## DMML:

## Team:

- Aritra Banerjee (MDS-2018-28)
- Trina De (MDS-2018-24)

## Packages Used:

- Efficient-apriori
- Time
- Warnings
- Pandas

## Documentation:

- Efficient-apriori :

  *efficient_apriori.apriori(transactions: List[tuple], min_support: float = 0.5, min_confidence:float = 0.5, max_length: int = 8, verbosity: int = 0)*

  It has functions like:

➔ "Apriori": - The Apriori algorithm works in two phases. Phase 1 iterates over the transactions several times to build up itemsets of the desired support level. Phase 2 builds association rules of the desired confidence given the itemsets found in Phase 1. Both of these phases may be correctly implemented by exhausting the search space, i.e.generating every possible itemset and checking it's support. The Apriori prunes the search space efficiently by deciding apriori if an itemset possibly has the desired support, before iterating over the entire dataset andchecking.

➔ "Join_step": Join k length itemsets into k + 1 length itemsets. This algorithm assumes that the list of itemsets are sorted, and that the itemsets themselves are sorted tuples. Instead of always enumerating all n^2 combinations, the algorithm only has n^2 runtime for each block of itemsets with the first k - 1 items equal.

➔ "Prune_step": Prune possible itemsets whose subsets are not in the list of itemsets.

➔ "Apriori_gen": Compute all possible k + 1 length supersets from k length itemsets. This is done efficiently by using the downward-closure property of the support function, which states that if support(S) > k, then support(s) > k for every subset s of S.

- ➔ “Itemsets_from_transactions”: Compute itemsets from transactions by building the itemsets bottom up and iterating over the transactions to compute the support repedately. This is the heart of the Apriori algorithm by Agrawal et al. in the 1994 paper.

- Time : time.clock() function in the package “time” gives us two timestamps for start and end times which when subtracted gives us the time spent on running the code.

- Warnings : Warning messages are used to alert the user of some condition in a program, where that condition (normally) doesn’t warrant raising an exception and terminating the program.

- Pandas : Pandas is an open source library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

## Code Description:

This code is written typically for the three datasets, enron, kos and nips in the UCI machine learning library. It may or may not run for files out of the UCI or in a different format.

The code is written in certain parts:

- Our function name is frequent_words. It takes 3 things as input. First is **file name**, the **minimum support** and the **length of the frequent itemsets we wish to obtain** .

  For example,
  frequent_words( “kos”, 0.7, 4)

  Where 0.7 is the minimum support and 4 is the length of the frequent itemsets that we want

- The first part of the code is loading the packages. Whatever packages are mentioned above, if not installed will be asked to be installed.

- The data is loaded from its location on the cloud and only the name in its appropriate format is required. If the computer is not connected to the internet, it will ask you to connect because otherwise the data cannot be imported.

- Starting the clock after importing the data since importing the data depends on the speed of internet.

- The next step is converting the data into a format so that it can be read by the function apriori to get the frequent itemsets. To do this we put the **"vocab"** and **"docword"** files in a pandas dataframe. **"vocab"** contains the distinct words that occur in the corpus with line number as **"WordID".** The data frame is composed by omitting the first 3 rows of **"docword"** which contains the values of D, W and NNZ. Still it is read as a single column, the 3 value tuples which correspond to **DocID**, **WordID** and count. We use the inbuilt function **str.split()** to split it up into 3 columns and store it in **"data_NNZ"**.

- The file **"data_NNZ"** we convert into a dictionary where the keys are the **DocID**s and the values are the **WordID**s of the words in the document. The name of this dictionary is **"dict"**. We ignore the count values as it doesn't matter wrt to our program. The set of values of **"dict"** are stored in **" words_in_doc"** which is the input file for the funcion **apriori**.

- Next step is running the **apriori function** where **our parameters are the list of transactions ; "words_in_doc"** and the **minimum support** which is provided by the user.

- The last step is filtering out the output of apriori to get what is required. The function **apriori** gives frequent itemsets of all size that satisfy the condition of minimum support but we are only interested in itemsets of a particular size provided by the user as k. The output of **apriori** is a dictionary stored in **"itemsets"** where keys are the varying lengths of frequent itemsets. The values are again each a dictionary where keys are the frequent itemsets and the values is the frequency with which it occurs in the documents. The output we get have the WordIDs and not the actual words. A short code maps the **WordID**s to the words for us.

- We will extract only those values out of the output where the key value is equal to k stored in **"frequent_word_id"**. We will map **"frequent_word_id"** to **"vocab"** to get the corresponding words and storing it in **"frequent_word"**.

- Stopping the clock and reporting the output. Output is all k-length itemsets with support more than min_support n. The time, minimum support provided by the user and number of such k-length itemsets called.