Assignment 1
Naive Bayes(Local and Hadoop implementation)

Vikram Bhatt [1]

Abstract

In this assignment,we implemented Naive Bayes in local machine using stream and sort method and same method is emulated on Hadoop framework.In this stream-sort method we never have to worry about running out of memory and can be done efficiently for big corpus in constant memory(because we are using the next best method to storing and reading events in memory,which is reading data sequentially on disk).Both training and testing is done through streaming assuming our model(hash table holding events counter) is large enough and test files are also huge.

1. Naive Bayes (Stream and Sort)

1.1. Training

Initially we generated hash table that stores all the counters needed for evaluation of class probabilities. Given training set has been processed sequentially through each document and outputs three types of messages:

- For every document we output the event strings "Y=y" and "Y=ANY"(y is the class label for the given document)
- For every word in a given training example,we output "Y=y and W=w" and "Y=y and W=ANY" to the standard output.During the processing we removed stop words and for stemming we used Porter stemmer in NLTK.
- Output is sorted through UNIX pipeline(sort command with LC_ALL=C option to turn off locale specified sort and use native byte order).Now that we have all the words sequentially,we can stream through the document and sum(reduce) the counts for each event locally in constant(very small) memory.This can be done very easily with the following algorithm 1 (I have used variations of this algorithm for reduction methods hereafter in streaming process)

Algorithm 1 Algorithm for reducing event counts while streaming through sorted hash table (Cohen, 2017)

Input: Sorted Hash Table
$previousKey = None$
$sumForPreviousKey = 0$
for every **event,delta** in hash table **do**
  if $event == precviousKey$ **then**
    $sumForPreviousKey+ = delta$
  else
    $OutputPreviousKey()$
    $sumForPreviousKey = delta$
    $OutputPreviousKey()$
  end if
end for
def $OutputPreviousKey()$:
if $previousKey$ is not **None then**
  print($previousKey$,$sumForPreviousKey$)
end if

1.2. Testing

Till this point we have all we need in hash table on disk(all the counters for words and class labels).But I don't want to do classification in memory, rather I want to again use stream and sort for testing phase also.The following steps are followed:

- We have all the event counts in sorted manner and stored in hash table(on disk),but still this is not good form for prediction.
- We reorganized our hash table such in the form "word,1,$wordEvents$" where $wordEvents$ are the word counters for the given word from all the classes.The dummy index 1 is used for sorting preference(explained below).For example

  abass,1,[('Y=Association_football_defenders and W=abass', '1'), ('Y=Villages_in_Turkey and

*Equal contribution [1]Department of Computational and Data Science,Indian Institute of Science. Correspondence to: Vikram Bhatt <vikrambhatt@iisc.ac.in>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

W=abass', '2')]

- After inverting the index of hash table, we generate requests for each document in the form "i,2,w_j" for i^{th} document and j^{th} word.The dummy index 2 serves the purpose of sorting(explained later).
- Generated requests are appended to the hash table containing word and corresponding word events and sorted.After sorting we have word count requests precede word count and event records and the dummy index make sure this happen correctly.(Once sorting done ,dummy index dropped.)

Stream through resulting file as follows:

- If a line contains a tuple of type "word, $wordEvents$", save that record in memory.
- The next following line contains "word, document id" with same word as above and print a message "id,word,$wordEvents$" to the standard output.
- Pipeline the standard output to sort and save it in csv file(call it indexWordCounter file).

Here comes the test phase:

- Stream through the indexWordCounter file and accumulate the sequentially contigous block for a single document i, and save them in memory(it will be small and exhaustive because the file is sorted).This block contains all the information needed to calculate posterior log probability for all the classes according the given formula.Pick out the one with maximum posterior probability
- For each possible class $y \in dom(Y)$
- $S_y = log(\frac{C["Y=y"]}{C["Y=ANY"]}) + \Sigma_{j=1}^{n} log(\frac{C["Y=yandW=w_j"]+mp_0}{C["Y=yandW=ANY"]+m})$
  where $mp_0 = 1$ and $p_0 = \frac{1}{V}$ V = vocabulary size
- Predict $\hat{y} = argmax_y S_y$

Here is the brief overview of pipelines for generating various files in the above described order.

1. ***python3 generateCtrUpdates.py DBPedia.full/full_train.txt | LC_ALL=C sort | python3 processCtrUpdates.py | python ReorganizeHashTable1.py | LC_ALL=C sort | python3 ReorganizeHashTable2.py > ReorganizedHashTable.csv***
2. ***python3 generateCtrUpdates.py | LC_ALL=C sort | python3 processCtrUpdates.py | grep -v "Y=.* and W=.*" > ClassLabelsCount.csv***
3. ***python3 generateCtrUpdates.py DBPedia.full/full_train.txt| LC_ALL=C sort | python3 processCtrUpdates.py | grep "Y=.* and W=ANY" > ClassLabelsCountForANY.csv***
4. ***python3 generateCtrUpdates.py DBPedia.full/full_train.txt | LC_ALL=C sort | python3 processCtrUpdates.py | grep -v "Y=.* and W=.*" > ClassLabelsCount.csv***
5. ***python3 generateTestMessages.py DBPedia.full/full_test.txt > TestMessages.csv***
6. ***cat TestMessages.csv ReorganizedHashTable.csv |LC_ALL=C sort -S 1G | cut –complement -f 2 -d , > MergedFile.csv***
7. ***python3 generateIndexWordCtrs.py MergedFile.csv | grep -v "^\[" |LC_ALL=C sort -n>indexWordCounters.csv***
8. ***python generateTestLabels.py DBPedia.full/full_test.txt > TestLabels.csv***
9. ***python3 generateProbs.py 263289 ClassLabelsCount.csv ClassLabelsCountForANY.csv indexWordCounters.csv > PredictedClasses.csv***

Note that commands 3,4 and 8 are merely used for generating class labels and counts for train and test files which is used for comparing ground truths and predicted labels.I calculated accuracy as ratio of number of correctly predicted examples and total number of test cases.

Table 1. Classification accuracies using stream and sort on local machine

Test Set	Development Set
10.2%	15.6%

2. Naive Bayes Hadoop Implementation

2.1. Training

- We used **Stanford Core NLP** package(written in java)(Manning et al., 2014) for stemming and lemmatization.
- We tried to emulate the local version of stream and sort process and converted into map-reduce framework.Initially, a map reduce programm written for creating hash table which holds counters required for prediction.The input key for map function is <Object

Key, Text Value> and the value is parsed for each document extracts labels and document content word by word. Reduce function just sums up the values which has same keys, which gives us count for every event.

- **hadoop jar NaiveBayesTrainer.jar NaiveBayesTrainer -libjars $NLPCOREJARS -D mapred.reduce.tasks=5 /user/ds222/assignment-1/DBPedia.full/full_train.txt /user/vikrambhatt/-NaiveBayes/HashTable**

  $NLPCOREJARS contains class-paths for Hadoop to locate Stanford CoreNLP jars,mapred.reduce.tasks used for setting number of reducers.

- We extracted vocabulary using VocabularyCounts.jar which saves unique words and the corresponding word counts, which we used it later for testing phase for calculating posterior probabilities for each class.

- **hadoop jar VocabularyCounts.jar VocabularyCounts -libjars $NLPCOREJARS -D mapred.reduce.tasks=5 /user/ds222/assignment-1/DBPedia.full/full_train.txt /user/vikrambhatt/-NaiveBayes/VocabularyCounts**

- We reorganized hash table to suit our needs in test phase.Our mapper takes input as hash table generated by above jar and and extracts word in the value and outputs (word,$wordEvents$). Reducer process the values with same keys and concatenate in a list.So at the end we have text file with (word, $wordEventsassociatedwithWord$) on same line.

- **hadoop jar ReorganizedHashTable.jar ReorganizeHashTable -libjars $NLPCOREJARS -D mapred.reduce.tasks=5 /user/ds222/assignment-1/DBPedia.full/full_train.txt /user/vikrambhatt/-NaiveBayes/ReHashTable**

- Generate test messages requests and append to the above HDFS file(Reorganized Hash Table) and sort them according to numerical order such that we have text file ready for final test phase. Each line contains (documentID,word,$wordEvents$) arranged such that words belong to same documentID appear in consecutive block.We didn't use map reduce paradigm just a write and append to HDFS and sort.

- **hadoop jar TestMessager.jar TestMessager -libjars $NLPCOREJARS -D mapred.reduce.tasks=5 /user/ds222/assignment-1/DBPedia.full/full_test.txt /user/vikrambhatt/-NaiveBayes/Test**

- The resultant file generated by above is streamed through sequentially with python scripts used in local stream and sort implementation as mapper and reducer scripts.The idea is simple and straightforward. We have input key pair mapped and reduced $(documentID, (word, WordCounts)) ==> documentID, PosteriorProbability(word, wordCounts))$, where $PosteriorProbability$ is a function which takes words and $wordEvents$ and outputs maximum probability class as described in 1.1 for each documentID. We used Hadoop streaming.jar for this method on my local machine, as our jobs are terminated with errors on cluster.

- **$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/hadoop-streaming.jar -input hdfs://input/indexWordCounters -output /output/ -mapper generateProbs.py -reducer reduceForAccuracy.py**

2.2. Results

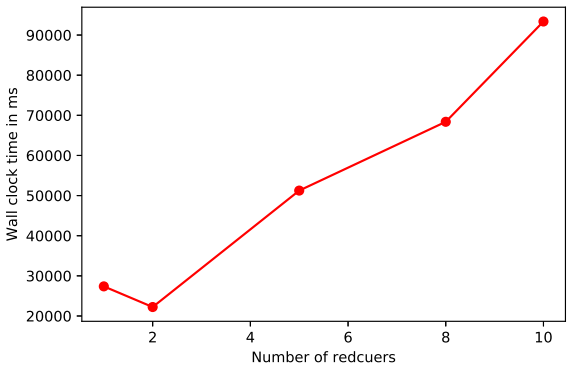

Figure 1. Wall clock time(ms) vs Number of reducers during train phase.

- Given file (full_train.txt) size is around 126MB which is smaller than HDFS block size.So our job by default produces only map input spill, so only one map task. But we can specify the number of reduce task by providing the option -D mapred.reduce.tasks in the job submission.We observed a decreasing trend initially and the increases and remains saturated after certain point.

- (White, 2009) Increasing the number of reducers makes the reduce phase shorter,since we get more parallelism.If we take this too far you can have a lot of small files(compared to HDFS block size) which is

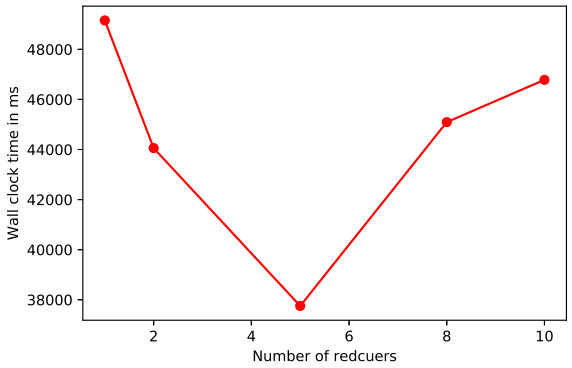

Figure 2. Wall clock time(ms) vs Number of reducers during test phase.

Table 2. Classification accuracies for naive Bayes on Hadoop framework

TEST	DEVELOPMENT
10%	14%

suboptimal because it cause load imbalance and communication overheads.An optimal number of reducers is to aim such that each reducer output atleast one HDFS block worth of output.

3. Acknowledgements

All source codes and syslogs are uploaded to github.I helped Kapil Pathak by explaining stream-sort Naive Bayes implementation,various concepts in Hadoop framework and debugging code.

References

Cohen, William. Naive Bayes and Map-Reduce. Technical report, 2017. URL http://www.cs.cmu.edu/{~}wcohen/10-605/notes/scalable-nb-notes.pdf.

Manning, Christopher D., Surdeanu, Mihai, Bauer, John, Finkel, Jenny, Bethard, Steven J., and McClosky, David. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pp. 55–60, 2014. URL http://www.aclweb.org/anthology/P/P14/P14-5010.

White, Tom. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009. ISBN 0596521979, 9780596521974.