Course Assignment

Master of

Applied Computer Science

Department of Technology

Assignment title	Exploring Science Fiction and Fantasy
Course code	MA140
Course name	Big Data
Due date	24th Feb 2020

Declaration:

Through the submission of this assignment, we hereby declare that this report is the result of our own work, and that all sources have been properly cited to throughout the text.

Names of students	-
Student ID numbers	865317, 866354

Exploring Science Fiction & Fantasy Q/A

ABSTRACT

In this assignment, we parsed through XML-based datasets using a Hadoop- and Pig Apache. We learned how to write MapReduce jobs and Pig scripts, and how effective it is on our dataset based on XML files from Science Fiction and Fantasy Q/A stack exchange.

I Introduction

This report is of a structure where each task is independent and separately reported, frequently referring to each other due to their similarity. Each contains our assumptions about what the task is asking for, implementation, and any notes/reflections if necessary, opting to add notes/reflection more than necessary rather than less.

We chose to work together because we both have python background, choosing python as the main programming language. We explored a Science Fiction & Fantasy Q&A StackExchange dataset based in XML-files as given from archive.org/download/stackexchange/scifi.stackexchange.com.7z (accessed 17.02.2020) through Hadoop and Pig.

Hadoop is used to store and process big amounts of data (Bit Data), it has enormous processing power potential and has the ability to handle virtually any number of tasks and jobs. Hadoop provides a framework that allows users to write and test distributed systems and does not rely on hardware to provide fault tolerance, making it less apt for our small-scale applications, yet we use it for the experience. Pig is a MapReduce abstraction, working on Hadoop methods, that we use in some tasks due to its efficiency. We have elected to use both Pig-scripts and pure MapReduce for very similar tasks (see 2c and 2d) to both show our ability to employ both methods and get experience with the two methods.

II Main functions

The datasets consist only XML files which, being of a document-oriented database, is structured into attributes attributed to each element of the file. Being oriented towards use in HTML, i.e. a website, the attributes contain ASCII characters, punctuation, numbers and HTML tags that we don't want to be there. For this, we designed the function cleanBody, meant to clean the text extracted from the XML-documents. Additionally, a function to parse the XML files was required, for which we created xmlparser. Lastly, we created a general function that would perform the duties of putting out the results of the mapper function; mapper_core. This section explains their implementations, details the modules imported and describes the bash-script used to run our python-scripts:

A *cleanBody*

The cleanBody function formats strings to be parseable by python interpreters. The function removes case sensitivity, ASCII characters, HTML formatting, and treats anything separated by blanked space or / as separated words. This simple implementation of text formatting will affect the results of some tasks. For instance, the name "Jens-Petter" will be interpreted as the word "jenspetter" and the file path /documents/folder/file.extension will be interpreted as the words "documents", "folder" and "fileextension". The function takes a string as input, formats it and outputs a list of all the words split by the string.split(" ") function. It uses the sub-function of the open source re module to remove HTML-tags and ASCII function that is inherent starting with in Python 3 to convert all characters to parse-able ASCII text. The symbols in ignore_char, which are characters we are not considering is defined by the builtin string functions string.punctuation and string.digits.

```
def cleanBody(body):
  body = body.lower()
  body = ascii(body)
  body = sub("<.+?>","",body)
  body = body.replace("/", " ")
  body = body.strip()

  for i in ignore_char:
      body = body.replace(i, "")

  body = body.split(" ")

  return body
```

Listing 1. cleanBody function

B *Mapper Core*

mapper_core is the core of the mapper function; it prints out the relevant data in a format parseable by Hadoop. It functions through three modes, as determined by the parameter mode: "single", "double", and "triple".

- Single: Assumes input "words" is a list of words to print. Prints word in words as (word, 1). Ignores empty strings and spaces.
- Double: Assumes input "words" is comprised of two nested lists to print. Prints word, count in words as (word, count). Ignores empty strings and spaces.
- Triple: Assumes input "words" is comprised of three nested lists to print. Prints id, score, title in words as (id, score, title). Does not ignore empty strings and spaces.

```
def mapper_core(words, mode="single"):
  if mode == "single":
    for word in words:
      if word not in ["", " "]:
       print("%s %s" %(word,1)) #Emit the word

  elif mode == "double":
    in1, in2 = words
    for word, count in zip(in1,in2):
      if word not in ["", " "]:
        print("%s %s" %(word,count)) #emit the
    words

  elif mode == "triple":
    in1, in2, in3 = words
    for id, score, title in zip(in1,in2,in3):
      print("%s %s %s" %(id,score, title)) #emit
    the words
```

Listing 2. mapper_core function

C *XML Parser*

The XML Parser function parses an XML using the open-source xml.etree.ElementTree. It uses sys.stdin as input for normal, Hadoop-based use, but for debugging purposes it also contains a clause that lets it accept a string (see the isinstance(infile, str) check) as a path to the location of an XML-file.
Parsing an entire XML-file takes up a significant amount of memory, so we separated it out as an imported function. This makes it simple to exchange it with another method that could more easily deal with larger files.

```
def xmlparser(infile):
  if not isinstance(infile, str):
    infile = infile.detach()
  mytree = ET.parse(infile)
  myroot = mytree.getroot()
  return myroot
```

Listing 3. xmlparser function

D *Import sys*

The sys module is an open-source, built in python function that allows us to add to the module search path with sys.path.append('../../'), enabling finding ProjectFunctions in a parallel folder, and read in the input for our mapper function and reduce function using sys.stdin.

```
import sys
sys.path.append('../../') #allows access functions
     in parallel folder
import ProjectFunctions.functions as proj
```

Listing 4. Import for sys

E *Import xml.etree.ElementTree*

ElementTree is a part of the XML open-source library. XML is an inherently hierarchical data format, being document-oriented, and is naturally represented using a tree-like structure. By using the library ElementTree will represent the XML document as a tree, and elements of the tree is represented as single nodes.

```
import xml.etree.ElementTree as ET
```

Listing 5. Import for ElementTree

F *Import re.sub*

re.sub is an open-source module we use in cleanBody to remove HTML-tags by removing anything that appears between two chevrons.

```
from re import sub
```

Listing 6. Import for re.sub

G *Bash-script*

As an example of files being run, this bash-script that was used while testing out scripts is provided. It was used on a Windows PC, meaning scripts had to be converted to Unix code. It automatically names, cleans and saves outputs locally. Below is the base bash-script used for the tasks. The three variables taskNumber, taskName, and sourceFile are used to operate it. The bash-scripts for the tasks containing Pig-scripts have an additional clause for running the Pig-script.

```
# Assumptions:
# 1. the dataset in the cluster is located under "
    Dataset/"
# 2. the current directory contains files mapper.
    py and reducer.py for mapper and reducer code
    respectively

#Simplify task change further
taskNumber=
taskName=
sourceFile=

#Simplify task change
mapperfile=$taskNumber$taskName.py
reducerfile="$taskNumber"Reducer.py
outfile=output$taskNumber

#Automatic removal
hadoop fs -rm -r $outfile

#For Windows-compatibility
apt-get install dos2unix
dos2unix $mapperfile
dos2unix $reducerfile

#Main MapReduce function call
hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/
    hadoop-streaming-2.7.3.jar \
-files $mapperfile,$reducerfile \
-mapper $mapperfile \
-reducer $reducerfile \
-input Dataset/$sourceFile.xml \
-output $outfile

#Read file and save it locally
hadoop fs -cat $outfile/*
hadoop fs -copyToLocal $outfile
```

```
#Automatic cleanup
hadoop fs -rm -r $outfile
```

Bash-script for a windows-based environment

H *Data trimming*

We use the environment Docker as a virtual data cluster. Files that are bigger than 128MB are not readable by the module we use to parse XMLs within the Docker environment (whether this is due to this being the standard size of a data block we were not able to tell). Between the choice of running our software outside Docker and trimming the files for data, we elected to trim the relevant file for this case: Posts.xml. We deleted lines 74 699 through 171 247 and used the remaining data as a basis for our analysis.

III Task 1 *Warmup*

A *WordCount*

Assumption: Count the words in body of questions, identified by PostTypeID="1", in *Posts.xml*. The result should be a list of each word and how many times it occurs in the bodies of questions.

Implementation: The mapper uses the xmlparser function to parse an XML-file, iterates through all rows in the parsed file, cleans text using the cleanBody function and prints output for the reducer script by using the mapper_core function. The reducer receives the mapper output, parses it and iterates through the parsed lines. A counter variable is used to track repeated words, outputting the word with the counter when the word stops repeating.

Notes/Reflection: Here the choices made building the cleanBody function affects the output, interpreting some things as words that are not. See line 2 in 1a_output for example; what is possibly an "accordingly" followed by three line skips and then an "in" is interpreted as "accordinglynnnin" and is counted as a word.

```
accordinglyn 2
accordinglynnnin 1
accordn 1
accordnits 1
accords 6
accosted 1
account 107
accountable 3
accountant 2
accounted 9
accounting 8
```

1a_output

B *Unique words*

Assumption: Write a MapReduce job which outputs unique words in the titles of questions, identified by PostTypeID="1", in *Posts.xml*. The result should be a list of each word as it appears in the titles of questions.

Implementation: This MapReduce job functions as 1a WordCount does, except it does not count the words and it looks through the titles instead of the bodies.

Notes/Reflection: Here again the choices made building the cleanBody function affects the output, see "althalus".

```
altered
altering
alternate
alternating
alternative
alternatively
alternatives
alters
althalus
although
altmode
```

1b_output

C *MoreThan10*

Assumption: Write a simple python script to check the amount of words in titles of posts in *Posts.xml*. The result should be a count of how many titles have more than 10 words.

Implementation: The mapper functions as the mapper in 1a WordCount, except it keeps a counter for the titles of more than 10 words and it outputs the final counter instead of using mapper_core. The reducer receives the mapper outputs and sums them up.

Notes/Reflection: This MapReduce job is simpler, the reducer is unnecessary outside an environment with clusters.

```
10264
```

1c_output

D *Stopwords*

Assumption: Write a simple python script based on task 1a to exclude stopwords: from body of questions, identified by PostTypeID="1", in *Posts.xml*. The output should be a word count excluding any stop words.

Implementation: This MapReduce job functions as 1a WordCount does, except it has a section for the iterative removal of stopwords, as defined in StopWords.txt.

Notes/Reflection: Given how we built cleanBody, we chose

to remove the apostrophes (') from the stopwords, turning words like "we're" to "were".

```
amazo 5
amazon 1
amazonian 1
amazons 1
amazos 1
ambassador 7
ambassadorclass 1
amber 7
ambush 3
ambushed 1
```

1d_output

E *Pig top 10*

Assumption: Write a pig script to select the top 10 listed words after removing the stopwords from *Posts.xml*. The output should be a list of the top 10 listed words and the corresponding occurrence count.

Implementation: Given this task being based on 1d StopWords, the mapper and reducer are the same for both tasks. Our pig loaded the output from the MapReduce job, ordered it by count, limited it to 10 elements and stored it into a text file: 1ePig_Output.txt.

Notes/Reflection: We chose for this pig script to load its dataset from an HDFS cluster as opposed to a local storage location.

```
the 2489
story 1024
a 952
book 863
short 565
movie 469
time 450
star 443
of 420
series 339
```

1ePig_output

F *Tags*

Assumption: Write a MapReduce job to create a dictionary over unique tags in *Posts.xml*. The result should be a list of unique tags.

Implementation: This MapReduce job functions similarly to 1a WordCount, except it has to be aware of whether the post has a tag at all or not, being implemented when tags are extracted from the row.

Notes/Reflection: The tags are separated by chevrons and would be fused by cleanBody, so we elected to replace them with spaces. Some of the posts lacked any tag attribute, so we elected to pass over those.

```
elantris
elizabethmoon
elshaddai
elves
elysium
emberverse
empire
empirefromtheashes
endersgame
endor
enemymine
```

1f_output

IV Task 2 *Discover*

A *Counting*

Assumption: Write a MapReduce job to count the total unique users there are in *Users.xml*. The result should be a count of how many unique users there are.

Implementation: This MapReduce job functions similarly to 1c MoreThan10, except it counts all unique users instead of titles with more than 10 words.

Notes/Reflection: We interpreted "unique user" as a user with a unique Id attribute.

```
92728
```

2a_output

B *Unique users*

Assumption: Write a MapReduce job based on 2a Counting to list the unique users in *Users.xml*. The result should be a list containing unique users in the dataset.

Implementation: This MapReduce job functions as 2a Counting does, except it outputs all unique users instead of a count of them

Notes/Reflection: Here the implementation of cleanBody is modified with a join function as cleanBody returns a list where we want a string. Since usernames can contain spaces, we elected to instead use | as a separator in hadoop.

```
100509 dawnfire
100510 josh
100511 gosala nihal
100512 da worior
100513 steveisworthyletsbereal
100514 md tamzid mazumder
100515 tom dufall
100516 matthew gutman
100517 glenn willen
100518 u ub
100519 darius mccoy
```

2b_output

C *Top users*

Assumption: Write a pig script or MapReduce job to find the top 10 users based on the Reputation attribute in *Users.xml*. The result should be a list of the 10 users based on the highest reputation value.

Implementation: This mapper job is based on 2b Unique users, except instead of putting out a name and its corresponding id, it puts out a name and its corresponding reputation attribute. Here, reputation is given to mapper_core as a one-element list (mapper_core called with arguments (name, [reputation])) due to its looping through its inputs. The reducer is the same as the reducer of 2b Unique users. The pig script functions the same way as in 1e Pig Top 10

Notes/Reflection: We elected here to use Pig-script to order the MapReduce output to get experience.

```
valorum 458690
user 458398
dvkonahchto 314049
thaddeus 201037
jason 170489
slytherincess 153478
the 130440
phantom 126901
jack 121302
fuzzyboots 120456
```

2c_output

D *Top questions*

Assumption: Write a pig script or MapReduce job to find top 10 title questions, identified by PostTypeID="1", in *Posts.xml* based on the Score attribute. The result should be a list containing the top 10 questions, showing id, question, and score.

Implementation: This mapper job functions similarly to 2c Top users except it puts out 3 parsed attributes using the "triple"-mode of mapper_core. The reducer performs the reduction and ordering, finding the 10 highest score values. It does this by inserting every input into a list instead of printing it, using pythons standard function string.sort to order it. It then iterates through the 10 highest values to give an output.

Notes/Reflection: The reducer could potentially be improved by only keeping 10 elements in the list at once, changing them out as higher scores were encountered.

```
48180,  can king cold transform, 450
59582,  which pacific rim jaeger had the most
    overall kills, 202
68005,  can harry potters invisibility cloak hide
    inanimate objects nonsentient beings, 202
61010,  what is the status of science in the seven
     kingdoms, 189
55044,  did dumbledore plan to have harry possess
    all  hallows, 174
35410,  how essential is jarvis to the iron man
    war machine armors, 165
64415,  s scifi childrens book two children and a
    good magician teach sea creatures to fight an
    evil wizard and his dwarves, 160
76856,  in interstellar do people age while in
    cryosleep, 140
54865,  what was the causeway span leading to the
    bifrost bridge made out of, 131
41339,  what happened to the espheni ships and the
     technology they brought with them, 130
```

2d_output

E *Favorite questions*

Assumption: Write a pig script or MapReduce job to find top 10 title questions, identified by PostTypeID="1", in *Posts.xml* based on FavouriteCount attribute. The result should be a list containing the top 10 questions, showing id, question and the favourite count.

Implementation: This MapReduce job is the same as 2d Top questions, except it extracts the FavoriteCount-attribute instead of the score

Notes/Reflection: Here, like 1f Tags, some posts had to be skipped due to their lacking any FavoriteCount-attribute.

```
2335,   why are slaaneshi daemonettes so ugly, 116

1520,   why could young tom riddle do wandless
    magic, 81
48180,  there is always an exterior light shining
    onto the ship in every star trek where does
    this light source come from, 78
4649,   does anyone know the name of this fantasy
    book series from about  to  years ago, 70
2611,   novel about genetically engineered
    children who dont need to sleep, 63
1586,   in what order should i read the robert
    langdon books, 57
2333,   what is the alternate universes fringe
    division logo derived from, 49
4056,   how does time travel really work in the
    terminator universe, 48
7914,   in snowpiercer why did the soldiers with
    hatchets seem not to care too much if they
    survived   got killed, 46
2937,   did anyone make a count of the number of
    times egwene nynaeve elayne got caught, 44
```

2e_output

F *Average answers*

Assumption: Write a MapReduce job that calculates the average number of answers per question. The result should be a calculated average of all AnswerCount attributes.

Implementation: This mapper functions similarly to 1c MoreThan10, except for having two counters, one for score and the other for users parsed, which are averaged at the end. The reducer is the same as the reducer in 2a Counting

Notes/Reflection: We have elected to not limit the amount of decimals.

```
2.451502036659878
```

2f_output

G *Countries*

Assumption: Write a MapReduce job to count the amount of users attributed to a country by the Location attribute in *Users.xml*. The result should be a list of different countries and corresponding users.

Implementation: This mapper functions similarly to 1f tags, except instead of cleaning the tags, it uses "|" as a separator since spaces are part of the location. If Location is separated by a ',' the last part is treated as the country.

Notes/Reflection: Because by our judgement locations are typed in by users, we have elected to separate the country out of Location very simply by using ',' as a separator. Anything after the last ',' is treated as a country. A location with no ',' is treated as a country. Locations like "California, USA", "California, United States", and "US" will be interpreted as "usa", "united states", and "us", respectively.

```
cet  1
channel islands  1
chile  40
china  119
ci  1
cloaked about to kill you  1
co  119
colombia  29
colorado  1
corner madness  1
```

2g_output

H *Names*

Assumption: Write a pig script or MapReduce job to find the 10 most popular names in *Users.xml*. The result should be a list of top 10 common names.

Implementation: This mapper functions similarly to 2g Countries, except it extracts the DisplayName-attribute in place of the Location-attribute. The reducer functions similarly to 2d Top questions, except it parses differently and has different text formatting for the output due to the decrease from 3 to 2 arguments

Notes/Reflection: Here the most common name is User. This is likely due to our choice of removing numbers in cleanBody, combining User1, User2, User3, etc. The task suggested explicitly that we should split up names by spaces, perhaps wanting to separate out last names from first names. Due to the nature of Displaynames usually not being full legal names, we have elected to interpret a "name" as the full name given in DisplayName.

```
user  5243
chris  155
john  149
mike  146
james  139
michael  125
alex  123
david  123
pmar  120
matt  119
```

2h_output

I *Answers*

Assumption: Write a python script to find how many questions, identified by PostTypeID="1", in *Posts.xml* have at least one answer based on attribute AnswerCount. The result should be a count of how many questions have been answered.

Implementation: This mapper functions in the same way as 1c MoreThan10 except it checks whether AnswerCount is larger than or equal to 1 instead of whether the title word length is larger than 10.

```
22680
```

2i_output

V Task 3 *Numbers*

A *Bigram*

Assumption: We chose to write a MapReduce job to find the most common pair of adjacent words in the titles of questions, identified by PostTypeID="1", in *Posts.xml*. A bigram is a combination of two words in a sequence. For instance the sentence "Big data is big" contains the bigrams "Big data", "data is", and "is big". The result should be the most common bigram along with a count of how many times it occurs.

Implementation: This MapReduce is the same as 1a WordCount, except we output each bigram in the body instead of each word. Because of this we he had to forego the mapper_core for a print function inside a loop looping through every bigram. The reducer finds the most common bigram by iteratively comparing each elements second element (the bigrams count) against a variable declared maximum.

Notes/Reflection: In the print-loop, we had to limit the loop to i in range(len(words)-1), since we printed both words[i] and words[i+1], and doing otherwise would result in and out of range error.

```
in the  , 2052
```

3a_output

B *Trigram*

Assumption: Perform the same task as in 3a Bigram, except operate with trigrams (sequences of 3 words as opposed to 2). The result should be the most common trigram.

Implementation: This is exactly the same as 3a Bigram, except we output trigrams instead of bigrams.

Notes/Reflection: The mapper, as in 3a Bigram, limited the print-loop to range(len(words)-2) and printed words[i], words[i+1], and words[i+2].

```
what is the  , 671
```

3b_output

C *Combiner*

Assumption: Perform the same task as in 1a WordCount, except add a combiner before the mapper sends information to the reducer. The result should be the same as in 1a WordCount, but the volume of the data transfer should be smaller.

Implementation The MapReduce function is the same as in 1a WordCount, with a combiner ahead of the print function in the mapper.

Notes/Reflection: Having a combiner in a MapReduce job saves bandwidth and computational strain by decreasing the volume of data sent from the mapper, as shown in 1.

```
abides 1
abidesnnthennnn 1
abiding 3
abigail 7
abigailus 1
abililties 1
abilities 303
abilitiesn 9
abilitiesna 1
abilitiesngandalf 1
```

3c_output

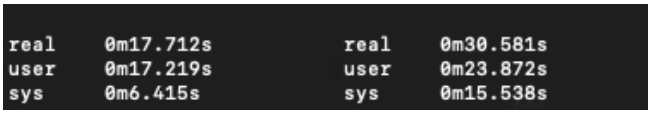

Figure 1. TIming for our two methods, Leftmost shows 3c combiner, rightmost shows 1a WordCount

D *Useless*

Assumption: Write a MapReduce job to find how many times the word "useless" in occurs in the bodies of questions, identified by PostTypeID="1", in *Posts.xml*. The result should be a count of how many times the word useless occurs.

Implementation: This mapper functions similarly to 1c MoreThan10 and 2i Answers, except instead of checking for titles with more words than 10 or whether AnswerCount is larger than or equal to 1, it counts whether the word "useless" occurs in the body.

```
The word useless occurs 44 times
```

3d_output

VI Task 4 *Search engine*

A *Title index*

Assumption: Write MapReduce job to create an index over titles, bodies, and answers of questions in *Posts.xml*. The result should be a simple index that lists posts words appear in by their Id's.

Implementation: This mapper function parses through all rows in posts.xml. It then forks over whether PostTypeId is 1 or 2, i.e. whether it is an answer or question, extracting the relevant id, body, and title from the row. It contains a combiner to speed up the process, removing duplicates. The reducer parses the input and iterates through all the lines. It uses two variables to know what the last word and associated Id was, skipping the word if both are the same, adding the Id to a list if *only* the word is the same, and putting out the word with an associated list of Id's if neither is the same, i.e. it has finished working with the word.

Notes/Reflection: We elected to have a combiner in the mapper although it is not strictly necessary since we are working with small file-sizes.

```
acegarageguy, ,86349
acellerator, ,49370
acen, ,76828
acended, ,45192
acenn, ,76252
acennndimensions, ,3819
acennthe, ,65530
acension, ,77473
acepl, ,1578
```

4a_output

VII Conclusion

Through this assignment, we gained experience in using MapReduce jobs and pig scripts. We parsed XML-documents and extracted specific information we were looking for to be formatted to an output for each task.