BigML Assignment: Scalable Linear Classification

Due Wednesday, May 1, 2013 via Blackboard

Out April 17, 2013

1 Important Note

As usual, you are expected to use Java for this assignment. It could take hours to run your experiments. Start early.

Bin Zhao (binzhao@cs.cmu.edu) is the contact TA for this assignment. Please post clarification questions to the Google Group:
`machine-learning-with-large-datasets-10-605-in-spring-2013`

2 Scalable Linear Classification

In this assignment, you'll implement a scalable classifier for a linear classifier, like the one that is learned by Naive Bayes. We will assume that there are a small number of classes, but that both the test data and classifier being used (the event counts) is too large to fit in memory. The algorithm was discussed in class on 1/31/2013.

The test data you will use has been re-formatted for Hadoop, so that the key is a document id, and the value is a tab-separated pair of strings: the class, and the contents of the document. The document id's are needed for this assignment.

The data are on AFS: `/afs/cs.cmu.edu/project/bigML/dbpedia_id`, and they are also in this public s3 bucket: `s3://bigml-s2013`

The data set contains 12 files in total, with the same name as in Assignments 2, with one small change: The data has format:

```
<doc id>\t<label>\t<text>
```

2.1 Infrastructure

You should implement a subroutine converts a (small) hashtable to a string of the form $k_1 = v_1, k_2 = v_2, \ldots$, and another which converts such a string to a hashtable.

We also need to implement the "request-and-answer" pattern for Hadoop. The way we did this with streaming is to to prepend a special string to all "messages", so they are seen last by a reducer. One way to do this in Hadoop is to use keys constructed using the `TextPair` class defined here:

`https://github.com/tomwhite/hadoop-book/blob/3e/ch04/src/main/java/TextPair.java`

A `TextPair` is sorted by the first element, and then the second; so if you want to ensure that a data record with key w is seen before the messages to that data record by a reducer, it's enough to use something like `new(TextPair(`w`,''data''))` for the data-record keys, and something like `new(TextPair(`w`,"msg"))` for the message keys. (Of course "data" and "msg" could be "0" or "1", or "a" and "z"—as long as the string for data comes before the string for messages.)

You also need to tell Hadoop to send all the messages to w to the mapper that has the data record for w. To do this, you need to change two parameters of the job: you need to use `job.setPartitionerClass(...)` and `job.setGroupingComparatorClass(...)`. There's an example of how to do this here:

`https://github.com/tomwhite/hadoop-book/blob/3e/ch08/src/main/java/JoinRecordWithStationName.java`

2.2 Phase 1: Build the word statistics table.

The goal is to convert the original event counters, which will be a key/value table that looks like this:

...	
W=google,Y=sports	321
W=google,Y=worldNews	43412
...	
W=hockey,Y=sports	9245
W=hockey,Y=worldNews	23
W=hockey,Y=...	...
W=...	
...	

To something that looks like this:

...	
google	sports=321,worldNews=43412,...
hockey	sports=9245,worldNews=23,...
...	

You also want to extract the "global" event counts, which are of the form "W=*,Y=n" and "Y=*".

Then you need two mappers, as defined below.

- A mapper with an null (identity) reducer, which outputs only the global counts. These will be tiny—you can grab them with the Hadoop file system operations, and pass them in as command-line parameters to your classifier.
- A mapper scans for non-global counts—of the form $W = w, Y = y$—and outputs new key-value pairs with key w and value $Y = n$. For instance, the pair ("W=hockey,Y=sports","9245") would be converted to ("hockey","sports=9245"). This would be coupled with a reducer that scans through all the values "$y_i = n_i$" associated with a data key for w, and appends them, separated by commas, to construct a new value, which encodes all the event counters for w. We'll call the output of this mapper/reducer pair the *word statistics table* below.

  The data key for w should be a `TextPair`, of course, e.g., `new TextPair("hockey","data")`.

2.3 Phase 2: Use the "request and answer" pattern to collect word statistics

- Step 2A. Write a mapper that converts the test documents to a "document table", by converting each key-value pair id, val to a pair with the same value, but key `new TextPair(`id`,"data")`. (The reducer is an identity reducer).
- Step 2B. Construct messages that request word statistics for the test documents. Write a mapper that loops through the documents and outputs a set of messages as follows: when

called on the document with id *id*, look at each word $w_1, w_2, \ldots$ in that document and produce for each w_i a message with key `TextPair(`w_i`,''msg'')` and value *id*. (Again, use an identity reducer.)

- Step 2C. Answer the word-statistic messages. Use an identity mapper to merge together the word statistics table *and* the messages produced in step 2B, and couple this with a reducer that answers those messages. Specifically, since `GroupingComparatorClass` and `PartitionerClass` have been set to group together the data and message keys for every word w, and since the data key comes first, the reducer's first value will be from the data record for w from the word statistics table—-the hashtable of event counters for that word. The reducer stores this in memory, and then cycles through all the messages to w. The value of each of these messages is the *id* of some document that requested those word statistics. To answer the message, the reducer will output a new message—ie., a key-value pair with a key `TextPair(`*id*`,''msg'')` and value which is w, plus the event-counter hashtable for w.

  The input to the reducer used in 2C should look like this:

...	
hockey,data	sports=9245,worldNews=23,...
hockey,msg	doc37
hockey,msg	doc146
hockey,msg	doc219
...	

  Combining two input tables in this way is often called a *reduce side join* in Hadoop.

- Step 2D. Classify the documents. Use an identity mapper to merge together the document table produced in step 2A, and word statistic messages produced in step 2C. This is sent to a reducer similar to that used in 2C. For every document *id* key, the first value seen by the reducer is the data for that document (which includes the true classes and the words in the document.) Subsequent values are event-counts for each word in the document. The reducer accumulates all this information in memory (since it's only as large as the document) and then outputs the final classification.

  The input to the reducer in step 2D should look like this:

...		...
doc37,data	sports	"In last night's hockey game, the Penguins...."
doc37,msg	in	sports=454545,worldNews=23748,...
doc37,msg	last	sports=64845,worldNews=4584,...
...		...
doc37,msg	hockey	sports=9245,worldNews=23,...
...		...
doc38,data	worldNews	"Events in North Korea have ..."
doc38,msg	events	...
...		...

  The final output should be in the same format as in the Hadoop naive Bayes assignment.

3 Deliverables

Submit a compressed archive (zip, tar, etc) of your code. Please also include the controller and syslog files **from AWS** for each of the mapreduce jobs in both of your pipelines. The controller and syslog files can be downloaded from the AWS console. Simply go to the Elastic Mapreduce tab. Click your job in the list, and then the debug button, the syslog link, and save the resulting file. Please include a pdf document with answers to the questions below.

In addition to your code and a makefile, please include a pdf document with answers to these questions:

1. How would you need to change the algorithm if the classifier was a logistic regression classifier produced by SGD instead of Naive Bayes?

2. The current approach assumes that documents are small enough to fit in memory. One way around this limitation would be to replace the document table with a table where there keys are pairs of id, j where id is a document id and j is the position of a word.

   Outline the changes you would need to perform classification on that sort of representation.

3. It is also possible to parallelize the classification process by splitting the word-statistic table into several small parts, computing a partial classification score with respect to each part of word-statistic table, and then combining the scores.

   Outline an algorithm you could use in this case (not necessarily using map-reduce).

4 Marking breakdown

- Code correctness [**70 points**].
- Question 1 [**5 points**]
- Question 2 [**5 points**]
- Question 3 [**10 points**]
- Controller/Syslog files [**10 points**]