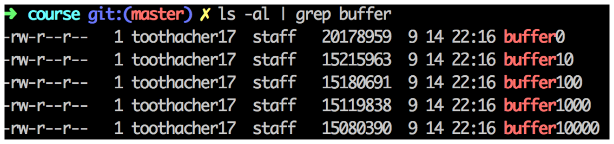
# Streaming Naive Bayes for 10605 15 Fall

**Jingyuan Liu**
AndrewId: jingyual
`jingyual@andrew.cmu.edu`

## 1 Question 1

As the question enlights, we could use a hashmap to store the event counter. The map key is the event key, and the value is the current counts of the key. We could save several keys as given by the question, 10, 100, 1000, and 10000. When the map is full, which is the key number reaches the pre-set threshold, we can print out all items in the map. the result is as follows:



```
➜  course git:(master) ✗ ls -al | grep buffer
-rw-r--r--   1 toothacher17  staff   20178959  9 14 22:16 buffer0
-rw-r--r--   1 toothacher17  staff   15215963  9 14 22:16 buffer10
-rw-r--r--   1 toothacher17  staff   15180691  9 14 22:16 buffer100
-rw-r--r--   1 toothacher17  staff   15119838  9 14 22:16 buffer1000
-rw-r--r--   1 toothacher17  staff   15080390  9 14 22:16 buffer10000
```

Figure 1: Different file size with different buffer size.

As the Figure 1 shows, the buffer0 is without buffer, the bufferX is with buffer of key size equals X. We can see that, adding buffer is obviously outperforms the algorithm without buffer. On the other hand, the buffer size, like 10, 100 or 10000, dose not influence much.

As far as I am concerned, the result is reasonable. Adding buffer will be of course better than without buffer, because it collects the items with the same key and store them in memory, thus making less printout. However, the buffer size is not so influential as expected. I think it is that the key distribution are random, so the map always tends to miss, just like cache miss. So the map does not differ much with different buffer size.

## 2 Question 2

The algorithm to calculate the vocabulary size is just similar to other count algorithms, Specifically:

1. First step: Before reading stream data, we set a previousKey = Null, and a totalCnt = 0.

2. Second step: For each stream instance, we parse the instance and get the current tmpKey.

3. Third step: If the tmpKey does not equal previousKey, then totalCnt++, and set previousKey = tmpKey. Else, pass.

4. Fourth Step: Save the totalCnt. It is the total vocabulary size.

This algorithm is able to work, because the stream data is sorted. If it does not equal the preivous key, then it is the first time to appear, so just add one to the totalCnt.

## 3 Question 3

The general framework is similar to Naive Bayes, but it is a little more complex when doing the MergeCounts. It would need two previous keys record both word information and doc information. Besides, before we know the df value of a word, we could not print out, and we would need two lists to store the values to print later. The general algorithm contains three major parts: ReadFile, Sort, MergeCounts.

### 3.1 ReadFile

In this step, we just read file and produce output. Besides, considering in the final steps we need to calculate the idf value, so we need to record the whole doc size. We can add some ascii prefix to make sure that after sorting, the whole size will be the first to print out.

Input: original file

Output: (wi, dj) and ds. The wi is the ith word, the dj is the jth doc, and ds is the doc size.

### 3.2 Sort

In this step, we sort the coming stream from ReadFile step. The sorting is based on two keys: the word and docId. Besides, the whole filesize ds will be the first to output.

Input: (wi, dj) and ds.

Output: ds, ordered (wi, dj). The order is by the combined key wi+dj. For example, (w1, d1), (w1, d1), (w1, d2), ...(wi,dj), (wi,dj+1), (wi+1,d1).

### 3.3 MergeWord

Merging is not only based on word key, but also related to doc key. We will use two temp lists to store some value, however, the two temp lists will not be big, because we just store different word and doc combinations.

Input: ds, ordered (wi, dj)

Output: (wi, dj)    (tf=k, idf=m)

Specific Steps:

```
Initialize wordPreKey, docPreKey, tfTmp = 0, dfTmp = 0, totalSize = ds, tmpKeyList, tmpValueList
For instance in Stream:
    parse instance, get wordTmpKey, docTmpKey
        if wordTmpKey == wordPreviousKey:
            if docTmpKey == docPreviousKey: tfTmp++
            else:
                tmpKeyList.add(wordPreKey+docPreKey); tmpValueList.add(tfTmp);
                reset tfTmp = 0 and docPreKey = docTmpKey; dfTmp++
        else:
            for key, value in keyTmpList, ValueTmpList:
                print    (key(0),key(1))    (tf=value, idf=(totalSize/dfTmp))
            reinitialize everything; then set preKeys = tmpKeys
```

## 4 Question 4

The figure is as follows, the x is the doc size, and the y is the word size:

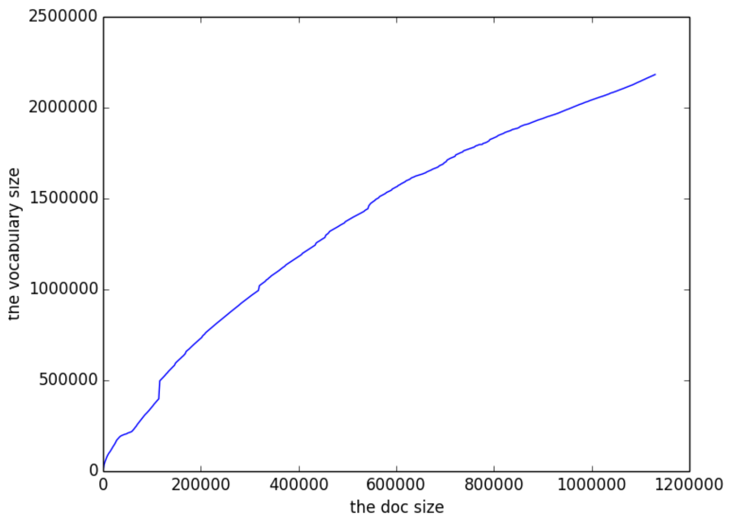


Figure 2: The trend of word increasement over doc.

The distribution is more like a logarithmically distribution. I think it is reasonable. As the doc size grows, the growing speed of word should decrease, because most words may already appread before.

For validation, we can try some other distributions. We can set x as sqrt(doc size), and y the word size. The figure is as follows:

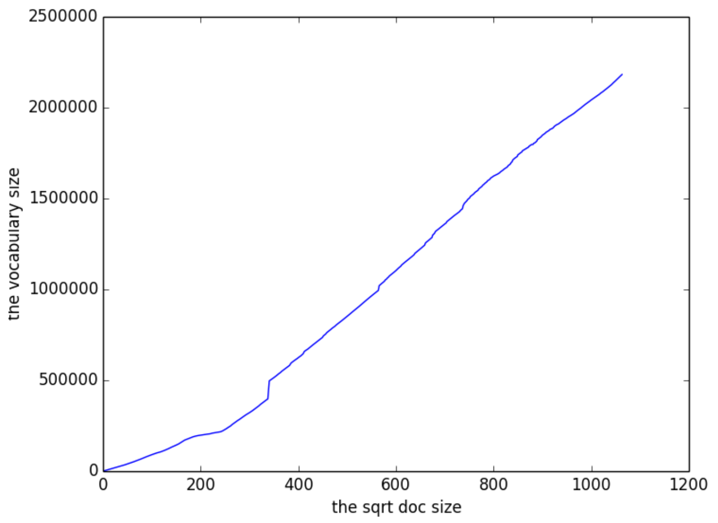


Figure 3: The trend of word increasement over sqrt(doc).

As we can see from the Figure 3, the doc increasement is more linearly related over the sqrt doc increasment size, which is just as mentioned in classes.