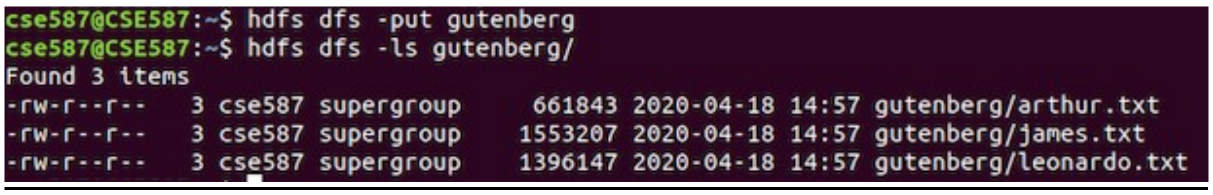
BIG DATA PROCESSING WITH HADOOP

Part1 : Wordcount on text documents
Part2 : Getting top 10 most occurring N-grams from text documents
Part3 : Inverted Index on text documents
Part4 : Relational Join on two excel sheets which replicate two RDBMS tables.
Part5 : Implementing K-Nearest Neighbours using Hadoop

PART 1: Setup and Wordcount

Gutenberg folder has been copied to Hadoop distributed file system.



```
cse587@CSE587:~$ hdfs dfs -put gutenberg
cse587@CSE587:~$ hdfs dfs -ls gutenberg/
Found 3 items
-rw-r--r--   3 cse587 supergroup     661843 2020-04-18 14:57 gutenberg/arthur.txt
-rw-r--r--   3 cse587 supergroup    1553207 2020-04-18 14:57 gutenberg/james.txt
-rw-r--r--   3 cse587 supergroup    1396147 2020-04-18 14:57 gutenberg/leonardo.txt
```

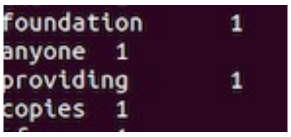
For this part, mapper and reducer algorithms are coded in such a way that mapper picks all the words and assigns 1 to each word and reducer gives the word with number of times they occurred.

Number of mappers and reducers are defined in the command which is given to run the mapreduce algorithm to get wordcount.
We have checked output with and without defining number of mappers and reducers just to be sure whether the code is compatible to multi node cluster, we found that the outputs of both cases same.

Command used:
hadoop jar /home/cse587/hadoop-3.1.2/share/hadoop/tools/lib/hadoop-streaming-3.1.2.jar -D mapred.map.tasks=6 -D mapred.reduce.tasks=6 -file /home/cse587/mapperp1.py -mapper "python3 /home/cse587/mapper_wordcount.py" -file /home/cse587/reducerp1.py -reducer "python3 /home/cse587/reducer_wordcount.py" -input gutenberg/* -output wordcount

Mapper Algorithm:
Libraries used: sys, nltk
All .txt files from Gutenberg folder will be considered as input for this mapper.
Mapper strips each line from the each of input .txt file and each line is split based on space to get each word of that line.
All words across all the files are pooled into a global list.
For each word, special characters and numbers are excluded from the word (only alphabets of a word are concated as a word), word is then changed to lower case, empty strings are also ruled out (not considered for stemming) and then the word is stemmed with the help of WordNetLemmatizer (.lemmatize function) and stemmed version of the word is stored into the list.
Stop words are not removed from the documents.
A ‘for’ loop is then used to iterate over the entire list, each word and its count (1 in this case as this mapper algorithm doesn’t combine them locally) is printed in tab delimited (\t) format line by line (\n).

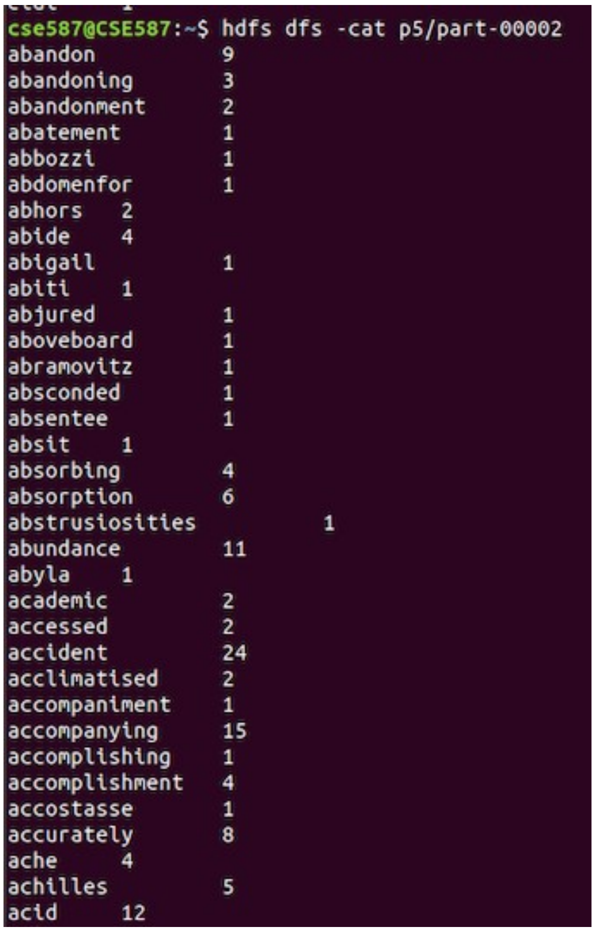
Similar to below snip:



```
foundation      1
anyone  1
providing       1
copies  1
```

This printed output is then received by the Reducer as input.

Reducer Algorithm: Libraries
used: sys
The tab delimited output is read as input for reducer. Each line from the input is stripped and split on tab (\t). Empty lines are ignored from the input. Now every line is a list with word and its count from mapper. All these lists are stored in a global list.
An empty list and dictionary are initialized to store unique words and word with its count respectively.
Global list is iterated, each word is checked in the unique word list. If the word is not present in unique word list, dictionary is updated with the word as key and its count as its value and the word is added to unique word list. But if the word is present in the unique word list, value of this word from the global list is added to the value corresponding to the word as key in the dictionary.
Dictionary is iterated, key and its value (word and its count) are printed in tab delimited format (\t) line by line (\n).



```
cse587@CSE587:~$ hdfs dfs -cat p5/part-00002
abandon         9
abandoning      3
abandonment     2
abatement       1
abbozzi         1
abdomenfor      1
abhors  2
abide   4
abigail         1
abiti   1
abjured         1
aboveboard      1
abramovitz      1
absconded       1
absentee        1
absit   1
absorbing       4
absorption      6
abstrusiosities         1
abundance       11
abyla   1
academic        2
accessed        2
accident        24
acclimatised    2
accompaniment   1
accompanying    15
accomplishing   1
accomplishment  4
accostasse      1
accurately      8
ache    4
achilles        5
acid    12
```

PART 2: N grams

For this part in layer1, mapper and reducer algorithms are coded in such a way that mapper picks all trigrams and assigns 1 to each word and reducer gives top 10 the trigrams with number of times they occurred in each reducer.
In layer2, mapper picks trigrams and counts from all reducers of layer1 and reducer picks top10 trigrams based on number of times they occurred.

Number of mappers and reducers are defined in the command which is given to run the mapreduce algorithm in layer 1 to get trigrams. And for second layer of mapreduce only one reducer is used to pick top trigrams.

Command used:
```
hadoop jar /home/cse587/hadoop-3.1.2/share/hadoop/tools/lib/hadoop-streaming-3.1.2.jar -D mapred.map.tasks=4 -D mapred.reduce.tasks=4 -file /home/cse587/mapperp2a.py -mapper "python3 /home/cse587/mapper_ngrams1.py" -file /home/cse587/reducerp2a.py -reducer "python3 /home/cse587/reducer_ngrams1.py" -input gutenberg/*  -output ngramsoutput1
```
Layer1:

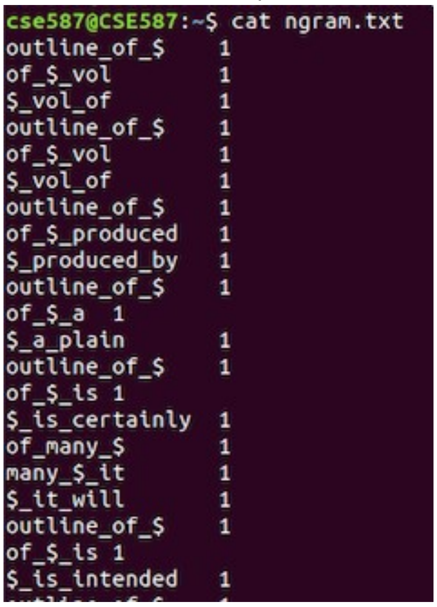
Mapper Algorithm:
Libraries used: sys, nltk
All .txt files from Gutenberg folder will be considered as input for this mapper.
Mapper strips each line from the each of input .txt file and each line is split based on space to get each word of that line.
All words across all the files are pooled into a list.
For each word, special characters and numbers are excluded from the word (only alphabets of a word are concated as a word), word is then changed to lower case, empty strings are also ruled out (not considered for stemming) and then the word is stemmed with the help of WordNetLemmatizer (.lemmatize function) and stemmed version of the word is stored into a global list.
Even in this algorithm stop words are not removed from the documents.

Keywords mentioned in the question ('science', 'fire', 'sea') are stemmed with the help of WordNetLemmatizer (.lemmatize function) and stemmed keyword is stored in a keyword list.
Global list is then iterated, each stemmed word from the list is compared with each of the stemmed keyword. If the stemmed version of the word from the global list matches with any of the stemmed keyword, the position of the word in the list is checked and trigrams are formed accordingly.
Only if the index is greater than 1, trigrams of form ‘X_Y_$’ are created. (Let X and Y be any words).
And only if the index is greater than 0 and less than (length of global list -1), trigrams of form ‘X_$_Y’ are created. (Let X and Y be any words).

And only if the index is less than (length of global list -2), trigrams of form '$_X_Y' are created. (Let X and Y be any words).

All of these formed trigrams are stored in a list
Trigram list is then iterated and each trigram and its count (1 in this case as this mapper algorithm doesn't combine them locally) is printed in tab delimited (\t) format line by line (\n).

Similar to below snip:



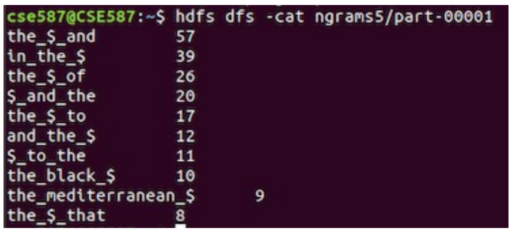
```
cse587@CSE587:~$ cat ngram.txt
outline_of_$    1
of_$_vol        1
$_vol_of        1
outline_of_$    1
of_$_vol        1
$_vol_of        1
outline_of_$    1
of_$_produced   1
$_produced_by   1
outline_of_$    1
of_$_a  1
$_a_plain       1
outline_of_$    1
of_$_is 1
$_is_certainly  1
of_many_$       1
many_$_it       1
$_it_will       1
outline_of_$    1
of_$_is 1
$_is_intended   1
```

This printed output from different mappers are sent to multiple reducers based on the trigram as input.

<u>Reducer Algorithm:</u> Libraries
used: sys
The tab delimited output is read as input for reducer. Each line from the input is stripped and split on tab (\t). Empty lines are ignored from the input. Now every line is a list with a trigram and its count from mapper. All these lists are stored in a global list.
An empty list and dictionary are initialized to store unique trigrams and word with its count respectively.
Global list is iterated, each word is checked in the unique trigram list. If the trigram is not present in unique trigram list, dictionary is updated with the trigram as key and its count as its value and the trigram is added to unique trigram list. But if the trigram is present in the unique trigram list, value of this trigram from the global list is added to the value corresponding to the trigram as key in the dictionary.
Items of this dictionary are sorted based on the count of the trigram with maximum count key-value in the first. And the resultant list is sliced till 10 to get top 10 frequently occurred trigrams.
Resulted list with top 10 trigrams and counts is iterated, trigram and its count are printed in tab delimited format (\t) line by line (\n). And also stored into a .txt file for next layer

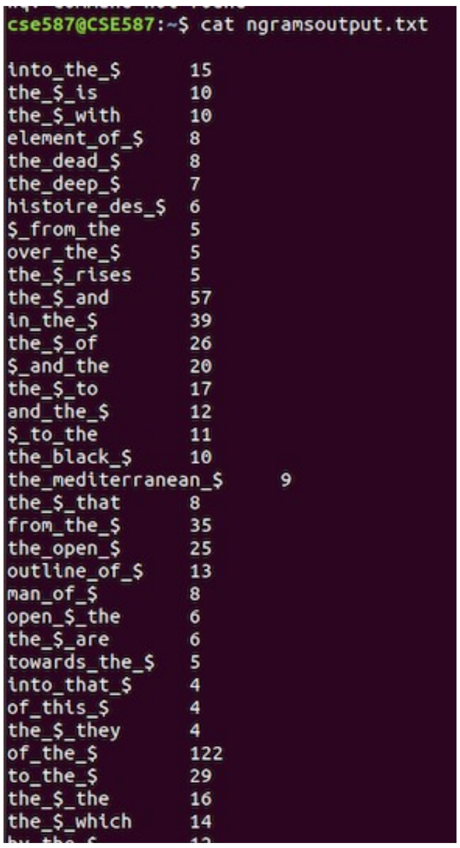
Each reducer gives top 10 frequently occurred trigrams within the reducer.
All these top 10 trigrams from each reducer are appended to a .txt file (ngramsoutput.txt).

Output from one of the reducer would be like :



```
cse587@CSE587:~$ hdfs dfs -cat ngrams5/part-00001
the_$_and          57
in_the_$           39
the_$_of           26
$_and_the          20
the_$_to           17
and_the_$          12
$_to_the           11
the_black_$        10
the_mediterranean_$        9
the_$_that         8
```

Aggregated output of all reducers:



```
cse587@CSE587:~$ cat ngramsoutput.txt

into_the_$       15
the_$_is         10
the_$_with       10
element_of_$     8
the_dead_$       8
the_deep_$       7
histoire_des_$   6
$_from_the       5
over_the_$       5
the_$_rises      5
the_$_and        57
in_the_$         39
the_$_of         26
$_and_the        20
the_$_to         17
and_the_$        12
$_to_the         11
the_black_$      10
the_mediterranean_$      9
the_$_that       8
from_the_$       35
the_open_$       25
outline_of_$     13
man_of_$         8
open_$_the       6
the_$_are        6
towards_the_$    5
into_that_$      4
of_this_$        4
the_$_they       4
of_the_$         122
to_the_$         29
the_$_the        16
the_$_which      14
```

<u>Layer2:</u>

Command used:
hadoop jar /home/cse587/hadoop-3.1.2/share/hadoop/tools/lib/hadoop-streaming-3.1.2.jar -file /home/cse587/mapper_ngrams2.py -mapper "python3 /home/cse587/mapperp2b.py" -file /home/cse587/reducer_ngrams2.py -reducer "python3 /home/cse587/reducerp2b.py" -input gutenberg/* -output ngramsoutput2

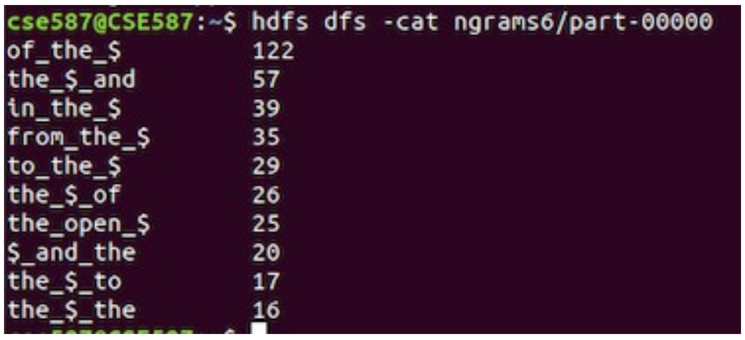
<u>Mapper Algorithm</u>:
Libraries used: sys

Text file from the layer1 reducer (ngramsoutput.txt) is considered as input for this mapper. Each line from the input is stripped and split on tab (\t). Empty lines are ignored from the input. Now every line is a list with a trigram and its count from each reducer.

Each trigram and its count are printed in tab delimited format (\t) line by line (\n).

Reducer Algorithm: Libraries
used: sys
The tab delimited output is read as input for this reducer. Each line from the input is stripped and split on tab (\t). Empty lines are ignored from the input.
Each with trigram and its count from each line is stored as key and value of a dictionary. All trigrams and its counts are stored in a dictionary.
Items of this dictionary are sorted based on the count of the trigram with maximum count key-value in the first. And the resultant list is sliced till 10 to get top 10 frequently occurred trigrams from this top 10 trigrams pooled from each reducer from layer1.
Resulted list with top 10 trigrams and counts is iterated, trigram and its count are printed in tab delimited format (\t) line by line (\n).

Below is the final output from layer2 reducer which are top 10 most occurred modified trigram in the dataset:


```
cse587@CSE587:~$ hdfs dfs -cat ngrams6/part-00000
of_the_$        122
the_$_and       57
in_the_$        39
from_the_$      35
to_the_$        29
the_$_of        26
the_open_$      25
$_and_the       20
the_$_to        17
the_$_the       16
```

PART 3: Inverted Index

For this part, mapper and reducer algorithms are coded in such a way that mapper picks every word along with document name and assigns 1 to each word and reducer gives the word and the documents in which they occurred along with the number of times they occurred in each document.

Number of mappers and reducers are defined in the command which is given to run the mapreduce algorithm to get wordcount.
We have checked output with and without defining number of mappers and reducers just to be sure whether the code is compatible to multi node cluster, we found that the outputs of both cases same.

Command used:

```
hadoop jar /home/cse587/hadoop-3.1.2/share/hadoop/tools/lib/hadoop-streaming-
3.1.2.jar -D mapred.map.tasks=3 -D mapred.reduce.tasks=3 -file
/home/cse587/mapperindex.py  -mapper  "python3  /home/cse587/mapperindex.py"  -file
/home/cse587/reducerindex.py -reducer "python3 /home/cse587/reducerindex.py" -input
gutenberg/*  -output invertedindex
```

Mapper Algorithm:
Libraries used: sys, os, re, nltk
All .txt files from Gutenberg folder will be considered as input for this mapper.
Mapper strips each line from the each of input .txt file and each line is split based on space to get each word of that line.
For each word, special characters and numbers are excluded from the word (only alphabets of a word are concated as a word), word is then changed to lower case, empty strings are also ruled out (not considered for stemming) and then the word is stemmed with the help of WordNetLemmatizer (.lemmatize function) and stemmed version of the word is obtained. Along with this stemmed word, document name of each line is obtained with the help of os.environ["map_input_file"] with which we can extract the path of the file and then regular expressions are used to pull out the exact document name. Even in this algorithm stop words are not removed from the documents.

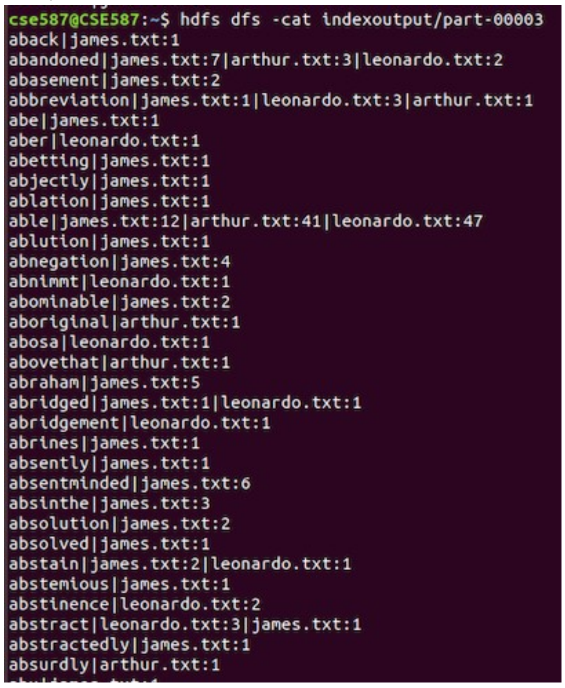
Now each stemmed word, its document name and its count (1 in this case as this mapper algorithm doesn’t combine them locally) is printed in tab delimited (\t) format line by line (\n).

Reducer Algorithm: Libraries used: sys
The tab delimited output is read as input for reducer. Each line from the input is stripped and split on tab (\t). Now every line is a list with word, document which it is in and its count in that document from mapper. All these lists are stored in a global list.
An empty list is initialized to store unique words. And an empty is dictionary initialized to store word as key and another dictionary as value and this dictionary has document name and word count in this document.
Global list is iterated, each word from each list in this global list is checked in the unique word list. If the word is not present in unique word list, dictionary is updated with the word as key and its value is a dictionary where document name is the key and the count as its value and the word is added to unique word list.
But if the word is present in the unique word list, value of this word is tried to update to the dictionary with document name. Try Except block are used for this operation, if the document name is not present in the value of outer dictionary and new dictionary with the count is updated to the value of the word in outer dictionary.

Format in which is saved would be like this:
{word1:{doc1:value,doc2:value,doc1:value}, word2:{doc1:value,doc2:value,….

Each time a word with doc id and value comes in, value gets added to the existing value if present else key and value is created and same is the case with the word.

Items in the outer dictionary are iterated, as every value of a key is a dictionary, the inner dictionary is being iterated.
Word with document names and its respective counts are printed with pipe separation (|) line by line (\n).
Output from one of the reducers of the inverted index looks like below:



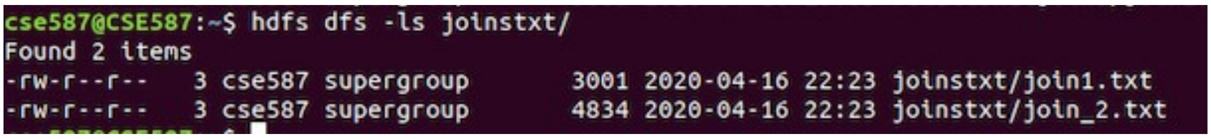
```
cse587@CSE587:~$ hdfs dfs -cat indexoutput/part-00003
aback|james.txt:1
abandoned|james.txt:7|arthur.txt:3|leonardo.txt:2
abasement|james.txt:2
abbreviation|james.txt:1|leonardo.txt:3|arthur.txt:1
abe|james.txt:1
aber|leonardo.txt:1
abetting|james.txt:1
abjectly|james.txt:1
ablation|james.txt:1
able|james.txt:12|arthur.txt:41|leonardo.txt:47
ablution|james.txt:1
abnegation|james.txt:4
abnimmt|leonardo.txt:1
abominable|james.txt:2
aboriginal|arthur.txt:1
abosa|leonardo.txt:1
abovethat|arthur.txt:1
abraham|james.txt:5
abridged|james.txt:1|leonardo.txt:1
abridgement|leonardo.txt:1
abrines|james.txt:1
absently|james.txt:1
absentminded|james.txt:6
absinthe|james.txt:3
absolution|james.txt:2
absolved|james.txt:1
abstain|james.txt:2|leonardo.txt:1
abstemious|james.txt:1
abstinence|leonardo.txt:2
abstract|leonardo.txt:3|james.txt:1
abstractedly|james.txt:1
absurdly|arthur.txt:1
```

PART 4: Relational Join

For this part, mapper and reducer algorithms are coded in such a way that mapper picks every line(row) of join files (the tables) and reducer combines the two tables for which employee id matches and gives us lines which are from joined table.

Number of mappers and reducers are defined in the command which is given to run the mapreduce algorithm to get wordcount.
We have checked output with and without defining number of mappers and reducers just to be sure whether the code is compatible to multi node cluster, we found that the outputs of both cases same.

Joins1 and joins2 csv files are converted into tab delimited text files and these files are stored in a folder and this folder was sent to has been copied to Hadoop distributed file system.



```
cse587@CSE587:~$ hdfs dfs -ls joinstxt/
Found 2 items
-rw-r--r--   3 cse587 supergroup       3001 2020-04-16 22:23 joinstxt/join1.txt
-rw-r--r--   3 cse587 supergroup       4834 2020-04-16 22:23 joinstxt/join_2.txt
```

It is observed that join1 has employee id, name and join2 has employee id, salary, country, passcode.

Command used:
hadoop jar /home/cse587/hadoop-3.1.2/share/hadoop/tools/lib/hadoop-streaming-3.1.2.jar -D mapred.map.tasks=2 -D mapred.reduce.tasks=2 -file /home/cse587/mapperjoins.py -mapper "python3 /home/cse587/mapperjoins.py" -file /home/cse587/reducerjoins.py -reducer "python3 /home/cse587/reducerjoins.py" -input gutenberg/* -output joinsoutput
<u>Mapper Algorithm</u>:

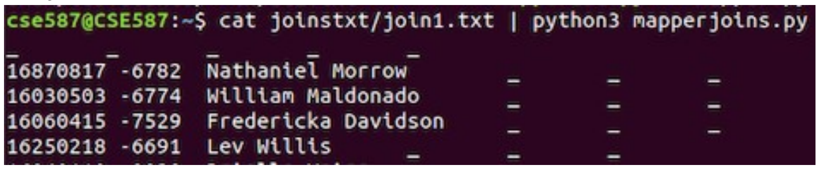
Libraries used: sys
All .txt files from joinstxt folder will be considered as input for this mapper.
Mapper strips each line from the each of input .txt file and each line is split based on tab (\t) and stored as a list.
Variables employee id, name, salary, country, passcode are initialized '_' for each line.
And based on the length of the list obtained, we can decide from which join table it is from.
Index is ignored from both the joins text file by looking for '-' in the first element of the list. If the length of the list is 2, it belongs to join1 and if its more, it's from join2.

For the list from join1, we populate employee id, name and rest are left with '_'. Similarly, for join2, we populate employee id, salary, country, passcode and rest are left with '_'.

For the line from join 2, we salary we recreated by concatenating second and third words of the list (as salary was split by ','). For country, we have to check whether fourth word and second word of the list are same, if found same it is assigned to country. If not matched they are concatenated and stored as country name. Finally, for variable passcode, last element of the list is populated (index = -1) as the list length arbitrary.

These variables are sent in employee id, name, salary, country, passcode format separated by tab (\t), line by line (\n).

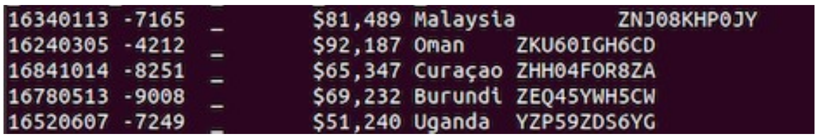
Output of Mapper is similar to this:
For join1,



```
cse587@CSE587:~$ cat joinstxt/join1.txt | python3 mapperjoins.py

_       _       _       _       _
16870817 -6782  Nathaniel Morrow        _       _       _
16030503 -6774  William Maldonado       _       _       _
16060415 -7529  Fredericka Davidson     _       _       _
16250218 -6691  Lev Willis      _       _       _
```

For join 2,



```
16340113 -7165  _       $81,489 Malaysia        ZNJ08KHP0JY
16240305 -4212  _       $92,187 Oman    ZKU60IGH6CD
16841014 -8251  _       $65,347 Curaçao ZHH04FOR8ZA
16780513 -9008  _       $69,232 Burundi ZEQ45YWH5CW
16520607 -7249  _       $51,240 Uganda  YZP59ZDS6YG
```

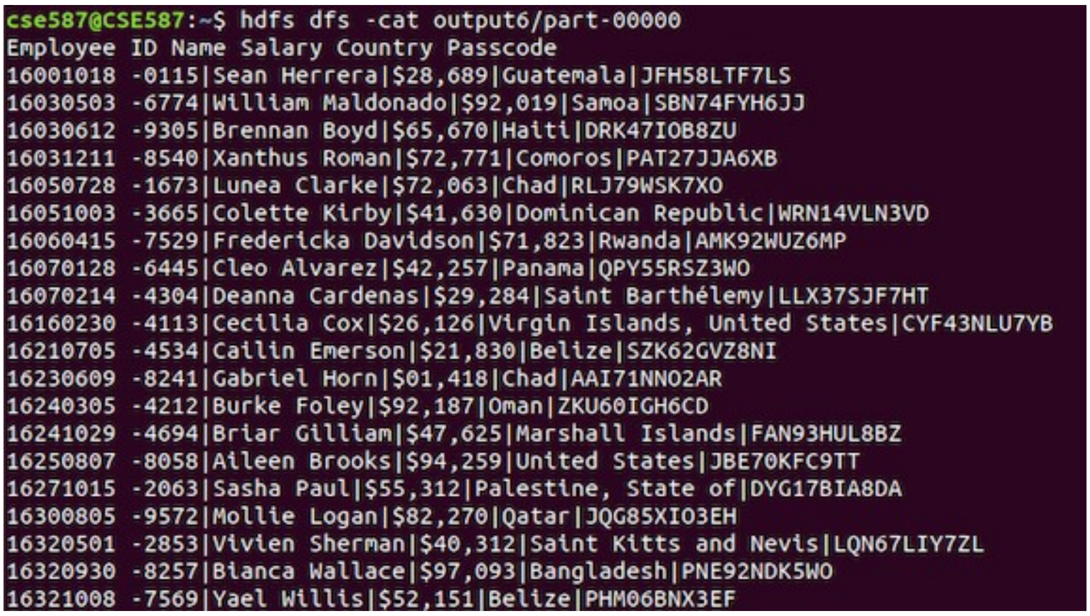
<u>Reducer Algorithm:</u> Libraries
used: sys
The tab delimited output is read as input for this reducer. Each line from the input is stripped and split on tab (\t). Now each line is a list

An empty list is initialized to store unique employee ids. And an empty is dictionary initialized to store employee id as key and a list as value and this list contains remaining variables.

For every list, first element of the list is checked in the unique employee id list. If the employee id is not present in unique employee id list, dictionary is updated with the employee id as key and remaining in a list as its value and the employee id is added to unique trigram list.
But if the employee id is present in the unique employee id list, value of this employee id from the dictionary is picked and checked whether first element of the list (which is name) is equal to '_' or not. With this condition we can get to know from which table is this line from and the remaining blank ('_') variables are populated with values.

Items of the resultant dictionary are iterated and populated employee id, name, salary, country, passcode are printed with pipe separation (|), line by line (\n).

Output from one of the reducers of relational join looks like below:



```
cse587@CSE587:~$ hdfs dfs -cat output6/part-00000
Employee ID Name Salary Country Passcode
16001018 -0115|Sean Herrera|$28,689|Guatemala|JFH58LTF7LS
16030503 -6774|William Maldonado|$92,019|Samoa|SBN74FYH6JJ
16030612 -9305|Brennan Boyd|$65,670|Haiti|DRK47IOB8ZU
16031211 -8540|Xanthus Roman|$72,771|Comoros|PAT27JJA6XB
16050728 -1673|Lunea Clarke|$72,063|Chad|RLJ79WSK7XO
16051003 -3665|Colette Kirby|$41,630|Dominican Republic|WRN14VLN3VD
16060415 -7529|Fredericka Davidson|$71,823|Rwanda|AMK92WUZ6MP
16070128 -6445|Cleo Alvarez|$42,257|Panama|QPY55RSZ3WO
16070214 -4304|Deanna Cardenas|$29,284|Saint Barthélemy|LLX37SJF7HT
16160230 -4113|Cecilia Cox|$26,126|Virgin Islands, United States|CYF43NLU7YB
16210705 -4534|Cailin Emerson|$21,830|Belize|SZK62GVZ8NI
16230609 -8241|Gabriel Horn|$01,418|Chad|AAI71NNO2AR
16240305 -4212|Burke Foley|$92,187|Oman|ZKU60IGH6CD
16241029 -4694|Briar Gilliam|$47,625|Marshall Islands|FAN93HUL8BZ
16250807 -8058|Aileen Brooks|$94,259|United States|JBE70KFC9TT
16271015 -2063|Sasha Paul|$55,312|Palestine, State of|DYG17BIA8DA
16300805 -9572|Mollie Logan|$82,270|Qatar|JQG85XIO3EH
16320501 -2853|Vivien Sherman|$40,312|Saint Kitts and Nevis|LQN67LIY7ZL
16320930 -8257|Bianca Wallace|$97,093|Bangladesh|PNE92NDK5WO
16321008 -7569|Yael Willis|$52,151|Belize|PHM06BNX3EF
```

BONUS: K-Nearest Neighbors

For this part, mapper and reducer algorithms are coded in such a way that mapper loads entire testing data and training data is given as input. Later distances of each testing sample with each training sample is computed and is sent to reducer along with its label.
Reducer combines the all distance and its labels for every test sample and picks the nearest neighboring labels based on specified k value and the most frequent label is assigned to the test sample.

Number of mappers and reducers are defined in the command which is given to run the mapreduce algorithm to get wordcount.
We have checked output with and without defining number of mappers and reducers just to be sure whether the code is compatible to multi node cluster, we found that the outputs of both cases same.

Command used:
```
hadoop jar /home/cse587/hadoop-3.1.2/share/hadoop/tools/lib/hadoop-streaming-
3.1.2.jar -D mapred.map.tasks=4 -D mapred.reduce.tasks=4 -file
/home/cse587/mapperknn.py -mapper "python3 /home/cse587/mapperknn.py" -file
/home/cse587/reducerknn.py -reducer "python3 /home/cse587/reducerknn.py" -input
knn/Train.csv  -output knnoutput
```

Mapper Algorithm:
Libraries used: sys, Pandas, NumPy
Csv file of training data is taken as input and test data is loaded within the mapper.
In case of multiple mappers, training data is split and sent to all mapper but test data is loaded within a mapper.
Test data is loaded using Pandas and later converted into NumPy for further use.
Mapper strips each line from the each of input training data and each line is split based on comma (,).
Index is ignored and each row of remaining data (each line) is stored into a NumPy array as a list and label which is the last element of the list is stored into a label list.
Training data in NumPy array is converted to float and labels are converted into integer.

Test data NumPy array and Train data NumPy array are normalized using mean and std in NumPy library.

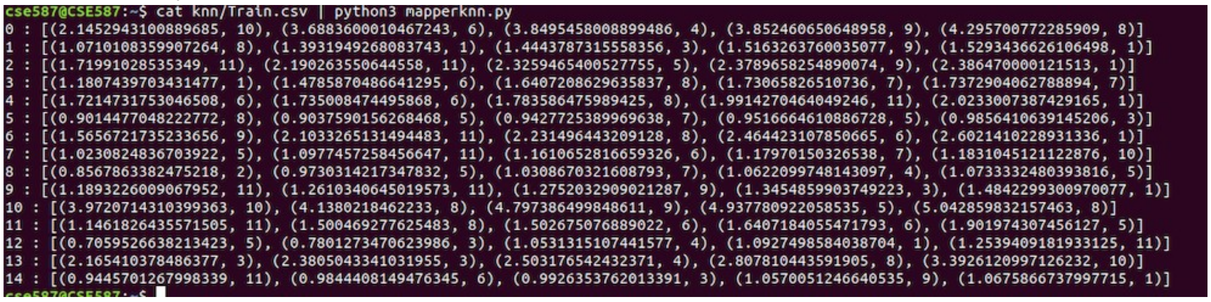
As the training data is huge, normalizing the data separately does not show much affect and is almost all similar to the data normalized before loading.(Results are observed to be same for multiple mapper which implies multiple pieces of individually normalized train data).

An empty distance list is initialized to store distances. Loaded test data is being iterated, distance of each test sample to each training sample is computed and stored in the distance matrix in such a way that each row represents a test sample has a list of size equal to length of training data and it stores the distance of that test sample to each training sample.

Distances are sorted across each row of the matrix and with the help of NumPy functions and the indices of the sorted distances across each row is used to get re arranged labels (in such a way that nearest label is in the front for every row i.e., test sample).

A loop is iterated over length of test data and during each iteration row number along with zipped nearest distances along with its corresponding labels are printed, line by line.

Output from any of the mapper would be in the following way if the 5 nearest labels with distances are picked:



```
cse587@CSE587:~$ cat knn/Train.csv | python3 mapperknn.py
0 : [(2.1452943100889685, 10), (3.6883600010467243, 6), (3.849545800889486, 4), (3.852460650648958, 9), (4.295700772285909, 8)]
1 : [(1.0710108359907264, 8), (1.3931949268083743, 1), (1.4443787315558356, 3), (1.5163263760035077, 9), (1.5293436626106498, 1)]
2 : [(1.71991028535349, 11), (2.190263550644558, 11), (2.3259465400527755, 5), (2.3789658254890074, 9), (2.386470000121513, 1)]
3 : [(1.1807439703431477, 1), (1.4785870486641295, 6), (1.6407208629635837, 8), (1.73065826510736, 7), (1.7372904062788894, 7)]
4 : [(1.7214731753046508, 6), (1.735008474495868, 6), (1.783586475989425, 8), (1.9914270464049246, 11), (2.0233007387429165, 1)]
5 : [(0.9014477048222772, 8), (0.9037590156268468, 5), (0.9427725389969638, 7), (0.9516664610886728, 5), (0.9856410639145206, 3)]
6 : [(1.5656721735233656, 9), (2.1033265131494483, 11), (2.231496443209128, 8), (2.464423107850665, 6), (2.6021410228931336, 1)]
7 : [(1.0230824836703922, 5), (1.0977457258456647, 11), (1.1610652816659326, 6), (1.17970150326538, 7), (1.1831045121122876, 10)]
8 : [(0.8567863382475218, 2), (0.9730314217347832, 5), (1.0308670321608793, 7), (1.0622099748143097, 4), (1.0733332480393816, 5)]
9 : [(1.1893226009067952, 11), (1.2610340645019573, 11), (1.2752032909021287, 9), (1.3454859903749223, 3), (1.4842299300970077, 1)]
10 : [(3.9720714310399363, 10), (4.1380218462233, 8), (4.797386499848611, 9), (4.937780922058535, 5), (5.042859832157463, 8)]
11 : [(1.1461826435571505, 11), (1.500469277625483, 8), (1.502675076889022, 6), (1.6407184055471793, 6), (1.901974307456127, 5)]
12 : [(0.7059526638213423, 5), (0.7801273470623986, 3), (1.0531315107441577, 4), (1.0927498584038704, 1), (1.2539409181933125, 11)]
13 : [(2.165410378486377, 3), (2.3805043341031955, 3), (2.503176542432371, 4), (2.807810443591905, 8), (3.3926120997126232, 10)]
14 : [(0.9445701267998339, 11), (0.9844408149476345, 6), (0.9926353762013391, 3), (1.0570051246640535, 9), (1.0675866737997715, 1)]
cse587@CSE587:~$
```

Output of the mapper can be scaled to pick top k nearest to send or all of the distances and labels can be sent to mapper. As we pick a particular k nearest among labels, scaling the data before sending to mapper does not affect the end goal. But as sending all of the data is not affecting computational time for this dataset, all of the computed data is being sent to reducer.

Reducer Algorithm:

Libraries used: sys, Pandas, NumPy

Three empty lists are initialized to store unique row numbers, obtained distances and labels respectively.

Output from mapper is read as input for reducer. Each line from the input is stripped and split on colon (:). And the second part is split in such a way that each distance and its corresponding labels are extracted for a particular row number and stored into separate list with one to one correspondence (with index for a distance and its label).

Now index is checked in the unique row list and if present then lists of distances and its labels obtained are appended to the existing records found based on the index of its row number in unique row list. If not found a new entry in global distances list and global labels list is created.

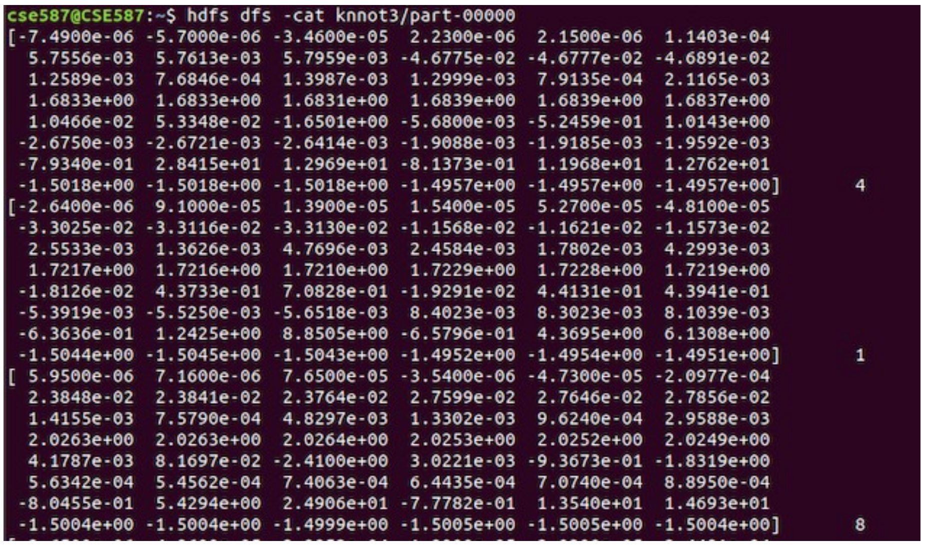
By end of the loop, we have a unique row number list which has row numbers. A Distances matrix which has distances for each test for each train with each row representing the test row number in unique row list with same index and similar is the case with labels matrix.

Similar to mapper, distances are sorted across each row of the matrix and with the help of NumPy functions and the indices of the sorted distances across each row is used to get re arranged labels (in such a way that nearest label is in the front for every row i.e., test sample).

The sorted list has been scaled with specific 'k' value.
Now with the help of NumPy functions like argmax and bincount, the most frequent occurring label for each test data is picked.

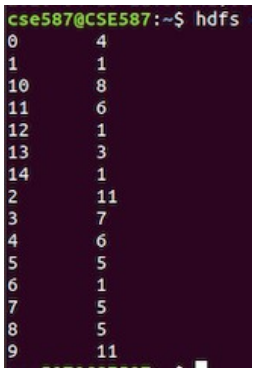
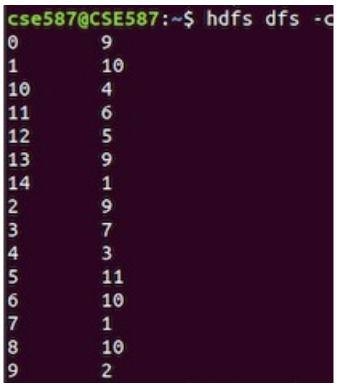
Now test data set is loaded into reducer.
Now a loop of test data length is started and the test row of each row number of unique row list along with predicted label of that row is printed by separating with tab (\t).

Output in test row and its predicted label form: For
k = 15,



```
cse587@CSE587:~$ hdfs dfs -cat knnot3/part-00000
[-7.4900e-06 -5.7000e-06 -3.4600e-05  2.2300e-06  2.1500e-06  1.1403e-04
  5.7556e-03  5.7613e-03  5.7959e-03 -4.6775e-02 -4.6777e-02 -4.6891e-02
  1.2589e-03  7.6846e-04  1.3987e-03  1.2999e-03  7.9135e-04  2.1165e-03
  1.6833e+00  1.6833e+00  1.6831e+00  1.6839e+00  1.6839e+00  1.6837e+00
  1.0466e-02  5.3348e-02 -1.6501e+00 -5.6800e-03 -5.2459e-01  1.0143e+00
 -2.6750e-03 -2.6721e-03 -2.6414e-03 -1.9088e-03 -1.9185e-03 -1.9592e-03
 -7.9340e-01  2.8415e+01  1.2969e+01 -8.1373e-01  1.1968e+01  1.2762e+01
 -1.5018e+00 -1.5018e+00 -1.5018e+00 -1.4957e+00 -1.4957e+00 -1.4957e+00]       4
[-2.6400e-06  9.1000e-05  1.3900e-05  1.5400e-05  5.2700e-05 -4.8100e-05
 -3.3025e-02 -3.3116e-02 -3.3130e-02 -1.1568e-02 -1.1621e-02 -1.1573e-02
  2.5533e-03  1.3626e-03  4.7696e-03  2.4584e-03  1.7802e-03  4.2993e-03
  1.7217e+00  1.7216e+00  1.7210e+00  1.7229e+00  1.7228e+00  1.7219e+00
 -1.8126e-02  4.3733e-01  7.0828e-01 -1.9291e-02  4.4131e-01  4.3941e-01
 -5.3919e-03 -5.5250e-03 -5.6518e-03  8.4023e-03  8.3023e-03  8.1039e-03
 -6.3636e-01  1.2425e+00  8.8505e+00 -6.5796e-01  4.3695e+00  6.1308e+00
 -1.5044e+00 -1.5045e+00 -1.5043e+00 -1.4952e+00 -1.4954e+00 -1.4951e+00]       1
[ 5.9500e-06  7.1600e-06  7.6500e-05 -3.5400e-06 -4.7300e-05 -2.0977e-04
  2.3848e-02  2.3841e-02  2.3764e-02  2.7599e-02  2.7646e-02  2.7856e-02
  1.4155e-03  7.5790e-04  4.8297e-03  1.3302e-03  9.6240e-04  2.9588e-03
  2.0263e+00  2.0263e+00  2.0264e+00  2.0253e+00  2.0252e+00  2.0249e+00
  4.1787e-03  8.1697e-02 -2.4100e+00  3.0221e-03 -9.3673e-01 -1.8319e+00
  5.6342e-04  5.4562e-04  7.4063e-04  6.4435e-04  7.0740e-04  8.8950e-04
 -8.0455e-01  5.4294e+00  2.4906e+01 -7.7782e-01  1.3540e+01  1.4693e+01
 -1.5004e+00 -1.5004e+00 -1.4999e+00 -1.5005e+00 -1.5005e+00 -1.5004e+00]       8
```

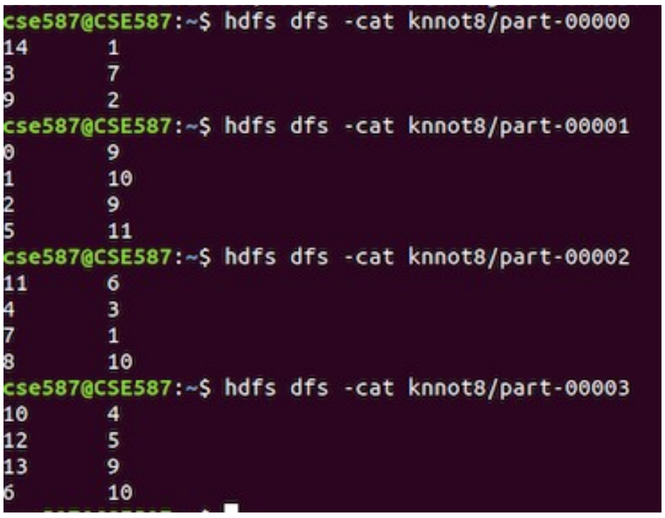
Output in test row number and its predicted label form:

Before normalizing the data	After normalizing the data
 Row Number | Predicted label	 Row Number | Predicted label

```
cse587@CSE587:~$ hdfs
0       4
1       1
10      8
11      6
12      1
13      3
14      1
2       11
3       7
4       6
5       5
6       1
7       5
8       5
9       11
```

```
cse587@CSE587:~$ hdfs dfs -c
0       9
1       10
10      4
11      6
12      5
13      9
14      1
2       9
3       7
4       3
5       11
6       10
7       1
8       10
9       2
```

For multiple mapper and reducers, we can observe that the result is same and does not differ even if data is normalized separately as the training data is huge. Output from each reducer with individually normalized training data:



```
cse587@CSE587:~$ hdfs dfs -cat knnot8/part-00000
14      1
3       7
9       2
cse587@CSE587:~$ hdfs dfs -cat knnot8/part-00001
0       9
1       10
2       9
5       11
cse587@CSE587:~$ hdfs dfs -cat knnot8/part-00002
11      6
4       3
7       1
8       10
cse587@CSE587:~$ hdfs dfs -cat knnot8/part-00003
10      4
12      5
13      9
6       10
```