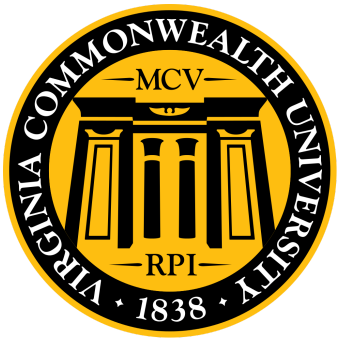
Understanding Mortality and Aging

Utilizing data-mining techniques to find conserved patterns in the different global causes of mortality



Center for the Study of BIOLOGICAL COMPLEXITY

Skyler A. Kuhn

https://github.com/skchronicles/Biodemography

Dr. Tarynn Witten, Dr. Alberto Cano, Dr. Allison Johnson

Center for the Study of Biological Complexity & Department of Computer Science

Virginia Commonwealth University

August 12th, 2017

Table of Contents

I. What is Biodemography?

Humans are mortal. We are susceptible to disease, and we all (eventually) die. No one can escape the process of human senescence-- also known as biological aging. The pre-Socratic Greek philosopher Heraclitus poignantly expressed his thoughts on this reality: "No man ever steps in the same river twice, for it's not the same river and he's not the same man."[7] And although we as a species are not immortal, we have managed to extend the brief time that we do have on Earth with advances in science and technology.

The scientific progress has pushed the boundary of human knowledge in ways that Heraclitus could have never imagined. Interestingly enough, this knowledge could have saved his life. Accounts of Heraclitus' death vary, but a many support this notion. On one fateful day Heraclitus' woke up with severe dropsy (an archaic term for edema)[7, 10]. He consulted a few physicians, but they were unable to prescribe a cure. Fed up with this plight Heraclitus decided to take matters into his own hands. In one account, it states that the philosopher "buried himself in a cowshed, expecting that the noxious damp humour would be drawn out of him by the warmth of the manure"[10]. Another account says he "treated himself with a remedy made of cow manure and, after a day prone in the sun, died."[10] Although Heraclitus was a man of Philosophy, a field of study focusing on the fundamentals of human knowledge and logic, it is almost certain that his tragic death was preventable.

Is there anything that we can learn from Heraclitus' death? Since his passing over two millennia ago new fields of science have emerged. One particular field, called Biodemography, seeks to understand how the biological determinates of death and birth vary across populations[9]. Biodemographers are the middle ground to understanding biological complexity at the cross section of science and demography. Unlocking the door to understanding, biodemographers generate useful information for researchers interested in gerontology and other fields within the biological sciences.

The multitudinous array representing the biological determines death is staggeringly large. There can be thousands if not millions of determinants related to mortality within a large population. Therefore, computational methods are needed to gain any new insights. Given a relatively modest set of biological determinants leading to death (say fifty), an exhaustive

combinatorics search to find all possible combinations would take decades to execute. While a brute force search approach is theoretically possible and simple to implement, it becomes impractical to implement as the size of the set increases[6].

This paper presents a pipeline that implements an unique method for utilizing frequent item-set data mining techniques to find conserved patterns in the different global causes of mortality[A]. An adapted version of the apriori algorithm is implemented to find the common causes mortality across the globe in a set of thirty-five icd codes. The theory being that once all of the conserved possible combinations leading to morbidity across the globe are determined, one may tentatively identify whether the underlying etiologies of these diseases or groups of diseases are genetic or environmental.

II. Exhaustive Search Methods, Metaheuristics, Oooh my

Exhaustive searches, also known as brute force searches, are class of iterative methods typically used to solve discrete problems where no efficient solution method is known[6]. This method enumerates through all possible combinations of a given set to test each candidate (or each generated combination) against a predefined problem statement. In the past, brute force searches have been implemented to bypass user authentication (passwords) and encryption methods where there are no efficient methods for finding a solution[3, 4].

Due to the very nature of exhaustive searching, its implementation may not always be a viable option. Although many exhaustive search algorithms are easy to implement and program, they suffer from severe performance issues. As the size of the set to be permuted grows, the time complexity increases in a polynomial manner-- i.e. O(n^k) where n is the cardinal number of the set and k is the cardinal number of the candidate itemset[6, 8]. That being said, these algorithms are generally used as a last resort where no other alternatives exist.

Metaheuristics are an alternative to exhaustive search methods that can be employed to find a sufficient solution in a set which is too large to be completely sampled[5]. A metaheuristic algorithm will use a heuristic to cut off subsets of the search, effectively reducing its search space and runtime[14]. Metaheuristics, unlike exhaustive search methods, do not always yield a global optimal solution (a solution to all possible combinations)[5, 8]. But when

compared to iterative methods like exhaustive searching, metaheuristics can often find good solutions with less computational resources and time; and because of this, they are often used to find the optimal solution from a large finite set[6, 14].

A classic example where exhaustive methods fail is the traveling salesman problem. This problem asks to find the shortest possible path between a list of cities given the distances between each city pair[12]. The path the salesman follows may only visit each city once, and they must return to their original starting point. In this scenario using an exhaustive search to find the shortest route is infeasible in many cases as the number of possible solutions grows faster than an exponential series with each added city[12].

The field of Computational Complexity Theory identifies the inherent difficulty that some problems yield and classifies them accordingly[8]. One of the biggest problems in theoretical computer science is whether P = NP. If true, it would mean that all very hard problems have seemingly easy solutions. The implications of this proof would be drastic and widespread, because many important problems would have more efficient solutions[8]. The effects of this discovery would even permeate to the field of bioinformatics-- hinting at the possibility of much more efficient methods for protein structure prediction[2].

III. Apriori Algorithm

Apriori Algorithm is a robust metaheuristic used in data mining for the evaluation of frequent itemsets for boolean association rules. Instead of exhaustively evaluating each generated itemset, the apriori algorithm evaluates itemsets which have met a minimum support (a count of how frequently the itemset appears within the dataset or database)[1]. Candidate itemsets grow in a bottom-up manner where a candidate k-itemsets are joined together. This searching method is based off of the apriori property which states that any subset of a frequent itemset must also be frequent[1].

As so given the set S {1,2,3,4}, each of the 1-itemsets {1}, {2}, {3}, and {4} will be evaluated to see if they meet the minimum support. Let's say that {1}, {2}, and {3} meet this heuristic. Next, candidate sets with cardinality of 2 will be formed. The candidate C_k {1,2} is generated by joining {1} and {2}. This new 2-itemset is then evaluated to see if it meets the

minimum support. Next {1,3} and {2,3} will be evaluated. If {1,2} is found to be infrequent (does not meet the minimum support), then the candidate {1,2,3} will never be evaluated. Again, this is because the 3-itemset, {1,2,3}, is the result of a join between {1,2} and {3}. The {1,2,3} itemset does not meet the apriori property so it is never evaluated; its subsets are not frequent. The use of this metaheuristic effectively reduces the search space in an otherwise exhaustive search.

The apriori property demonstrates that the minimum support for the candidate itemsets must be greater or equal to those of its constituent subsets[1]. If not, the search space can be pruned. This is due to anti-monotone constraints, which state if a given constraint is violated its further mining can be stopped[1, 11]. This pruning constraint shows if an itemset violates a given constraint, all of its supersets will too. That being said, this property can be employed to prune the search space.

This pipeline developed for this project uses a p-value calculated from a one-way chi-squared test as its heuristic. If a candidate itemset does not meet this threshold then its search space is effectively pruned. Given a set of 35 icd codes and their corresponding mortality rates, runtime is reduced from several years to a few minutes.

IV. Getting Started

One of the project's main goals was to develop software that anyone with a basic inclination towards programming could use. That being said, the pipeline needed to meet the following criteria: easy invocation and implementation, open-source, platform independent, and quick runtime. The pipeline was developed in Python-- a programming language that holds dear to my heart. Although the pipeline was developed in Python 2.7, it is also compatible with its more recent releases (≥ 3.0). The project's home page, which includes all of its source code and datafiles, can be found on github:

https://github.com/skchronicles/Biodemography

A setup.py file was created to generate a source distribution, which includes all the data and metadata needed for distribution purposes or for creating a build[13, 17]. Once the project is in this format, all the files and metadata can be appropriately allocated to create a build

distribution[17]. This is python's way of achieving platform independent installation. After the package is downloaded, the following command can be run to install the build:

```
python setup.py build
python setup.py install
```

These commands are analogous to "make" and "make install" on unix-like operating systems. The former command will use "easy_install", a built-in package manager, to download any dependency files or packages specified within setup.py[13]. The latter command, "python setup.py install", will move all of the application files into the correct directory-- i.e. /usr/bin/-- so they are in the operating system's path.

Pipeline usage can be seen below:

```
python pipeline.py
```

Execution is that easy. The only requirement from the user is that the datafiles are in the same directory as the program, and that they save in comma separated value (.csv) format. Additionally, two required packages that are not a part of Python's standard library need to be downloaded: scipy and numpy. If the build is downloaded and setup.py is executed in the manner described above, python will handle the download of these dependencies.

Scipy is an open-source library used within the scientific community for aiding mathematics, engineering, and statistical analysis. It includes a module called "stats" that contains a chi-square analysis function. This function is used to generate p-values that are used against a heuristic in the apriori algorithm. If problems arise during the build's setup, python's package manager ("pip") can be used to download scipy and numpy. It should be noted that if you are using Python $\geq$ 2.7.9, pip is included with python's standard library[13]. If not, pip can be downloaded using the two commands below in most unix-like operating systems.

```
sudo apt-get install build-essential gfortran libatlas-base-dev python-pip python-dev
sudo pip install --upgrade pip
```

Once downloaded numpy and scipy can be downloaded using the two following commands:

```
sudo pip install numpy
sudo pip install scipy
```

If you are using a windows computer, and do not have pip installed. Please download "get-pip.py" from this secure location:

```
https://pip.pypa.io/en/stable/installing/#do-i-need-to-install-pip
```

Once downloaded, please make sure the file is saved with the ".py" file extension. After the download is complete, open cmd in administrator mode, and go to the directory or folder you stored it in (most likely the Downloads folder). Then run the following command:

```
python get-pip.py
```

Again, the setup.py file with this project's build should handle all of these steps. If not, please flag any issues on the project's github page[17].

V. Pipeline

The pipeline can be divided into three major constituents: data preprocessing, data management, and data analysis[A]. An old proverb states that a chain is only as strong as its weakest link, and as so each constituent in the flow of data processing and manipulation is only as good as its weakest link. That being said, throughout the development process extensive testing has been conducted to ensure the accuracy and efficiency.

Before any of the data can be analyzed, it must be pre-processed. This step is instantiated by "preprocess.py". This program does a few things. The first being cleaning up the datasets. Before any data analysis begins, the function "pre_process_data" will analyze all columns containing icd code mortality rates, and it will replace null fields with "0"[17]. This is done to prevent any problems when the chi-squared analysis begins. The program's second major role is to parse each of the datasets (which contain biodemographic data on the top icd mortality rates for a range of ages for a given country) by biological sex into two new files [H]. So for each file, two new files will be created for males and females. This program is instantiated if the number of country files to biological-sex-country-files is not a 1:2 ratio.

The project's second program entitled "pipeline.py" is responsible for data management and data analysis[17]. Within this program lies the a function implementing the apriori algorithm and dozen other supplemental functions. These other functions perform various tasks from converting files to a nested dictionaries to finding the support counts.

For the management and housing of all the project's data, two dictionaries were created for each biological sex to contain all the data within the twelve datasets[E]. This data structure was chosen for its exceptional time complexity. Dictionaries, also know as hash tables, provide the benefit quick lookup times, insertions and deletions. Once these dictionaries were created, each age was evaluated to see if it met the minimum support (which in this case was the number countries or 6)[G]. If an age did not meet the minimum support, then one or more of the data set(s) did not contain any information on that age group, and that age was not evaluated. The goal is to find diseases or groups of diseases conserved across all six countries.

The last component of the pipeline implementation of the apriori algorithm. The function called "apriori_v3" implements a modified version of this metaheuristic for frequent itemset mining[F]. The adapted version contains a heuristic that differs from calculating minimum support counts. A p-value calculated from a one-way chi-squared test acts as this project's heuristic[A]. If a candidate itemset does not meet the threshold for its p-value then its search space is effectively prune. To implement this algorithm, a queue and a list of insignificant itemsets were used generate tentative candidates[F]. If an itemset was found to be significant, it was added to the queue and appended to a list of significant itemsets[F]. If found to be insignificant, the tentative candidate was popped from the queue[F]. This effectively and efficiently reduces the algorithm's search space.

VI. Benchmarking Results

To determine the generation times of k-itemsets of varying cardinality, benchmarking was performed. All benchmarking was conducted on Compile-- a 16-core server with 32 gigabytes of RAM located in Center for High Performance Computing at Virginia Commonwealth University. Timing of an exhaustive search for the the set S {1,2,3, ..., 35} verified what is termed as combinatorial explosion[C]. As the cardinality of the itemset to be permuted increases, its time complexity increases in a polynomial manner[C]. Essentially, the number of combinations grows faster than the fastest computer can evaluate in a tolerable amount of time[C D]. With the observed timing information, generation times of itemsets outside of tested range were extrapolated{D}.

VII. Limitations

Metaheuristics, such as the apriori algorithm, often do not yield the global optimum solution. This algorithm may not find the all solutions an exhaustive search would find, but it will yield the optimal solution within a set of neighboring candidate solutions. As so it should be stated that the apriori algorithm's heuristic (p-value threshold) can be tweaked in such a way as to approach or meet the global optimum solution. That being said, requiring a less strenuous heuristic will yield more results, and a p-value threshold of 0.0 will mimic an exhaustive search.

VIII. Acknowledgements

Dear Supporters and Committee Members,

It is of genuine pleasure to express my deepest sympathies to you all today. None of this would have been possible without you; thank you for your timely feedback, contagious kindness, and boundless knowledge that has enabled myself to accomplish this tumultuous task. And for that, I say thank you!

I would personally like to thank Dr. Tarynn Witten for all of her support and wisdom throughout this journey, Dr. Alberto Cano for his remarkedly in depth knowledge of data-mining techniques, and Dr. Bruce Carnes for initiating this project by providing the problem statement.

Thank you all for supporting the power of mentorship. I believe it has the power to transform the world into a better place. The passing of knowledge is one of humanity's most sacred acts of veneration. I am humbled and honored to have worked with each one of you.

Sincerely,

Skyler Kuhn

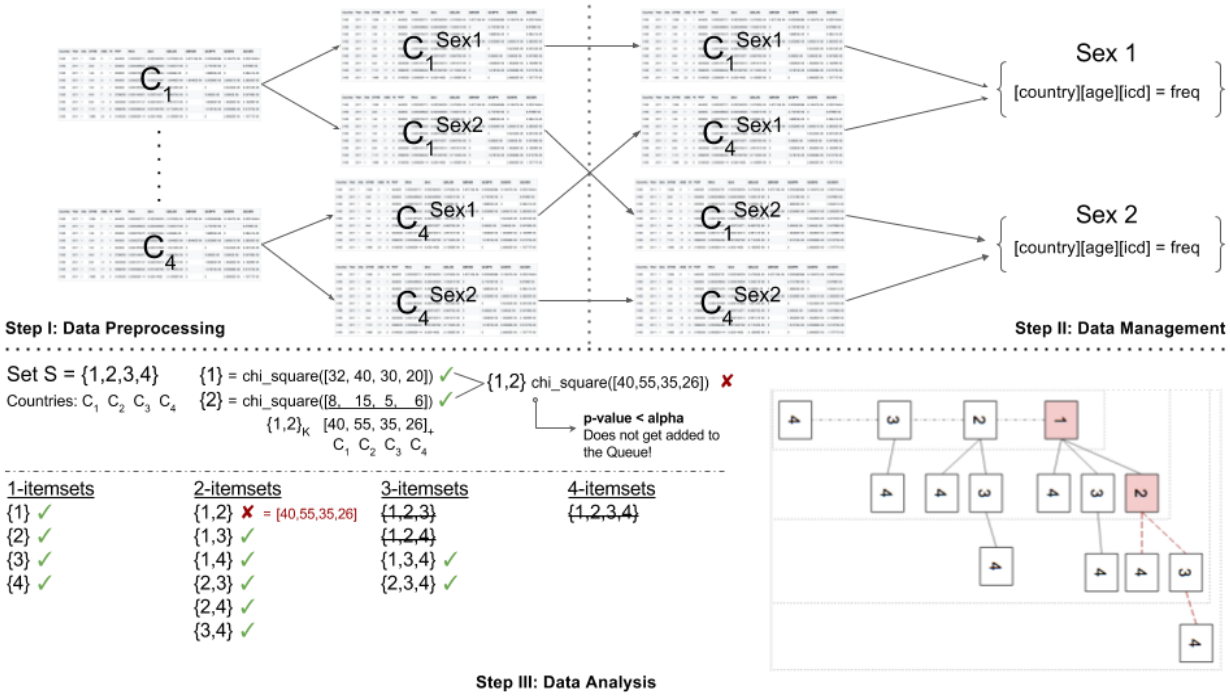
IX. References

1. Adamo, Jean-Marc. "Apriori and Other Algorithms." *Data Mining for Association Rules and Sequential Patterns* (2001): 33-48. Print.
2. Berger, Bonnie, and Tom Leighton. "Protein Folding in the Hydrophobic-Hydrophilic (HP) Model Is NP-Complete." *Journal of Computational Biology* 5.1 (1998): 27-40. Print.
3. Blaze, Matt. *Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security. A Report by an Ad Hoc Group of Cryptographers and Computer Scientists*. United States: Information Assurance Technology Analysis Center Falls Church Va, 1996. Print.
4. "Blocking Brute Force Attacks." *Blocking Brute Force Attacks - System Administration Database*. Web. 07 Aug. 2017.
5. Blum, Christian, and Andrea Roli. "Metaheuristics in Combinatorial Optimization." *ACM Computing Surveys* 35.3 (2003): 268-308. Print.
6. "Brute-force Search." *Wikipedia*. Wikimedia Foundation, 04 Aug. 2017. Web. 07 Aug. 2017.
7. Chitwood, Ava. *Death by Philosophy: The Biographical Tradition in the Life and Death of the Archaic Philosophers Empedocles, Heraclitus, and Democritus*. Ann Arbor, MI: U of Michigan, 2004. Print.
8. "Computational Complexity Theory." *Wikipedia*. Wikimedia Foundation, 06 Aug. 2017. Web. 07 Aug. 2017.
9. Crimmins, Eileen, Jung Ki Kim, and Sarinnapha Vasunilashorn. "Biodemography: New Approaches to Understanding Trends and Differences in Population Health and Mortality." *Demography* 47.S (2010). Print.
10. Fairweather, Janet. "Death of Heraclitus". Bedford, UK: Bedford College London. Print.
11. Leung, Carson Kai-Sang. "Anti-monotone Constraints." *Springer*. Springer US, 01 Jan. 1970. Web. 07 Aug. 2017.
12. "Travelling Salesman Problem." *Wikipedia*. Wikimedia Foundation, 06 Aug. 2017. Web. 07 Aug. 2017.

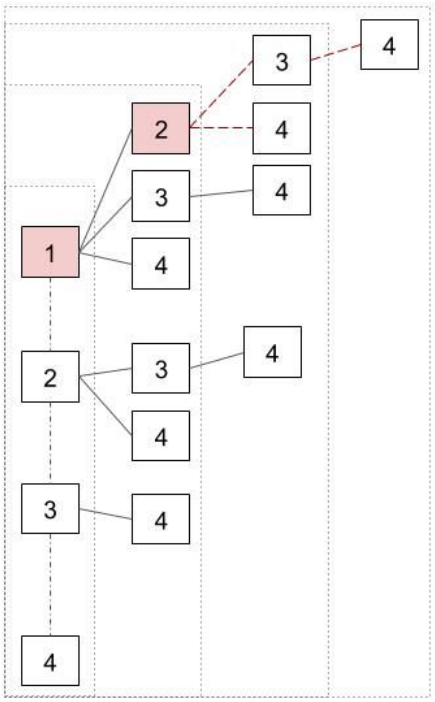
13. "Welcome to Python.org." *Python.org*. Web. 07 Aug. 2017.
14. Yang, Xin-She. "Metaheuristic Optimization." *Scholarpedia* 6.8 (2011): 11472. Print.
15. Carnes, B.A., Holden, L.R., Olshansky, S.J., Witten, T.M., and Siegel, J.S. (2006). Mortality partitions and their relevance to research on aging. Biogerontology, 7: 183-198. DOI 10.1007/s10522-006-9020-3, URL http://dx.doi.org/10.1007/s10522-006-9020-3
16. Carnes, B.A. & Witten, T.M. (2013). How long must humans live? Journal of Gerontology, Biol. Sci. Med. Sci., DOI: 10.1093/Gerona/glt164. Print version, J. Gerontol. Biol. Sci. Med., 69A (8): 965-970.
17. Kuhn, Skyler. "https://github.com/skchronicles/Biodemography." *GitHub*. 08 Aug. 2017. Web. 12 Aug. 2017.

X. Appendices

A.



B.



Exhaustive Search of set S {1,2,3,4} yielding k-itemsets

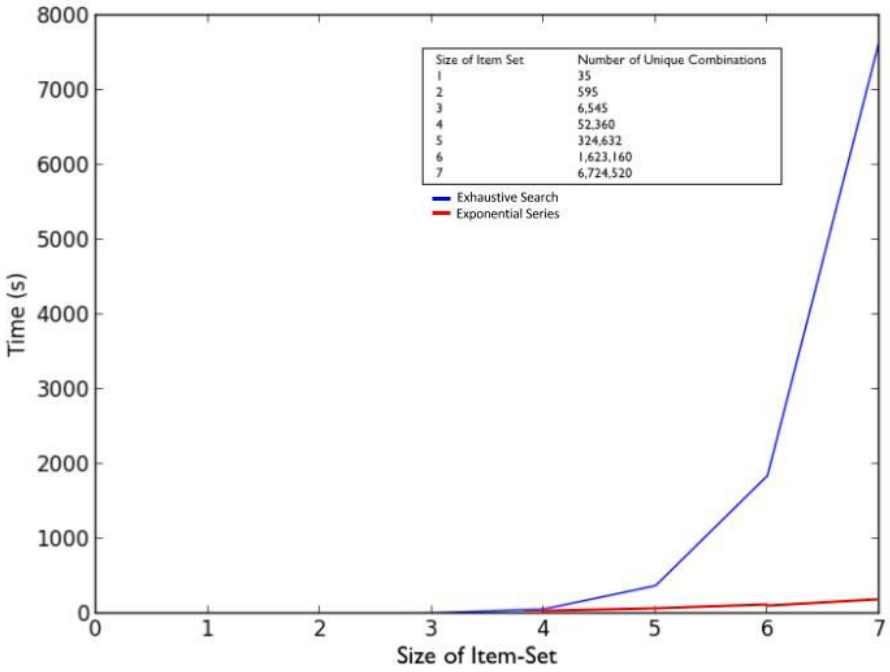
1-itemsets	2-itemsets	3-itemsets	4-itemsets
{1}	{1,2}	{1,2,3}	{1,2,3,4}
{2}	{1,3}	{1,2,4}	
{3}	{1,4}	{1,3,4}	
{4}	{2,3}	{2,3,4}	
	{2,4}		
	{3,4}		

Apriori Algorithm of set S {1,2,3,4} yielding k-itemsets

1-itemsets	2-itemsets	3-itemsets	4-itemsets
{1} ✓	{1,2} ✗	~~{1,2,3}~~	~~{1,2,3,4}~~
{2} ✓	{1,3} ✓	~~{1,2,4}~~	
{3} ✓	{1,4} ✓	{1,3,4} ✓	
{4} ✓	{2,3} ✓	{2,3,4} ✓	
	{2,4} ✓		
	{3,4} ✓		

C.

Combinatorial explosion of k-itemset generation in an exhaustive search of set S {1, 2, 3, …, 35}



D.

Observed

Itemset Cardinality	Number of Unique Combinations	Generative Timing (secs)
0	0	0.0
1	35	0.04
2	595	0.68
3	6,545	7.39
4	52,360	59.46
5	324,632	373.39
6	1,623,160	1845.85
7	6,724,520	7660.54

Extrapolated

Itemset Cardinality	Number of Unique Combinations	Generative Timing (secs)
8	23,535,820	26934.39
10	183,579,396	210088.26
12	834,451,800	954946.64
14	2,319,959,400	2654961.54
16	4,059,928,950	4646182.69
18	4,537,567,650	5192792.42
20	3,247,943,160	3716946.15
22	1,476,337,800	1689520.98
24	417,225,900	477473.32
26	70,607,460	80803.18
28	6,724,520	7695.54
30	324,632	371.51
35	1	0.0011444

E.

```
def files2dictionary(filename, countryID, supp_dict):
    """Creates a dictionary to hold all the information for each biological sex below.
        It has following structure
        {"Sex1_mdlt2450":
          {"0":
               {"1":datavalue, "2":dataValue...},
           "2.5":
               {"1":datavalue, "2":dataValue...}
          }
         }
        -- where: [Sex#countryID][Age][ICDcode] = one_datapoint"""

    fh = open(filename)
    header = next(fh)

    data_dict = {}
    data_dict[countryID] = {}

    numlist = range(1, 36)
    agelist = []
    for line in fh:
        linelist = line.strip().split(",")
        age = linelist[4]
        agelist.append(age)
        for icdrep in numlist:
            if str(age) not in data_dict[countryID]:
                data_dict[countryID][str(age)] = {}
                data_dict[countryID][str(age)][str(icdrep)] = float(linelist[icdrep+8])
            else:
                data_dict[countryID][str(age)][str(icdrep)] = float(linelist[icdrep+8])

    fh.close()
    supp_dict.update(support_counter(header.split(","), agelist, supp_dict))
    return data_dict, supp_dict
```

F.

```
def apriori_v3(q, insig, sex_file_dict, countries_list, age):
    """ This function implements a modified version of the apriori algorithm
        which can be used for speeding up an otherwise exhaustive high-
        performance computing problem. The apriori algorithm is commonly
        used in mining frequent itemsets for boolean association rules.
        It uses anti-monotonic "bottom up" approach, where frequent subsets
        are extended one item at a time."""

    q = [[int(num)] for num in q]  # queue is formatted as a nested list
    insignificant = [[int(num)] for num in insig]
    significant = []

    #print("\nInsig", insignificant)
    #print("Sig", significant)
    #print("Queue\n", q)

    while len(q) > 0:
        element = q[0]
        obs_freqs = []

        for country in countries_list:
            icd_freq = 0
            for freq in element:
                icd_freq += round(float(sex_file_dict[country][age][str(freq)]) * 1000000)
            obs_freqs.append(icd_freq)
        if pvalue >= 0.05:
            significant.append(element)

            for i in range(int(element[-1])+1,36):
                if [i] not in insignificant:
                    tentativeCandidate = sorted(list(element)+[i])
                    # add the two lists together (element is a list)
                    if tentativeCandidate not in q and tentativeCandidate not in significant:
                        q.append(tentativeCandidate)   # then add it to the queue
            q.pop(0) #remove it from the queue after we have created all the tentativeCandidates

        else:        # when the p-value not significant
            q.pop(0)
            insignificant.append(element)
    return significant  # grab the last values before breaking out of the while loop
```

G.

```
def support_counter(linelist, agelist, supp_dict):
    support_count_dict = supp_dict
    try:
        start_index = linelist.index("Qtot") + 1
    except ValueError:
        start_index = 9

    countslist = linelist[start_index:start_index + 35] + agelist
    for item in countslist:
        if item in support_count_dict:
            support_count_dict[item] += 1
        else:
            support_count_dict[item] = 1
    return support_count_dict
```

H.

```
def parse_file(filename):
    """This function takes a filename as a argument and creates
    new files according for each biological sex."""

    print(filename)
    fh = open(filename)
    header = next(fh)
    output_file = open("Sex" + "1_" + filename, "a")
    output_file.write(header)
    output_file2 = open("Sex" + "2_" + filename, "a")
    output_file2.write(header)

    for line in fh:
        line = line.strip().split(",")  # split by "," because it is a csv file
        line = pre_process_data(line)
        if int(line[2]) == 1:
            output_file.write(",".join(line) + "\n")
        else:
            output_file2.write(",".join(line) + "\n")
    output_file.close()
    output_file2.close()
    fh.close()
```