ELEN4020A - Data Intensive Computing
Lecturer: Prof E. J. Otoo
16 April 2019

Laboratory 3: Using MapReduce Framework

Tyson Cross 1239448   Michael Nortje 1389486   Josh Isserow 675720

Abstract

MapReduce techniques were used to process several text files of different lengths with word-count and indexing algorithms in Python with the mrjob 0.6.8 framework for the Hadoop 2.9.2 libraries. Three algorithms were developed: to return the number of occurrences of each word in a text; to return the top-K frequencies of word occurrence in a text; and to return an inverted line index for each word in a text.

1 Introduction

A recent projection of the global growth in the Big Data market has forecast the segment to exceed a valuation of $70 billion by 2022, almost doubling the current market size [1]. The use of very large data sets is central to the future plans of many major corporations and international organisations, with the quantity of global big data predicted to rise to above 40,000 Exabytes by 2020 [2].

One of the most important tools to help process these very large data sets is MapReduce, a framework capable of processing copious amounts of data in a fault-tolerant method [3] using implementations such as the Apache Hadoop framework, written in Java. MapReduce can utilise massive parallel clusters for data processing, accessing terabyte and petabyte level of data [3]. The principal behind MapReduce is to utilise an uncomplicated library capable of parallelization, distribution of data and load-balancing.

This report explains the use of a framework to implement of MapReduce for three algorithms: to return the number of occurrences of each word in a text; to return the top-K frequencies of word occurrence in a text; and to return an inverted line index for each word in a text.

2 MapReduce

MapReduce is central to the Apache Hadoop library [3]. A diagram showing a high level overview of the data-flow steps in MapReduce is shown in Figure 1. There are two basic methods of operation: `map` and `reduce`. Map takes a single key and value pair as input, to produce a (key,values) output (the mapper can also return nothing, if no values for the input key are found). Reduce uses the output produced by Map and amalgamates the key/values pairs.

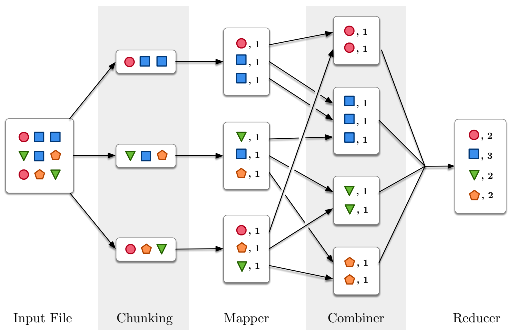


Fig. 1: MapReduce diagram

There is also an optional extra optimisation step, `combiner` which takes a key, and subset of values, and returns none or more key/values pairs, to decrease the overall amount of data passing through the system before being handed to the reducer. MapReduce also can include partitioners, that split up the key space before reduction, and counters which can be used to provide statistics for analysis of performance of the mapper and reducer operations.

When a program conforms to the MapReduce standard (such as Hadoop), parallelization and execution are managed from the controlling node, and carried out by the workers, which can be affordable commodity machines rather than specialised cluster computers [4], [5]. Run-time systems assume responsibility of separating input data, arranging execution of the programs, managing failures experienced by the machines as well inter-machine communication. This simplifies the ability to utilise large distributed systems without prior parallel system experience [4]. Part of the Hadoop framework is the Hadoop Distributed File System (HDFS), which is intended to be highly fault-tolerance, assuming that hardware failure involving large networks of many machines can occur at any time. The framework used for this laboratory, mrjob 0.6.8 supports multiple non-HDFS systems.

The MapReduce framework consists of two basic components: a main controlling node (either a "job tracker" or "resource manager"), and the number of worker nodes, the controller schedules and directs the jobs, which are a series of tasks (or steps) executed by the workers [6].

2.1 mrjob

Mrjob is an Apache Licensed (Version 2.0) Hadoop framework which allows for easy routing for programs in Python that run on Hadoop [6]. The framework allows for easy writing and management of MapReduce jobs. With mrjob, the jobs are written as classes that execute functions according to a `step` order, controlling of the flow of execution.

Mrjob is well documented, and lets the user run code in Python instead of Java without the need for directly controlling or installing Hadoop [6]. This allows for writing and running tests for MapReduce using the Hadoop streaming back-end. The framework offers a consistent interface for all environments that are supported - i.e. the Python code itself does not change depending on the operating system or hardware that it is run on. Mrjob offers easy debugging through the use of local error trace-backs in the console, rather than in a log file. Part of mrjob's ease of use is that it automatically handles serialization and de-serialization of input and output key/value pairs. This removes the need for manually loading and dumping `json` objects.

One disadvantage of using mrjob is actually its ease of use. Since it is greatly simplified, it does not have the same access to the Hadoop API that a lower-level framework might offer.

3 ALGORITHMS

MapReduce was designed and intended to be highly scalable, able to perform data operations on a local computer, or across very large clusters of multiple machines with their own storage. For the laboratory code execution a single 4-core laptop was used. All algorithms were implemented in Python 3.7 using the mrjob API and the Hadoop back-end. The inputs for the laboratory are plain-text files, of varying lengths and line counts. An additional input file required is a stop-words text file, which lists all the words that should be discounted. Stop words are commonly used to filter out common joining words so that only the data considered useful remains [7].

3.1 Word Count

A simple word count algorithm providing the number of occurrences of all words in a given text was implemented. The implemented code is listed in Code B1.

The algorithm is implemented as a class `WordCount`, which inherits from the principle `MRJob` class. The steps of the algorithm are as follows: first an initialisation mapper step (`mapper_init`) loads and separates out the stop words onto an property on the main class. The input text file is automatically split up into chunks, and distributed to the mappers before the main mapper step, shown as pseudocode in Algorithm 1:

Input: line
Output: word, 1
for *each* `word` *in* `line` *(stripped of punctuation)* **do**
  make `word` lower case
  if `word` *not in* `stop_words` **then**
    return word, 1
end

Algorithm 1: Word Count: mapper

Each mapper finds a single unique instance of a word in a line (in the backend, storing an internal local and total byte-offset to determine if a word has been considered yet). The next step creates combiners, which sort the summed sub-values (word occurrences per mapper per line) received for each key, which are then passed to the reducer step, which sums the total word occurrences across the entire input text. These value are sorted by mrjob and Hadoop streaming, before printing the final steps' generator values to `std.out` as a list of words and their count in the text.

A command-line argument is included for the python script, to allow for providing a required external custom stop-words file. The file can be empty. By default, the stop words file listed in Table A2 was used for all of the processing of the texts, except for *Hamlet*, which used the list of words shown in Table A3, due to the Elizabethan language in the play.

The `mapper` functions are responsible for processing the words from the input text file one line at a time. The combiner function is responsible for combining sub-values associated with each key. The reducer functions perform the addition of the values for each key of numbers, yielding the final summed (key,value) pairs. In order to control the individual task steps, the mrjob scheduler function `MRStep` was overridden, with manually defined mapper, combiner and reducer functions.

3.2 Top K Word Frequencies

This algorithm returns a list of the K most frequently occurring words in a given text, excluding stop words, for K = 10 and K = 20. The implemented code is shown in Code B2. The steps of the algorithm are similar to the previous word count, only with an extra reducer step. In this final reducer stage, the last and current frequency of word incidences are kept track of inside a nested conditional statement within a *for* loop, which traverses through the sorted word/counts pairs, and stops returning values once K values have been returned. This step is just a post-process step to restrict the computed key/values being returned, and altering K has no affect on the asymptotic behaviour of the algorithm. Because the entire text has to be counted (and so every word must be considered) it is unclear how K could be used to more substantially shorten the number of execution steps required, given the parallel process of computation.

Input: word_counts
$K \leftarrow$ user input
i $\leftarrow$ 0
last_freq $\leftarrow$ 0
current_freq $\leftarrow$ 0
for *each* `count` *and* `key` *in* `word_counts` *(in descending order)* **do**
  if *i equal to K* **then**
    `current_freq` $\leftarrow$ `count`
  else if *i less than K* **then**
    `last_freq` $\leftarrow$ `current_freq`
    `current_freq` $\leftarrow$ `count`
    if `last_freq` *not equal to* `current_freq` **then**
      i $\leftarrow$ i+1
    end
end

Algorithm 2: Word Frequency: reducer_sort_counts

The frequency limit was implemented as a command-line argument, which defaults to the top 10 if not provided explicitly. The algorithm returns the number of most common frequencies, which can be more or less than the number of words returned. This is because more than one word can occur at the same frequency of occurrence, or there could be fewer words than the requested number of frequencies. Because the entire text must be processed and then sorted before determining the most common frequencies, there is no performance or speed difference irrespective of the number of top frequencies requested.

The top 10 word frequency results for the test texts are shown in Figure A.1f.

3.3 Inverted Line Index

This algorithm provides the line numbers where a particular word can be found. Each individual mapper only receives an individual line of text, with no idea of where in the text it was located. When returning the processed key (word) and value (line), it quickly becomes difficult for a human to read the results, particularly with long line lengths. The implemented code is shown in Code B3.

One solution to contextualise the line index, while still returning each entire line of text with each word, is to pre-process the input text, pre-pending each line with a number. The provided text for the laboratory, `File2ForLab3.txt` (a Gutenberg plain-text copy of Plato's *Republic*) has already been pre-processed in this manner. For the testing in this report, multiple additional texts were used. For brevity and readability, instead of returning the entire line of test pre-pended with a line-number, it was decided to return only the line number, and the integrate the pre-processing programatically.

```
Input: word_counts
for each num and line in f do
    if i greater than or equal to K then
        break
    for each word in line do
        word ← lowercase word
        if word not a stopword then
            yield word and num
        end
    end
end
```

Algorithm 3: Word Inverse Index: mapper_raw

To accomplish this, the `mapper_raw` function was used, which provides a path to the entire file, available for each mapper. The psuedocode for this function is shown in Algorith, 3. Mrjob and Hadoop take care of all the low level data processing, providing the correct locations, lines and accessible local copies of the input file. The mapper first enumerates each line, and then performs the word processing. The function then yields a generator for each line, returning the key (word) and value (line number). This is much easier for a human to read the final returned result. A optional command-line argument was added to limit how many lines of the file to process, and another option to limit the number of indexed lines that are listed per word in the output.

4 Performance evaluation

Several differently sized tests from the Gutenberg project were used as data for processes, listed in Table A1. The files were input as UTF-8 encoded plain-text files, with the Gutenberg metadata stripped. The timing of the results is shown in Figure A.2. Timing was performed using the datetime function, and output using a call to `sys.stderr.write` to output the delta time to the console.

The Inverse Index algorithm is extremely rapid when asked to only process 50 lines, returning an unsorted result in less than 165 ms, with an average of 150 ms (for all texts). When the entire text is processed, with no limit of the number of indexed lined returned per word, the shortest text *Symbols and Signs* with 200 lines returned a result in 208 ms. The longest test, *In Search of Lost Time* (comprised of all 7 volumes of the English translation of Marcel Proust's *À la recherche du temps perdu* at 116365 lines) returns the indexed results in 8421 ms. The most common word, 'time', occurs on 2695 lines in the text.

5 Conclusion

A variety of common data tasks was processed using the MapReduce framework. A word count algorithm, K-query algorithm and inversion index of text algorithm were written and evaluated, using Python and the mrjob framework. The algorithms were run on a variety of short, medium and long texts. The performance of the algorithms were timed, and the results were compared and discussed.

References

[1] Wikibon.com, "Big data market revenue forecast worldwide 2011-2027," https://www.statista.com/statistics/254266/global-big-data-market-forecast/, March 2019, (Accessed on 04/15/2019).

[2] D. Srivastava and D. Sinwar, "Statistical analysis of big data using hadoop: A review," vol. 5, 01 2018.

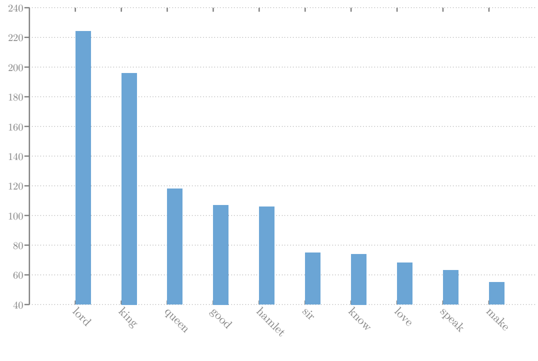
[3] S. N. Khezr and N. J. Navimipour, "Mapreduce and its applications, challenges, and architecture: a comprehensive review and directions for future research," *Journal of Grid Computing*, vol. 15, no. 3, pp. 295–321, 2017.

[4] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

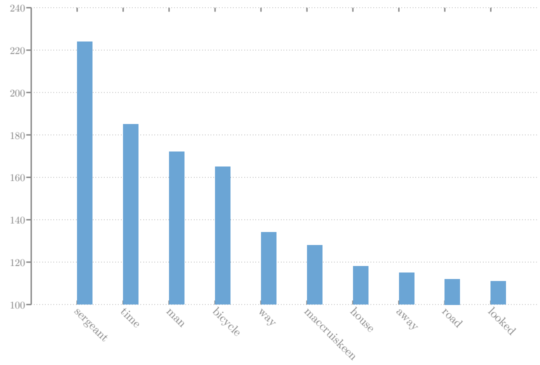
[5] S. Maitrey and C. Jha, "Mapreduce: simplified data analysis of big data," *Procedia Computer Science*, vol. 57, pp. 563–571, 2015.

[6] Yelp and Contributors, "MRJob v0.6.8.dev0 API Documentation," https://mrjob.readthedocs.io/, January 2019, (Accessed on 04/15/2019).

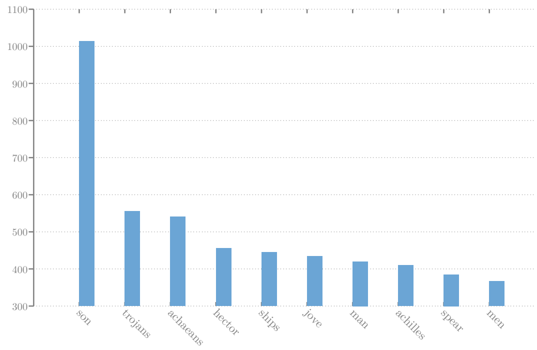
[7] A. Rajaraman and J. D. Ullman, *Mining of massive datasets*. Cambridge University Press, 2011.



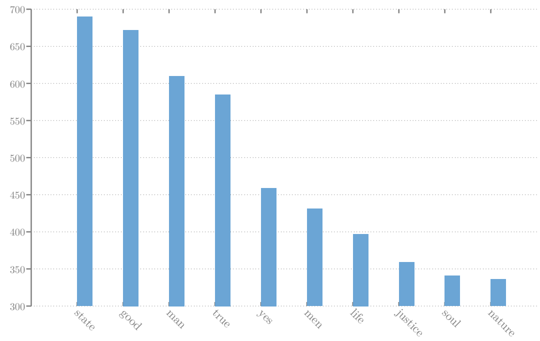
(a) Top 10 Word Frequencies in 'Hamlet'



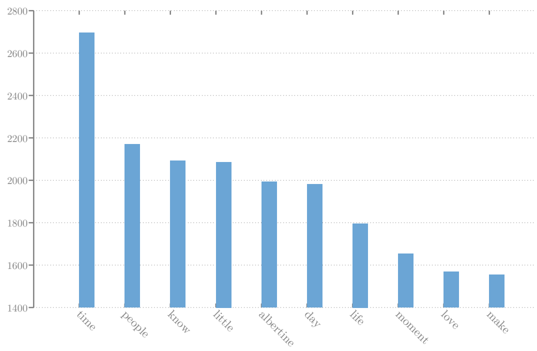
(b) Top 10 Word Frequencies in 'The Third Policeman'



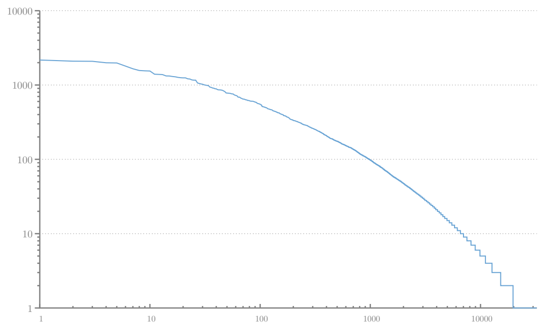
(c) Top 10 Word Frequencies in 'The Iliad'



(d) Top 10 Word Frequencies in 'The Republic'



(e) Top 10 Word Frequencies in 'In Search of Lost Time'



(f) Top 1000 Words Frequencies in 'In Search of Lost Time'

Fig. A.1: Main Caption

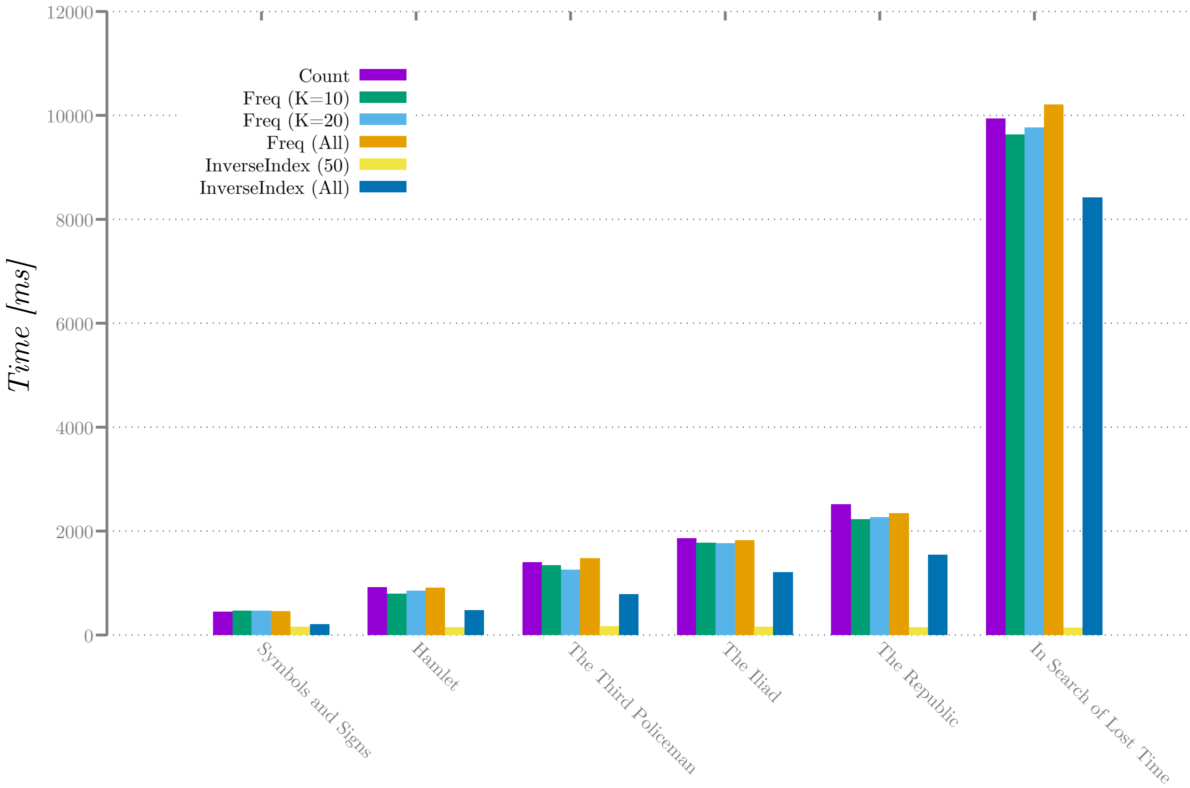


Fig. A.2: Algorithm Timings

TABLE A1: Texts Used in Testing

Text	**Author**	**Words**	**Lines**	**Characters**
Symbols and Signs	Vladimir Nabokov	2230	206	12580
Hamlet	William Shakespeare	32065	4460	191726
The Third Policeman	Flann O'Brien	74728	3729	403872
Iliad	Homer	153320	13440	808161
The Republic	Plato	217340	24295	1195060
In Search of Lost Time	Marcel Proust	1361977	116365	7498967

TABLE A2: `stop_words.txt`

```
[numerals 0-100] [numerals for dates 1990-2020] [punctuation marks] '|'i 'll a about
above across after afterwards again against all almost alone along already also
although always am among amongst amount an and another any anyhow anyone anything
anyway anywhere are around as at b back be because been before beforehand being below
beside besides between both bottom but by c call can cannot cant co come con could
couldnt d de did didn't do does doesn't doing don don't done down due during e each
eg eight either eleven else elsewhere enough etc even ever every everyone everything
everywhere except f few fifteen fifty fill find fire first five for former formerly
forty found four from front full further g get give go h had has hasnt have having
he hence her here hereafter hereby herein hereupon hers herself him himself his how
however hundred i i'll ie if in inc indeed into is it its itself j just k keep l last
latter latterly least less like ll ltd m made many may me meanwhile might mill mine
mme more moreover most mostly move much must my myself n name namely neither never
nevertheless next nine no nobody none noone nor not nothing now nowhere o of off often
on once one only onto or other others otherwise our ours ourselves out over own p part
per perhaps please put q r rather re s said same say see seem seemed seeming seems
serious several she should since six sixty so some somehow someone something sometime
sometimes somewhere still such system t take ten than that the their theirs them
themselves then there thereafter thereby therefore therein thereupon these they thing
third this those though three through throughout thru thus to together too toward
towards twelve twenty two u un under until up upon us v very via w was we we'll well
were what whatever when whenever where whereafter whereas whereby wherein whereupon
wherever whether which while who whoever whole whom whose why will with within without
would x y yet you your yours yourself yourselves z
```

TABLE A3: `stop_words_shakespeare.txt`

```
[numerals 0-100] [numerals for dates 1990-2020] [punctuation marks] 'tis 'a 'a' 'and
'as 'but 'by 'for 'he 'if 'in 'it 'o 's 'seems 'should' 'so 't 'that 'there 'thine
'thus 'tis 'to 'twas 'tween 'twould 'we 'well 'while 'would a a' about above across
after afterwards again against all almost alone along already also although always
am among amongst amoungst amount an and another any anyhow anyone anything anyway
anywhere are around art as at ay b back be because been before beforehand being below
beside besides between both bottom but by c call can cannot cant claud co come con
could couldnt d de did didn't do does doesn't doing don don't done dost doth down due
during e each eg eight either eleven else elsewhere enough enter ere etc even ever
every everyone everything everywhere except exeunt exit f few fifteen fifty fill find
fire first five for former formerly forty found four from front full further g ger get
give go h had ham has hasnt hast hath have having he hence hence her here hereafter
hereby herein hereupon hers herself him himself his hither ho hor how however hundred
i i' i'll ie if in inc indeed into is it its itself j just k keep l laer last latter
latterly least less let like ll ltd m made many may me meanwhile might mill mine more
moreover most mostly move much must my myself n name namely neither never nevertheless
next nigh nine no nobody none noone nor not nothing now nowhere o o'er of off oft
often on once one only onto oph or other others otherwise our ours ourselves out
over own p part per perhaps please pol put q r rather re ros s same see seem seemed
seeming seems serious several shall she should since six sixty so some somehow someone
something sometime sometimes somewhere st still such system t take ten th' than that
the thee their theirs them themselves then thence there thereafter thereby therefore
therein thereupon these they thine thing third this thither those thou though three
through throughout thru thus thy tither to together too toward towards twelve twenty
two u un under until up upon us v very via w was wast we we'll well were what whatever
when whence whenever where whereafter whereas whereby wherefore wherein whereto
whereupon wherever whether which while whither who whoever whole whom whose why
will with withal within without would x y ye yet yon yonder you your yours yourself
yourselves z
```

APPENDIX B

Code B1: WordCount.py

```
from mrjob.job import MRJob
from mrjob.step import MRStep
from datetime import datetime
import sys
import re

WORD_RE = re.compile(r"[\w']+")

class WordCount(MRJob):
    '''Returns the number of occurances of words in the input text'''

    FILES = ['stop_words.txt']
    SORT_VALUES = True

    def configure_args(self):
        super(WordCount, self).configure_args()
        self.add_file_arg(
            '--stop-words',
            metavar='STOP_WORDS_FILE',
            dest='stop_words',
            type=str,
            default='stop_words.txt',
            help='Input stop words text file')

    def steps(self):
        return [
            MRStep(
                mapper_init=self.mapper_init,
                mapper=self.mapper_get_words,
                combiner=self.combiner,
                reducer=self.reducer,
            ),
            MRStep(
                reducer=self.reducer_sort_counts,
            )
        ]

    def mapper_init(self):
        with open(self.options.stop_words) as f:
            self.stop_words = set(line.strip() for line in f)

    def mapper_get_words(self, _, line):
        for word in WORD_RE.findall(line):
            word = word.lower()
            if word not in self.stop_words:
                yield (word.lower(), 1)

    def combiner(self, word, counts):
        yield (word, sum(counts))

    def reducer(self, key, values):
        yield None, (sum(values), key)

    def reducer_sort_counts(self, _, word_counts):
        for count, key in sorted(word_counts, reverse=True):
            yield (count, key)

if __name__ == '__main__':
    start_time = datetime.now()
    WordCount.run()
    end_time = datetime.now()
    elapsed_time = (end_time - start_time)*1000
    sys.stderr.write("Total Seconds WordCount.py: ({0}) microseconds\n".format(elapsed_time.total_seconds()))
```

Code B2: WordFreq.py

```
from mrjob.job import MRJob
from mrjob.step import MRStep
from datetime import datetime
import sys
import re

WORD_RE = re.compile(r"[\w']+")

class WordFreq(MRJob):
    ''' This function returns the top K frequency occurrences of words in a
        provided text.
        The number of words may be greater or less than K, depending if there
            are several words with
        the same number of occurrences in the text, or if there are fewer
            words than K in the text. '''

    FILES = ['stop_words.txt']
    SORT_VALUES = True

    def configure_args(self):
        super(WordFreq, self).configure_args()
        self.add_passthru_arg(
            '--limit',
            metavar='K',
            dest='K',
            type=int,
            default=10,
            help='Input stop words text file ')
        self.add_file_arg(
            '--stop-words',
            metavar='STOP_WORDS_FILE',
            dest='stop_words',
            type=str,
            default='stop_words.txt',
            help='Number of highest occurances to return')

    def steps(self):
        return [
            MRStep(
                mapper_init=self.mapper_init,
                mapper=self.mapper_get_words,
                combiner=self.combiner,
                reducer=self.reducer,
            ),
            MRStep(
                reducer=self.reducer_sort_counts,
            )
        ]

    def mapper_init(self):
        with open(self.options.stop_words) as f:
            self.stop_words = set(line.strip() for line in f)

    def mapper_get_words(self, _, line):
        for word in WORD_RE.findall(line):
            word = word.lower()
            if word not in self.stop_words:
                yield (word.lower(), 1)

    def combiner(self, word, counts):
        yield (word, sum(counts))

    def reducer(self, key, values):
        yield None, (sum(values), key)

    def reducer_sort_counts(self, _, word_counts):
        K = self.options.K
```

```
            i = 0
            last_freq = 0
            current_freq = 0
            for count, key in sorted(word_counts, reverse=True):
                if (i==K):
                    current_freq = int(count)
                if (i < K):
                    last_freq = current_freq
                    current_freq = int(count)
                    if (last_freq != current_freq):
                        i+=1
                    yield (count, key)

if __name__ == '__main__':
    start_time = datetime.now()
    WordFreq.run()
    end_time = datetime.now()
    elapsed_time = (end_time - start_time)*1000
    sys.stderr.write("Total Seconds WordFreq.py: ({0}) microseconds\n".format(elapsed_time.total_seconds()))
```

Code B3: WordInverseIndex.py

```
from mrjob.job import MRJob
from mrjob.step import MRStep
from datetime import datetime
import itertools
import sys
import re

WORD_RE = re.compile(r"[\w']+")

class WordInverseIndex(MRJob):
    ''' Returns the inverse index of the words in the first K lines in the
        input text'''

    FILES = ['stop_words.txt']
    SORT_VALUES = True

    def configure_args(self):
        super(WordInverseIndex, self).configure_args()
        self.add_file_arg(    '--stop-words',
                                    metavar='STOP_WORDS_FILE',
                                    dest='stop_words',
                                    type=str,
                                    default='stop_words.txt',
                                    help='Input stop words text file ')
        self.add_passthru_arg( '--limit',
                                    metavar='K',
                                    dest='K',
                                    type=int,
                                    default=-1,
                                    help='Number of lines of the text to process')
        self.add_passthru_arg( '--index-limit',
                                    metavar='L',
                                    dest='L',
                                    type=int,
                                    default=-1,
                                    help='Number of indexed lines per word to return')

    def steps(self):
        return [
            MRStep(
                mapper_init=self.mapper_init,
                mapper_raw=self.mapper_raw,
                reducer=self.reducer,
            ),
            MRStep(
                reducer=self.reducer_sort_counts,
            )
        ]
```

```
    def mapper_init(self):
        with open(self.options.stop_words) as f:
            self.stop_words = set(line.strip() for line in f)

    def mapper_raw(self, path, _):
            with open(path, 'r') as f:
                for num, line in enumerate(f,1):
                    if (self.options.K!=-1) & (num >= self.options.K): break
                    for word in WORD_RE.findall(line):
                        word = word.lower()
                        if word not in self.stop_words:
                            yield (word.lower(), num) # optionally, can pass [num, line.
                                rstrip()]

    def reducer(self, key, values):
        if (self.options.L==-1):
            lines_list = list(values)
        else:
            lines_list = list(itertools.islice(values, self.options.L)) #
        yield None,(key, ', '.join( str(line) for line in lines_list ))

    def reducer_sort_counts(self, _, values):
        for key, lines in sorted(values):
            yield (key, lines)

if __name__ == '__main__':
    start_time = datetime.now()
    WordInverseIndex.run()
    end_time = datetime.now()
    elapsed_time = (end_time - start_time)*1000
    sys.stderr.write("Total Seconds WordInverseIndex.py: ({0}) microseconds\n".format(elapsed_time.
        total_seconds()))
```