Machine Learning - Fall 2023 - Class 20

- admin
  - assignment 9 out
  - Midterm 2 will be posted on Monday
    - Cover everything from gradient descent up through deep learning
      - won't ask you to write mapreduce code
    - open book, open notes

- All hadoop demos can be found in the examples directory

- what does the "map" function do in other languages?

- general mapreduce program pipeline
  1. the input is provided to the mapreduce program
     - think of the input as a giant list of elements
     - elements are ALWAYS some key/value pair
     - however, the default key/value pair is:
       - key = byte offset into the file
       - value = line in a file
  2. each key/value pair is passed to the mapping function
     - the mapping function takes a key/value pair as input
     - does some processing
     - and outputs zero or more key/value pairs as output (not necessarily the same types as input)
  3. all of the output pairs are grouped by the key
     - this results in: key -> value1, value2, value3, ... for all the values associated with that specific key
     - this is still a key value pair
       - the key = key
       - the value = and iterator over values
  4. these new key/value pairs are then passed to the reducer function
     - input is key -> value iterator
     - does some processing (often some sort of aggregation)
     - outputs the final key/value pairs, which should the the answer (or at least answer to the subproblem)

- let's try and write a function that counts the number of word occurrences in a file

- writing a mapreduce program
  - three components:
    1. map function
    2. reduce function
    3. driver

- to write your own program, here's how I recommend doing it
  1. Figure out what your input is
     - in particular, what will the map step get as input for it's key/value
     - the default is just a line in a file
  2. Figure out how to break the program down into a map step and a reduce step
     - The map step will take the input, do some processing and produce a new collection of key/value

pairs
- The reduce step will take the output from the map step as input and then produce another new collection of key/value pairs
- Sometimes, you may have to break the program into multiple map/reduce steps!
- most of the programs we'll look at can be accomplished with just 1-2 map/reduce steps

3. Write pseudo-code for the map and reduce functions
- be very specific about what the key/value input/output types are for the map/reduce step
- think about what processing needs to happen in each function
- ideally, you should keep this processing down to a minimum
- there cannot be shared state between calls to the map function!
- if you find that you need it, you need to rethink how to write the program

4. Write the code

I'll leave this here for context, but we won't be writing any code for this class!

a. Decide whether you want to have a single class (with nested map and reduce classes) or three classes

b. Write your map function
- convert your input types into the appropriate hadoop types (IntWritable, DoubleWritable, Text, ...)

c. Write a basic driver function
- setup the job configuration, in particular
- create a new JobConf item based on the driver class
- set the job name
- set the key/value output types
- Optional: if the key/value output types of the map are *different* than the output types of the reduce stage, set these as well
- set the mapper and reducer classes
- Optional: if you're using a combiner class, set that as well
- set the input and output directories
- Optional: if your program requires additional input, set these as well
- setup code to run the job

d. Debug your map function
- I strongly encourage you to use the NoOpReducer and make sure your map function it printing out what you expect before trying to put the whole thing together
- Run it on some test data and make sure your map function is working

e. Write your reduce function
- convert your input types into the appropriate hadoop types

f. Put it all together and run it!

- general overview: first, let's look at how we can break this down into a map step and a reduce step
- map step for word count?
- input is a line
- two output options
- option 1: word -> # of occurrence in this line
- option 2: word -> 1 for each word in the line
- either of the options is fine, however, most often will choose option 2
- simpler to implement
- you want the map function to be as fast and as simple as possible
- you want to avoid having to declare/create new objects since this takes time
- remember that this map function is going to be called for *every* line in the data
- you want the processing for each call to map to be as consistent as possible

- reduce step
  - the reduce step gets as input the output from the map step with the values aggregated into an iterator per key
    - in our case: word -> 1, 1, 1, 1, 1, 1, 1, 1 (an iterator with a bunch of 1s)
  - all we need to do is some these up and output a pair of word -> sum

- input/output types
  - before you can actually write your program you need to figure out what types of the input and output should be
    - the input and output are always key/value pairs

  - mapreduce types
    - The main types we'll use are:
      - Text
      - IntWritable
      - LongWritable
      - DoubleWritable
      - BooleanWritable
    - Why do they have their own built-in types (instead of say, Integer, Double, Long, ...)?
      - They're mutable!
      - In MapReduce programs we try hard to minimize the number of objects created

- types for word count
  - map
    - the default input to a mapper is
      - key = number (the specific type is LongWritable)
      - value = line (the specific type is Text)
    - the output types will depend on what computation you're doing
      - for word count?
        - key = Text
        - values = IntWritable (could actually use almost anything here)
  - reduce
    - the input to reduce will always be the output from the map function, specifically
      - input key type = map output key type
      - input value type = Iterator<map output value type>
    - the output to reduce will depend on the task (but the key is often the same as the input key)
      - for word count?
        - key = Text (the word)
        - value = IntWritable

- look at WordCount code
  - both the map and reduce function MUST be written in their own classes
    - the map function should be in a class that implements Mapper
      - three methods to implement: map, close and configure
      - often we'll extend MapReduceBase which has default methods for close and configure
    - the reduce function should be in a class that implements Reducer
      - three methods to implement: reduce, close and configure
      - often we'll extend MapReduceBase again
    - two options for generating these classes (we'll see both in examples for this class)

- stand alone classes
- as static classes inside another class
- for simple approaches (and approaches where we don't need any state) this is a nice approach

- WordCountMapper class
- when implementing the Mapper interface, we need to supply the types for the input/output pairs
- then we just have to implement the map function
- takes 4 parameters
- first is the input key
- second is the output key
- third is the collector, which is where we'll put all of our input/output pairs that we're *outputting* (to the reduce phase)
- fourth is a reporter, which we'll talk about later
- functionality:
- split up the line into words
- for each word, add an output pair word -> 1
- why do we have the two instance variables?

- WordCountReducer class
- when implementing the Reducer interface, we need to supply the types for the input/output pairs
- then we just have to implement the reduce function
- takes 4 parameters
- first is the input key (it will be the same type as the output key type from the map function)
- second is an iterator over values (the type of the iterator will be the output value type from the map function)
- third is the collector, which is where we'll put all of our input/output pairs that we're *outputting* (for the final output)
- fourth is a reporter, which we'll talk about later
- functionality
- the iterator should have all of the word occurrence counts (in our case, a lot of 1s)
- iterate over this and keep track of the sum

- run
- To run a mapreduce job you need to tell it a number of things, e.g. what the output types are and what the map and reduce classes are
- This is specified in the JobConf configuration file
- look at the run method as a good example of how to set this up
- Based on this configuration, you then instantiate a JobClient and actually run the job by calling .runJob

- main
- There still needs to be an entry into the Java program, so we need a main method somewhere
- It doesn't have to be in the same class as the "run" method, but we'll often put it there for convenience

- Note about performance
- The map and reduce function will get called many, many times (e.g. the map function for each line in the file)
- Because of this, even small changes in efficiency of these functions can drastically impact the overall run-time

- A few observations about the WordCount code regarding efficiency:
  - Avoid instantiating variables wherever possible (you see this in both the map and reduce methods)
    - use the "set" methods on a single instance variable
    - the collector copies the data so it's fine to reuse a variable
  - Avoid data structures
    - This is why we prefer outputting 1 for a word rather than the word count per line
  - Use static final constants when you can

- A few more details on how the MapReduce framework works:
  - http://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/
    - (this link also has some more nice MapReduce examples)

- Look at the NoOpReducer

- Write grep (search of occurrences of text in a file)
  - Key highlight for this example: how we can pass some data that is shared across all map/reduce calls
  - Map:
    - Input
      - key: LongWritable
      - value: Text
    - Output
      - key: LongWritable (byte offset)
      - value: Text (line with an occurrence of the word)
    - Check if the word being searched for is in the input text. If so, output key/value pair.
  - Reduce: NoOpReducer... we're already done!

  - How do we get the word to each of the map method calls?
    - Use the configure method and an attribute we set in the JobConf

- Look at Grep code

- Finding location with the largest temperature given:
location temp1, temp2, temp3, ...

- Find out how many places had each temperature

- inverted index