **Python** (*Python 2.7.10*) programming language and the **PyPy** (*PyPy 4.0.0 with GCC 4.2.1*) "just-in-time" compiler.

This is the programming language that was chosen for this project because of its clean and straight-forward syntax and large standard library, which provided great flexibiliy when working with different data structures to achieve the required results.

Additionally, Pypy was used over CPython due to its focus on speed and high memory usage optimization. Note that PyPy works best when executing long-running programs, which applies to the both algorithms implemented in for project.

Compiling and running the code

A short script was used to run the respective code with varying support threshold basket sample size. *Note that the bash script applied is an adaptation of a script code provided by Alexandar Mihaylov*.

The support threshold used for both algorithms was `200`, `500`, `1000`, and `10,000`. A threshold of `2` was attempted for both methods; however, it took over 11 hours to complete, and therefore it could not be used. Instead, the support of `200` was ran.

Each program can be compiled manually the following way:

```
luisarojas  [CSCI 4030] $ pypy <apriori/pcy>.py <support> <baskets>
```

The code will not run unless two arguments are provided.

A-priori Algorithm

This approach limits the need for main memory. Its key idea is *monotonicity*.

For example, if a set of items ***I*** appears at least ***s*** (support threshold) times, then so does every subset ***J*** of ***I*** and vice-versa.

This algorithm consists of two passes and was implemented in the following way:

Pass 1

In this pass, the baskets - represented as lines in the file - are read in and the occurences of each individual item is counted.

To keep track of this information, the data structure used was a dictionary, where keys (items) are each assigned a value (count): `items_count = dict()`.

For each item in the basket being read, if the item is not already in the dictionary, then add it and set it's count to `1`; otherwise, increment the count by `1` for that item (key):

```
if item in items_count:
    items_count[item] += 1
else:
    items_count[item] = 1
```

Then, they are filtered so that only those items that reach the support threshold are taken into account in a **list**, since *item* counts are no longer relevant at this point: `freq_items = list()`.

Between Passes

After pass 1 - but before pass 2 - candidate pairs are created. These are pairs of frequent items that could potentially be frequent themselves.

These pairs are stored in a second dictionary and its count is initialised to `0`, since the number of appearances of said pair in the dataset is still completely unknown:

```
freq_pairs_count = dict()
freq_pairs_count[(freq_items[i], freq_items[j])] = 0
```

Pass 2

In this pass, the goal is to make sure that the candidate pairs of frequent items calculated in the "Between Passes" stage are *actually* frequent **pairs**.

The baskets are read in again and the occurences of each pair created is counted in the dictionary created previously:

```
if(pair[0] in basket and pair[1] in basket):
    freq_pairs_count[pair] += 1
```

Once finished, print to the terminal only those pairs that reached the support threshold set: the *final* frequent pairs of items.

Park-Chen-Yu (PCY) Algorithm

Pass 1

Like in a-priori, the count of each item is tracked using the same data structure. Additionally, however, a "hash table" is maintained; each pair of items is hashed to a bucket and it's count is stored and kept track of:

```
hashed_val = hash(int(pair[0]), int(pair[1]))
    if hashed_val in pairs_hash_map:
        pairs_hash_map[hashed_val] += 1
    else:
        pairs_hash_map[hashed_val] = 1
```

The hash function used is `x ^ y % bucket_filed`, `x` and `y` each being the first and second value of a specific pair of items. Note that the bucket field value was raised from `1,000` to `10,000` due to a high collision rate with the algorithm when using the first. The ideal value is very likely to be between them.

Finally, as before, the items are filtered and stored in a list, leaving us only with the items that meet the support threshold: frequent items.

Between Passes

Before pass 2, the buckets need to be replaced by a bit-vector or bitmap, where `1` means the bucket count exceeded the support threshold (*frequent bucket*) and `0` means it did not.

By implementing a bitmap, only 1/32th of the memory is used, since 4-byte integer counts are replaced by bits instead.

If a bucket contains a frequent pair, then the bucket itself must be frequent; however, a bucket with a total count less than the support, none of the pairs it contains can be frequent. Therefore, the pairs that hash to said bucket can be eliminated as "candidates" - even if both pairs happen to be frequent items.

A tentative option was to use the Python bitmap library: *BitMap*. However, this library will not work when using PyPy as a compiler, so it was not implemented.

Instead, a list was adapted into a bitmap using the fragments of code provided through Slack:

```
bitmap = [chr(0)]*bucket_field
for pair in sorted(pairs_hash_map.keys()):
    if pairs_hash_map[pair] >= support:
        bitmap[position/8] = chr(ord((bitmap[position/8])) | ord(chr(1 << 
(position%8))))
    position += 1
```

First, the list is initialised to `0`'s using the `chr()` function, which converts an ASCII value into its corresponding character. In this case, it would be `NUL`.

The `ord()` function, on the other hand, returns an integer representing the value of the byte, given a string of length one.

These two functions are used in order to convert the value into the type needed.

Then, candindate pairs of *frequent* items were generated. If the bucket this pair was hashed to in the bitmap is higher than `0` (i.e. `1`), then it is added to the `freq_pairs_count` dictionary with an initialised count of `0`. This is because it is still not clear if said pair even appears at all in the data set.

Pass 2

In this pass, only the pairs that hashed to frequent buckets were counted.

So, for every pair in the `freq_pairs_count` directory, it was checked if each item constituting said pair was also part of the the same basket. If so, then the count for that pair was incremented by `1`.

Lastly, only the *frequent* pairs, those that reached the support threshold, are printed to the terminal.

Results

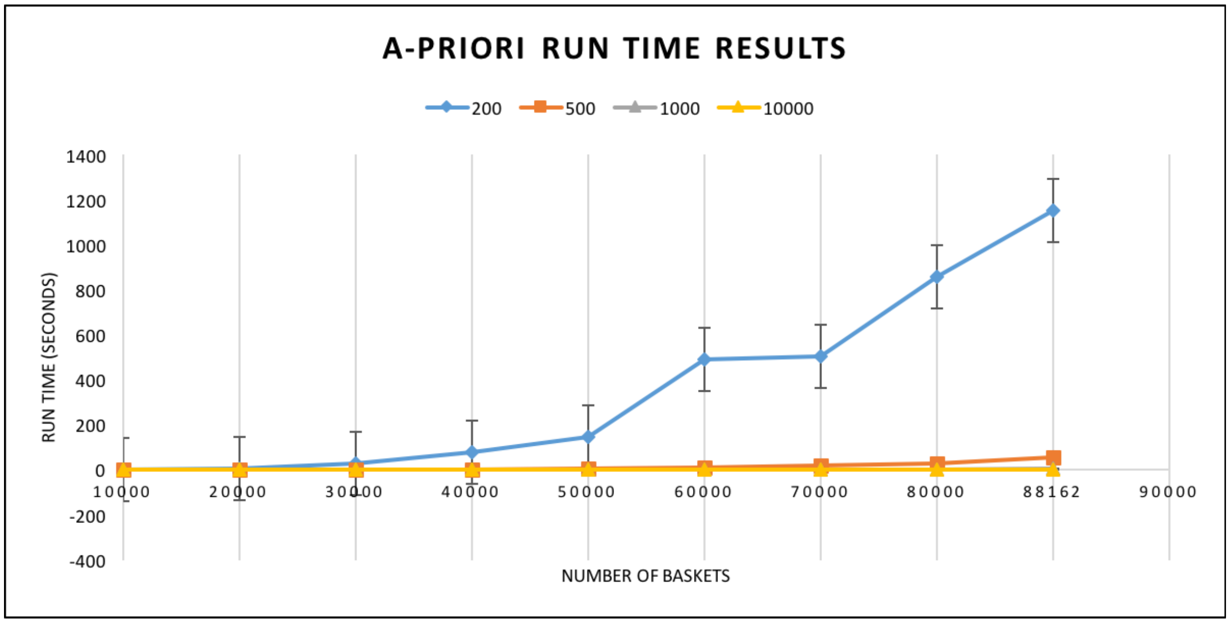
In order to compare both methods with the different values used, an output file containting the support threshold, bucket sample size and run time was created for each.

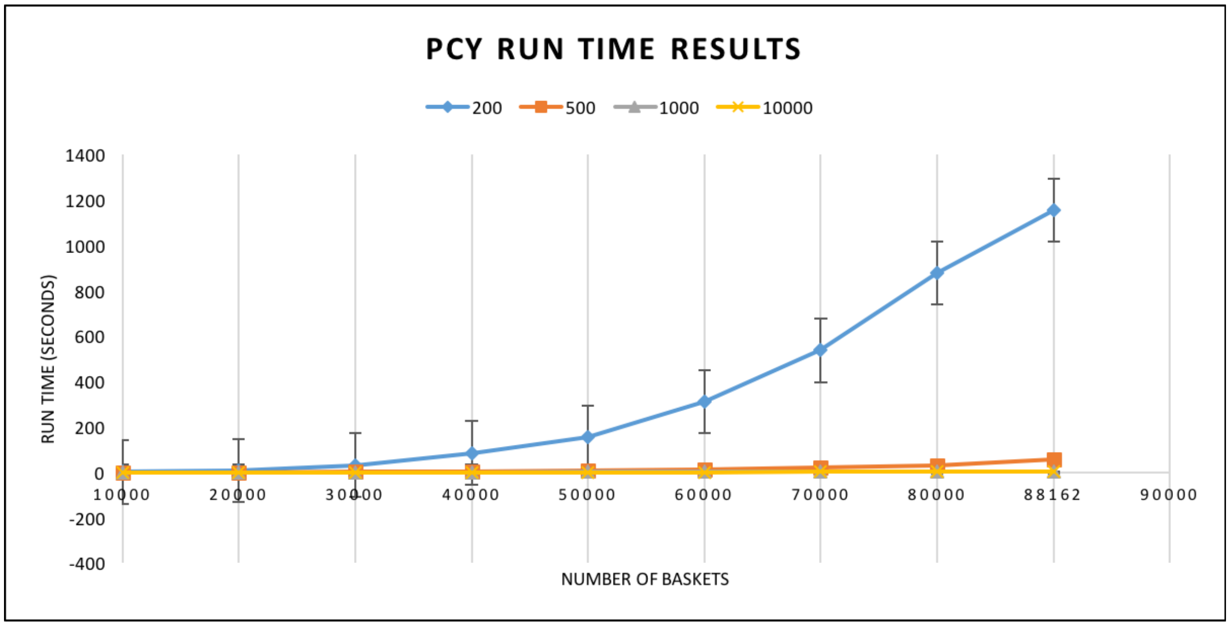
In order to calculate the runtime for each algorithm, the `timeit` module was used, and implemented in the code the following way:

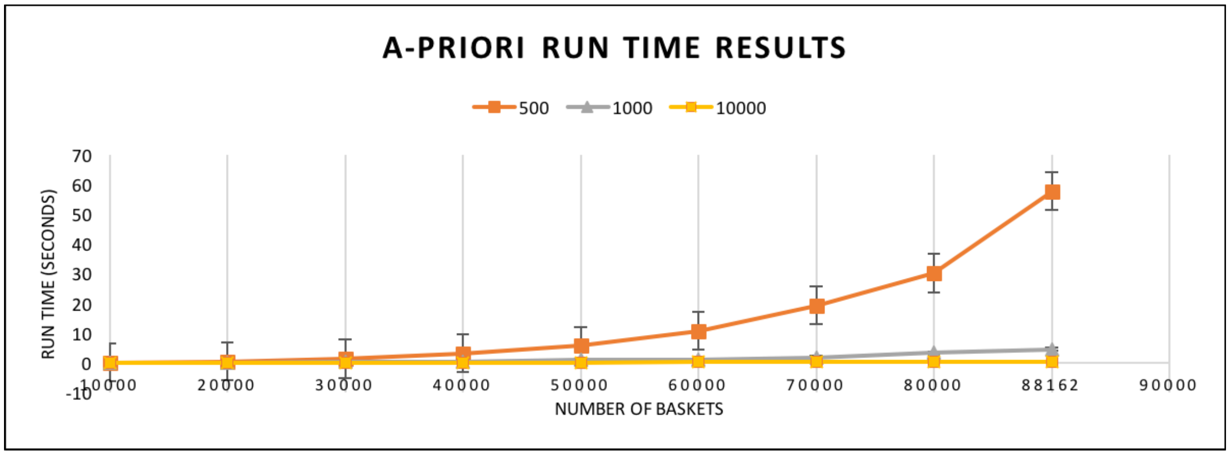
```
import timeit

start = timeit.default_timer()

set_init_vals()
pass1()
between_passes()
pass2()

end = timeit.default_timer()
runtime = end - start
```

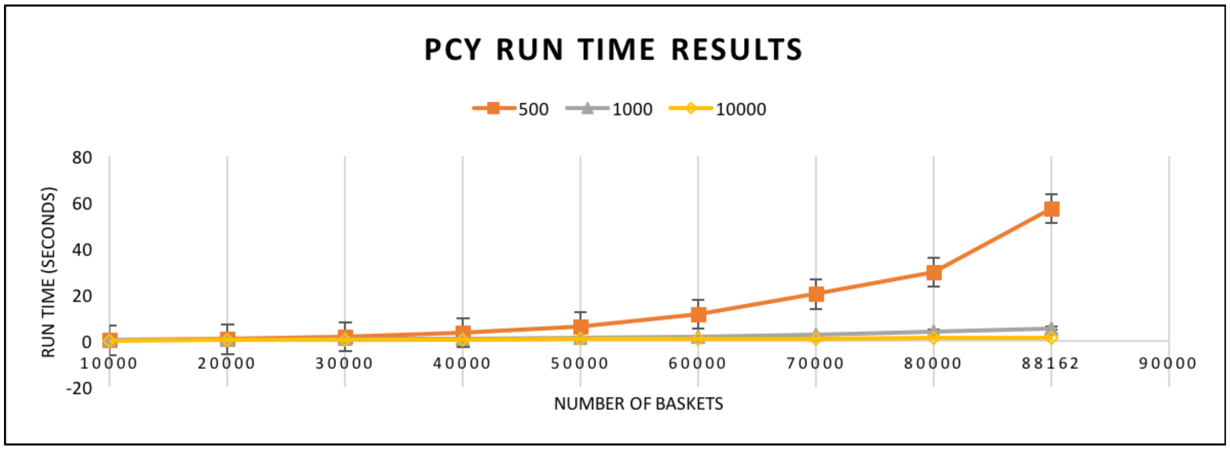
At the end of the code, the program outputs an apriori_results.txt and a pcy_results.txt, respectively, which includes the baskets to be read in, the support threshold and the run time - in this order.

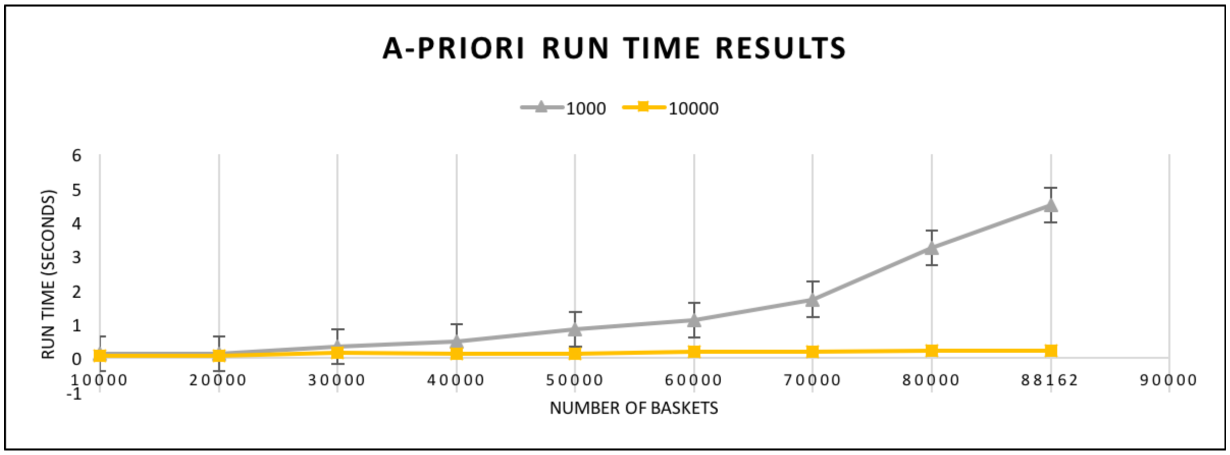
With the data collected, the following line graphs below were generated. I have split each of them into three for a better visualization of the values being graphed.

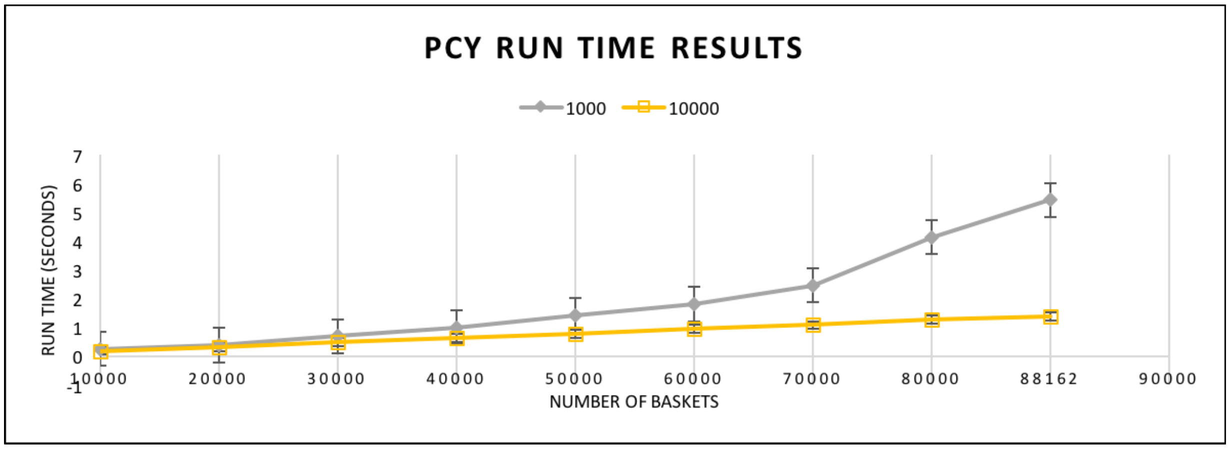





A-PRIORI RUN TIME RESULTS
500
1000
10000
RUN TIME (SECONDS)
70
60
50
40
30
20
10
0
-10
10000
20000
30000
40000
50000
60000
70000
80000
88162
90000
NUMBER OF BASKETS


PCY RUN TIME RESULTS
500
1000
10000
RUN TIME (SECONDS)
80
60
40
20
0
-20
10000
20000
30000
40000
50000
60000
70000
80000
88162
90000
NUMBER OF BASKETS


A-PRIORI RUN TIME RESULTS
1000
10000
RUN TIME (SECONDS)
6
5
4
3
2
1
0
-1
10000
20000
30000
40000
50000
60000
70000
80000
88162
90000
NUMBER OF BASKETS



Little differences regarding the run time for each algorithm can be appreciated in the graphs shown; however, it is apparent that the a-priori algorithm seems to be taking slightly less time than PCY.

It's important to note that the hashing stage of the PCY algorithm must eliminate approximately 2/3 of the candidate pairs for PCY to run faster than A-priori on a given data set.

Nonetheless, PCY is still very much considered an improvement over A-priori because of it's efficient use of memory and its flexibility when refining it, such as

Sources

Built-in Functions. (n.d.). Retrieved March 07, 2016, from https://docs.python.org/2/library/functions.html#

Szlichta, J. (2016) *Frequent Itemset Mining & Association Rules: Big Data Analytics CSCI 4030* [PDF document]. Retrieved from http://data.science.uoit.ca/wp-content/uploads/2016/01/BDA_06-assocrules.pdf

What is PyPy? (n.d.). Retrieved March 07, 2016, from http://pypy.org/features.html