3/10/2017

Boris Kogan and Yuval Lieberman

**Final project: GloVe implementation in Spark**

**Attached files**

1. BuildMatrix.scala – source code for reading text files and building co-occurrence matrix
   1. BuildMatrix.log – sample run log
2. BuildWordVectors.scala – source code for reading co-occurrence matrix and building word vectors
   1. BuildWordVectors.log: – sample run log
3. GloveGradient.scala – helper class used in BuildWordVectors
4. ReadWordVectors.scala – source code for evaluating the model
   1. output.txt – sample output
5. pom.xml – maven project file

**Implementation**

The implementation is divided into 3 parts:

1. Build a co-occurrence matrix out of a set of input text files:
   1. all\_words – RDD[String] containing input text as Strings of one word, transform all words to lower case
   2. wordKeys – RDD[(String,Int)] – run on all\_words, count each word, filter out words below VOCAB\_MIN\_COUNT, filter out words with one letter only, filter out words without at least two consecutive letters (a-z characters). Each resulting word is assigned a unique id.

The RDD is then converted to a map and broadcasted as wordKeysMap

* 1. counts – RDD[((Int,Int),Int)] – run on all\_words, in a sliding window of WINDOW\_SIZE size.

On each window, create a pair of words from combination of the first word and each of the following words. Other combinations will be created on the next sliding windows.

Filter out pairs of the same word.

Filter out pairs where one of the words does not exist in the wordKeysMap

Organize each item a: ((larger word id, smaller word id) , 1)

Reduce the RDD by the word id’s key, to sum the word pair counts.

* 1. Save the RDD of the word keys and counts

1. Build word vectors out of the co-occurrence matrix. This program runs Stochastic Gradient Descent in MAX\_ITER iterations. The parallelization was modeled in a similar way to MLLIB SGD.
   1. Read the RDD of the word keys and counts that were saved in the previous step
   2. Setting initialWeights – Array[Vector]. An array where each element is a word id, so the size in max word id +1. Each element is a vector in the size of VECTOR\_SIZE. The vectors are initialized either to a random value, or to vectors that were saved in a previous run of the program.
   3. Setting initialBiases – Vector, where each element is a bias for a specific word, so the size of the vector is max word id + 1. The vector is initialized either to a random value, or to the vector that was saved in the previous run of the program.
   4. In each iteration, out of MAX\_ITER:
      * Broadcast weights and biases
      * Set gradientSum, biasVector, batchCounts to a single minibatch run.
        1. gradientSum is sums of weight gradients
        2. biasVector is sum of bias gradient
        3. batchCounts indicates how many gradients were summed for each word id
      * Sample counts using miniBatchFraction and a changing random seed
      * Aggregate the sampled counts in a multi-level tree (treeAggregate)
      * Start with all zero values of gradientSum, biasVector, batchCounts
      * Compute the gradient of a single point and update gradientSum, biasVector, batchCounts. The gradient calculation in done using the compute function in the GloveGradient.scala class. It receives as an input the variables gradientSum, biasVector, batchCounts, the word ids, their count and the broadcasted weight and biases.
      * Combine gradientSum, biasVector, batchCounts by summing up the values. The combination is done using the aggregate function in the GloveGradient.scala class.
      * Update the weights and biases according to the summed gradients and stepSize divided by the square root of the iteration number to allow of decreasing updates. The update is done using the update function in the GloveGradient.scala class.
   5. The resulting weights and biases are saved to the disk. This allows reading them for additional optimization or for evaluating words.
2. Evaluate the word vectors. This step reads word vectors resulting from the optimization done in the second program, and evaluates several test cases.

The test cases are built as 2 word pairs, where the first 3 words are the input, and the 4th word is the expected output. The cosine distance is measured between the 1st and 2nd word. The program scans the word vectors to find to top 10 words with the most similar cosine distance to the 3rd word. In addition, the 4th word is ranked according to the cosine similarity difference.

**Evaluation**

The implementation was tested on two datasets.

Dataset 1: COCA magazine corpus. Size 0.5GB

Dataset 2: the full COCA corpus. Size 2.5GB

All tests were done on a standard PC in local spark mode.

The run time of the first program of building the co-occurrence matrix is detailed in the table below. The increase in the run time was lower than the increase in the corpus input, which can indicate that the implementation scales well.

|  |  |  |  |
| --- | --- | --- | --- |
|  | COCA Magazine | COCA full | Increase Factor |
| Size | 0.5GB | 2.5GB | x5 |
| Vocabulary after filtering | 129,506  5.7MB | 280,913  13MB | x2.2 |
| Co-occurrence observations (matrix non-empty items) | 83,773,231  4.4GB | 210,964,245  11GB | x2.5 |
| Run time (seconds) | 1,886 | 7,499 | x4 |

The second program was tested on these two cases with the following results. Both test runs were configured to 15 iterations of 0.3 fraction mini batches.

The x3 increase in run-time compared to x2.5 increase in co-occurrence observations can indicate a linear increase with some overhead due to the increased number of partitions.

|  |  |  |  |
| --- | --- | --- | --- |
|  | COCA Magazine | COCA full | Increase Factor |
| Vocabulary after filtering | 129,506  5.7MB | 280,913  13MB | x2.2 |
| Co-occurrence observations (matrix non-empty items) | 83,773,231  4.4GB | 210,964,245  11GB | x2.5 |
| Total cost on initial random weights | 884,446,465 | 2,469,708,904 |  |
| Cost per word | 6,829 | 8,792 |  |
| Total cost on optimized weights | 11,368,061 | 20,664,199 |  |
| Cost per word | 88 | 74 |  |
| Run time (seconds) | 4,630 | 13,892 | x3 |

Optimization of the full COCA dataset was done 3 times, resulting in total of 45 SGD iterations with 0.3 mini batch fraction on each iteration. The optimization results are presented in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
|  | First run (1-15 iterations) | Second run (iteration 16-30) | Third run (iteration 31 to 45) |
| Total cost on initial weights | 2,469,708,904 | 20,664,199 | 17,005,714 |
| Cost per word | 8,792 | 74 | 61 |
| Total cost on optimized weights | 20,664,199 | 17,005,714 | 15,878,615 |
| Cost per word | 74 | 61 | 57 |
| Run time (seconds) | 13,892 | 14,127 | 14,320 |

The third program evaluates the optimized vectors on 4 test cases. For each case, using the vectors of each of the 3 runs, the 4th word was ranked according to the similarity of the cosine distance from the 3rd word – to the cosine similarity between the 1st and 2nd words (lower is better). Results are presented in the table below. The results indicate that the optimization process brings words closer to their meaning in a consistent manner.

|  |  |  |  |
| --- | --- | --- | --- |
|  | First run (1-15 iterations) | Second run (iteration 16-30) | Third run (iteration 31 to 45) |
| man -> woman =  king -> queen | 13,893 | 10,531 | 9,794 |
| france -> paris =  italy -> rome | 46,475 | 21,690 | 17,061 |
| boy -> man =  girl -> woman | 1,611 | 1,024 | 871 |
| strong -> stronger =  clear -> clearer | 19,193 | 16,096 | 14,885 |