Naive Bayes and Map-Reduce

William Cohen

September 6, 2017

1 A Baseline Naive Bayes Algorithm

We'll start out with a very simple learning algorithm: multinomial Naive Bayes. Our implementation is in Table 1. Each training example is a labeled document $d = (i, y, (w_1, \ldots, w_{n_i}))$ with an identifier i, a label y from a small set $Y = \{y_1, \ldots, y_K\}$, and a "bag of words". The bag of words are w_j's, encoded here as a list of strings, so that w_j is the word/token at position j of document i. For example, the bag of words for the paragraph below would be the strings: "when", "scaling", "to", ..., "large", "datasets", and ":". The test documents are the same, but their labels are unknown. Our goal is to read a training set, build a classifier, and then predict a label for each test example.

When scaling to large datasets, the first thing to remember is that main memory (RAM) is, relatively speaking, limited and expensive, while disk space is less limited and cheap. So we want to carefully control how much memory we use. In particular, if you care about processing large datasets:

Principle 1 *Never load the training or test data into main memory. More generally, never load the entire input file into memory.*

Also, disk memory is much cheaper to access if we stream through a file sequentially, and much more expensive if we try and simulate random access on disk. This suggests another "principle", and a definition, for a learning scheme called *streaming learning*.

Principle 2 *The fastest way to access data on disk is to read the data sequentially.*

Multinomial Naive Bayes: Version 1

1. Inputs: a training dataset T; a test dataset U; and V, the number of distinct words in the training set.
2. Initialize an *event counter* C.
3. (Train.) For each training example $d = (i, y, (w_1, \ldots, w_{n_i}))$ in the stream of examples from T:
   (a) Generate the event string e_y: "Y=y".
   (b) Generate the event string e_*: "Y=ANY"
   (c) Increment counters: $C[e_y]$++; $C[e_*]$++
   (d) For each word w_j at position j in d:
      i. Generate the event string $e_{j,y}$: "Y=y and W=w_j"
      ii. Generate the event string $e_{*,y}$: "Y=y and W=ANY"
      iii. Increment counters: $C[e_{j,y}]$++; $C[e_{*,y}]$++
4. (Test.) For each test example $d = (i, (w_1, \ldots, w_{n_i}))$ in the stream of examples from U:
   (a) For smoothing, let $m = 1$ and let $p_0 = \frac{1}{V}$.
   (b) For each possible class $y \in Y$:
      - Estimate the log-probability $\log P(y|w_1, \ldots, w_{n_i})$ as

$$S_y = \log \frac{C[\text{“Y=}y\text{”}]}{C[\text{“Y=ANY”}]} + \sum_{j=1}^{n_i} \log \frac{C[\text{“Y=}y \text{ and W=}w_j\text{”}] + mp_0}{C[\text{“Y=}y \text{ and W=ANY”}] + m}$$

      - predict $\hat{y} = \text{argmax}_y \; S_y$

Table 1: A streaming implementation of the multinomial Naive Bayes algorithm

Definition 1 (Streaming Machine Learning) *In a* streaming machine learning system*, the training examples are read one at a time from disk, and only* one example at a time *is stored in main memory. More generally, a* streaming algorithm *processes input sequentially, and uses only a constant amount of storage.*

Some people use this term for a program that may stream through the examples on disk more than once, or it might need to stream through them in a special order (say, randomly). However, our implementation of naive Bayes just streams through the training example once, so it satisfies even a very strict definition of "streaming".

In our algorithm, rather than building a classifier explicitly, we will accumulate sufficient statistics for performing classification. I'm doing this to emphasize that the main activity in this program is simply counting.

In particular we use an an *event counter* C. $C[e]$ stores an integer count for an event e, where events are encoded as strings. It is implemented as a memory-based hash table mapping strings to integers, where the count for a so-far-unseen key is assumed to be zero, by default.[1]

For simplicity, our implementation ignores the computation of one other statistic we use in streaming—the number of distinct words V in the corpus, which is used in the smoothing. For now, we just assume that $V$is another input to the learner.

2 Scaling up the baseline

2.1 Event-counting without an in-memory hashtable

The bottleneck in version 1 of our NB algorithm is memory. When there are too many events to store in memory, the implementation simply breaks. There are some obvious ways to get around this, such as:

- We could store C in a B-tree, or some other sort of disk-based store. In practice, this is very slow, since we're using a disk store for random access updates to C.

[1] So in Java, C might be a HashMap⟨String,Double⟩, and accessing $C[e]$ might be done by: tmp = c.get(e); if (tmp==null) return 0; else return tmp.getValue(); For efficiency, it's important that zero counts are *not* explicitly stored in the hash table.

- We could store C across multiple machines, with some sort of distributed hash table. This is doable, but expensive in terms of hardware.

We're going to use a perhaps less obvious approach, where we will rewrite the algorithm with a limited set of operations, all of which are efficient for large datasets. We have one operation already: streaming. The other operation is *sorting*.

Principle 3 *Sorting large files can be done efficiently, with a mimumum amount of memory*[2]*, and can also be easily parallelized.*

So now we should ask: can we radically reduce memory usage by making use of sorting? The answer turns out to be "yes": we can rewrite the algorithm to use *only* streaming and sorting. We'll assume that there is some constant upper bound d on the length n_i of every example, so we can stream through the examples, holding one example at a time in memory: by assumption this uses $O(d) = O(1)$ memory only. Let us begin start with the task of building a disk-based version of the event counters. The code is shown in Table 2, and the basic idea is illustrated in Figure 1.

In brief, instead of incrementing a counter in a hashtable with key e, we simple construct a "message" indicating that we need to increment this counter. Conceptually, the message will be "sent" to another routine that will process these counter-updates, but in implementation, a message is just a pair (e, δ) where δ is the amount we need to increment e: here all the δ's are 1. For instance, one such message might be (aardvark,1).

To process the messages, we first *sort the messages by key*. An easy way to separate the keys and values by a tab when we write them out to a file, rather than using the (k, v) syntax I'm using in the writeup. If we always do this, then the sorting operation can be implemented in Unix with the sort command. If we've written the messages to the file msgs.txt, for instance, we'd use the command

```
% LC_ALL=C sort -k1 msgs.txt > sorted-msgs.txt
```

[2]If you care to read the details, one efficient and easy-to-understand memory-efficient algorithm is *merge sort*. The main idea in a merge sort is to sort small subsets of the data in-memory, saving these chunks onto disk, periodically merging together two previously-sorted subsets to create a larger sorted subset. Merging two sorted subsets requires streaming through those two subsets in parallel, which is efficient with modern drives.

Multinomial Naive Bayes: Version 2

1. Inputs: a training dataset T, and a test dataset U.
2. (Train 1: Generate counter updates.) For each training example $d = (i, y, (w_1, \ldots, w_{n_i}))$ in the stream of examples from T:
   (a) Generate the event string e_y: "Y=y".
   (b) Generate the event string e_*: "Y=ANY"
   (c) Output key-value pairs: $(e_y, 1)$, and $(e_*, 1)$
   (d) For each word w_j at position j in d:
      i. Generate the event string $e_{j,y}$: "Y=y and W=w_j"
      ii. Generate the event string $e_{*,y}$: "Y=y and W=ANY"
      iii. Output key-value pairs: $(e_{*,y}, 1)$ and $(e_{j,y}, 1)$
3. (Train 2: Sort the counter updates.) Sort the key-value pairs produced in step 2 by key (i.e., by event string).
4. (Train 3: Accumulate the counter updates.) Stream through the key-value pairs and, for each *contiguous* run of pairs $(k, v_1), \ldots, (k, v_m)$ that have the *same key*, output a single key-value pair (k, s) where $s = \sum_{j=1}^{m} v_j$.
5. (Test.) See Table 5.

Table 2: A naive constant-memory implementation of the multinomial Naive Bayes algorithm

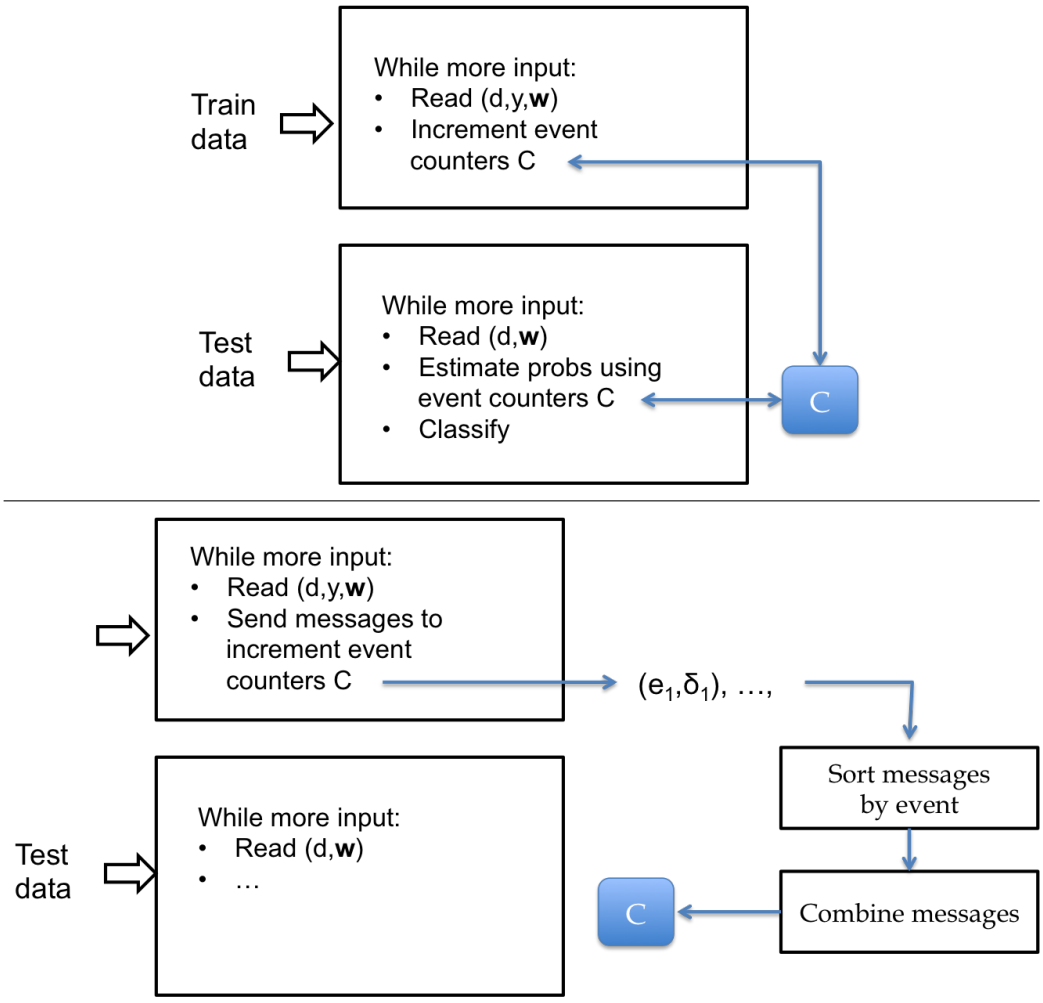

Figure 1: Top, accumulating even counts in memory. Bottom, accumulating counts via a stream-and-sort process. The test process will be discussed below.

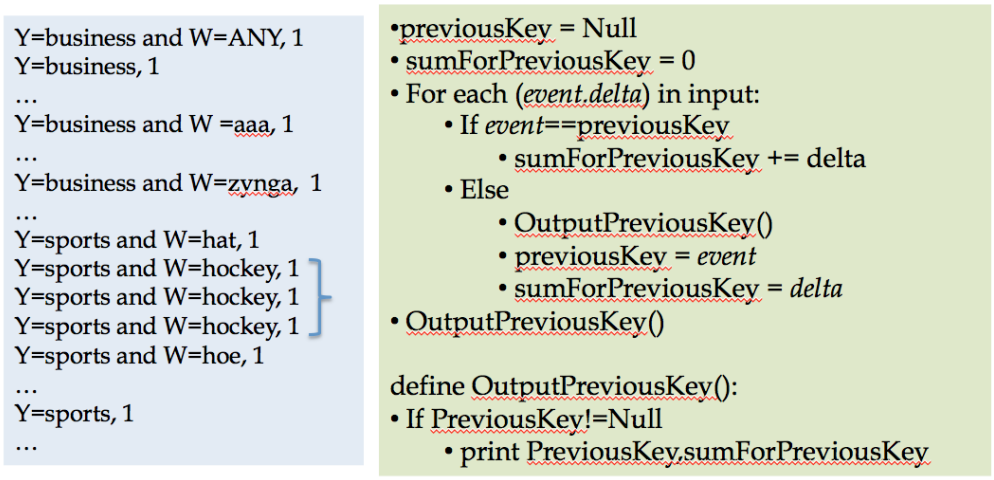

Figure 2: Accumulating counts in a sorted message file using constant memory. The bracket picks out a contiguous run of pairs with the same key.

Setting the environment variable LC_ALL=C means the sort will sort by the normal ASCII values, instead of trying to sort "intelligently" by an locale-specific encoding. Once the messages are sorted by event, it's simple to compute the proper count using only constant storage, as Figure 2 illustrates, with a sample input and some pseudo-code. Note the only memory needed is for the variables sumForPreviousKey and previousKey, and a single key-value pair. Below, we'll call this sort of aggregation of messages a *reduce* operation.

To review, the design pattern we've used here is to (1) replace random access operations by turning them into messages, then (2) sorting those messages to remove the "randomness" and allow us to (3) process them systematically, with a streaming process.

Putting it together, a possible Unix implementation would consist of one main program generateCtrUpdates that streams through T and outputs a list of key-value pairs; a call to sort; and a second main program processCtrUpdates that streams through the sorted key-value pairs (i.e., counter-update messages) and processes them. In Unix a working pipeline for this could be:

```
% generateCtrUpdates train.txt | LC_ALL=C sort | processCtrUpdates
```

It may not be obvious to you why this implementation of sorting is better than, say, loading all the event counts into an array and using Java's built-in

sort method. In fact, sorting is (obviously) not always memory-efficient: we're exploiting the fact that this particular implementation is memory-efficient. In more detail, this pipeline works both because Unix "pipes" were very carefully designed, and because unix sort obeys Principle 1. Under the hood, here's what happens.

Unix will start up three processes, one for sort, and one for your mains. Processes connected by pipes will run concurrently, with a reading process being put to sleep if it runs out of input, and a writing process put to sleep if its reader gets too far behind. Further, the algorithm used for sort is a merge sort: it will read a "small" chunk of data from the file (by default, a few million lines at a time), sort that chunk and save the result in a temporary file, and then repeat for each later chunk, periodically merging the temp files so that it doesn't produce too many on a huge input. After the last line of input is read, sort begins to output the result of the final merge of its temp files.

So the execution will happen like this:

- Phase 1: generateCtrUpdates and sort run concurrently, producing messages and sorting them into larger and larger temp files, each of which contains a sorted subset of the data.
- Phase 2: once all the messages are sorted, sort and processCtrUpdates run concurrently, printed sorted messages and turning them into final values for the event counter.

To review, we've now taken our original algorithm and made it "bullet-proof" with respect to large datasets, in that it will never run out of memory. Put another way, we've gone from an implementation that uses way too much memory to one that uses an very small amount of memory.

2.2 Optimizing by caching

Suppose the original corpus has a total of n words, counting duplicates (i.e., total length $\sum_i n_i$ of all the documents is n) and m distinct words, and there are k classes. For English text m is often $O(\sqrt{n})$.

Training the in-memory Naive Bayes version takes time $O(n)$ and uses memory $O(km)$. The stream-and-sort version takes time $O(kn)$ for generateCtrUpdates, since it will produce multiple counters for each word, time $O(kn \log kn)$ to sort, and $O(kn)$ for processCtrUpdates. The bottleneck is the

sort step, because we generate so many event-counter related "messages". This is typical: particularly when we start parallelizing things, we must consider the "message complexity" of an algorithm.

One practical way to reduce the cost of sorting here is to *reduce the size of the input to the sort* by doing some message aggregation in memory. One approach to this is to use a "small" event-counter to accumulate partial counts, somewhat the way sort accumulates "small" chunks of its input to sort in-memory. When the event-counter gets too large (for instance, when it has counts for too many events in its hash table) then you simply output the partial counts, to be sorted and later processed with processCtrUpdates, and then clear the entries in the event counter. This optimization is shown in Table 3.

The key difference between this algorithm and Version 1 is that this algorithm still needs constant memory, where that the constant is now B, rather than needing $O(m)$ memory. The difference between this algorithm and Version 2 is that we have introduced a cache of size B to accumulate counts, which will reduce the size of the input to the sort. The major savings here comes from frequent words: note that there will be some reduction in sort-input size whenever B is large enough to have any counts larger than one. Since many words are frequent in English, there is likely to be a large savings, even for a B that is only a few thousand.

To review, the algorithm is not only "bulletproof", in that it will never run out of memory: it is also completely *adjustable* in that we can take advantage of whatever space is available: in the limit, we can make it nearly as fast (and space-hungry) as Version 1.

As an aside: when you run out of memory, the obvious response is to start optimizing memory usage. We could certainly do much better in algorithm 1 than using a hashtable with string keys, but no matter how well we optimized, we would still have code that would break on a large enough dataset. What we've discussed here is an alternative: we started with a slow constant-memory algorithm that was robust against large datasets, and added some memory-intensive processing as an optimization.

3 Using a huge classifier

This learning algorithm leaves us with event classifiers too large to load into to memory and use for classification. In the lectures, I explain one simple

Multinomial Naive Bayes: Version 3

Subroutine: To "increment-and-spill" a counter for e:

- If e has not already been stored in the hash table for C, and C already contains at least B keys, then
  - For each event-count pair (e, c) in C, output the key-value pair (e, c).
  - Re-initialize C to be an empty event counter, by clearing the hash table (implicitly setting the counts to zero in C).
- Increment $C[e]$ by one.

1. (Train 1′: Generate *optimized* counter updates.) For each training example $d = (i, y, (w_1, \ldots, w_{n_i}))$ in the stream of examples from T:
   (a) Generate the event string e_y: "Y=y".
   (b) Generate the event string e_*: "Y=ANY"
   (c) Increment-and-spill counters for e_y, e_*
   (d) For each word w_j at position j in d:
      i. Generate the event string $e_{j,y}$: "Y=y and W=w_j"
      ii. Generate the event string $e_{*,y}$: "Y=y and W=ANY"
      iii. Increment-and-spill counters for $e_{*,y}$, $e_{j,y}$
2. "Spill" any remaining event counters: i.e.,
   - For each event-count pair (e, c) in C, output the key-value pair (e, c).

Table 3: An optimized constant-memory implementation of the multinomial Naive Bayes algorithm

("found", ("Y=happy and W=found",12067), ("Y=sad and W=found",14875)),
("aardvark", ("Y=happy and W=aardvark",12), ("Y=sad and W=aardvark",2)),
...
("today", ("Y=happy and W=today",43422), ("Y=sad and W=today",47242))

("I, ("Y=happy and W=I",95621), ("Y=sad and W=I",99532)),
("failed", ("Y=happy and W=failed",171), ("Y=sad and W=failed",65534)),
...

Table 4: Desired inputs for a streaming test routine, with a horizontal lines separating the input for document 1 and the input for document 2.

way to "finesse" this problem when the test set is small. However, to explain better the power of stream-and-sort processing, we will discuss now the more complicated task of *classifying examples* using the large-memory version of Naive Bayes.

Assume the test set is also large, and again, assume that we cannot load the event counters into memory, and that we can use only use two operations: constant-memory streaming algorithms, and sorting. We'll need to be somewhat devious to perform classification with these limited operations.

To be concrete, suppose I want to classify tweets by sentiment, so $Y = \{\mathsf{happy},\mathsf{sad}\}$, that I've stored counts for all relevant events on disk as key-value pairs, and the test document has id 123 and the text "Found an aardvark in Zynga's Zooville today!", i.e., after stopwording our test case d is

$$d = (123, (\text{found},\text{aardvark},\ldots,\text{today})$$

To classify d we need all the relevant event counts for the words in d, and to classify it in constant space, we can't afford to store much more than that in memory. So we must arrange things so that d and the required counts appear together in a file that we can stream through, something like the example shown in Table 4.[3]

Constructing the inputs of Table 4 would be easy if we could just put the counters in memory: then we could just fetch all the counts for each word in d, and put them in some data structure. What we'll discuss next is another "design pattern", where we conceptually use *messages* between a document d and an "event-count database" to reorganize our data.

To make the next bit of code a little easier assume we've re-organized the event counts: instead of key-value pairs of the form *(event,c)* we have key-value pairs structure *(w,wordCounts$_w$)* where *wordCounts*$_w$ is a data structure listing all the counts associated with the word w. For instance, *wordCounts*$_{\mathsf{aardvark}}$ would be be

[("Y=happy and W=aardvark",12), ("Y=sad and W=aardvark",2)]

(where the square brackets indicate a list) and the correspondng key-value pair might be

[3]The table actually shows the inputs for two example documents, the one above an the more somber "I failed my midterm last week". Also, to keep the discussion simple, I'm ignoring the counts for class frequencies.

("aardvark",[("Y=happy and W=aardvark",12), ("Y=sad and W=aardvark",2)])

Let's call this reorganized data structure C'. Our algorithm will work as follows.

- For each document i, and each word position j in document i, send a message to C' requesting all event-counts concerning word w_j. The message contains the word w_j, and the index of the requesting document, i.

  For example, the messages associated with document 123 might be the following, written as key-value pairs:

  ("found", 123)
  ("aardvark", 123)
  ... ...
  ("today", 123)

- Sort the counter-request messages to C' by word.

- Merge the sorted the messages to C' with the sorted content of C', so that the requests for counters related to the word "aardvark" are adjacent to the record $wordCounts_{\mathsf{aardvark}}$. After the merge, the result will be something like the following:

  ("aardvark", [("Y=happy and W=aardvark",12), ("Y=sad and W=aardvark",2)])
  ... ...
  ("aardvark", 123)
  ... ...
  ("zynga", [("Y=happy and W=zynga",3761), ("Y=sad and W=zynga",973)])
  ... ...
  ("zynga", 123)
  ... ...

- Exploit the locality of the merged data to stream through it and, in constant space, construct another set of messages, which conceptually "reply" to the "request messages" by forwarding the *wordCounts* records "to" every requesting document i. This is easy to do, especially if the messages from documents requesting a word (e.g., "aardvark")

are sorted into the stream right *after* the tuple holding the appropriate wordcounts (e.g., $wordCounts_{aardvark}$),[4] as shown above.

The messages that we construct for this "reply" will again be key-value pairs. We will use i as the key, and a combination of w and $wordCounts_w$as the value, so we can easily sort together the messages "to" document i. For instance, the messages with 123 as the key would be:

```
...
(123, ("aardvark", [("Y=happy and W=aardvark",12), ("Y=sad and W=aardvark",2)]) )
...
(123, ("zynga", [("Y=happy and W=zynga",3761), ("Y=sad and W=zynga",973)]))
...
```

Of course, these messages will not be grouped this way: instead we will initially see replies to all the requests for counts for "aardvark" together, then requests for "aaron" and other a-words, ending up with requests for z-words like "zynga".

- Finally, we will sort these messages again, by recipient, and stream through them to classify. The sorted stream of replies will collect together all the word count records for all the words in a document with a particular id: for instance, the records for document 123, shown above in this example, will be sorted together. This is enough information to produce a classification for a document (again, for simplicity, I'll ignore the class frequencies).

The control flow for the whole algorithm is shown in Figure 3, and the algorithm itself is shown in Table 5.

4 Review, and Connections to Hadoop

Above we stepped through a simple program, implementing it in an unusual, memory-efficient style, where we used two operations: sorting, and streaming.

[4]This can be easily arranged, with some slight tweaks to our message and data formats. For instance, we could replace each word-count record pair *(w,wordCounts$_w$)* with a triple *(w,1,wordCounts$_w$*, replace each word-count request pair *(w,i)* with a triple *(w,2,i)*, and sort of the first two fields of the triple. In the class slides I suggest an alternative approach that involves prefixing messages with special characters that have a known sort position.

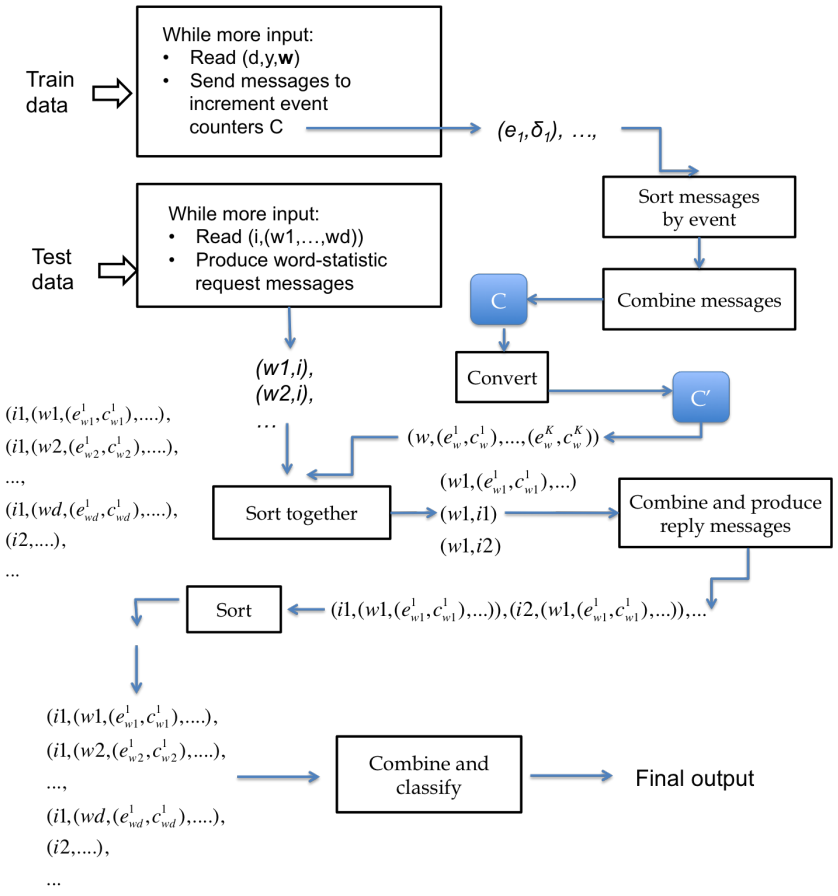

Figure 3: The workflow for testing a classifier stored on disk.

1. Reorganize C to make C'.

   (a) Stream through C and for each event count (e, n_e), let W by the word such that $e =$"Y=y *and* W=w". Print the message (w, e, n_e) to temporary file 1.

   (b) Sort temporary file 1 by word w.

   (c) Stream through the sorted temporary file 1 and convert it to pairs $(w, \mathit{wordCounts}_w)$, exploiting the fact that all the event counts for w are now stored sequentially. Store the output in file C'.

2. (Produce word-count requests). For each document $i, (w_1, \ldots, w_{n_i})$:

   (a) For each word position j in document i, write a message (w_j, i) to temporary file 2.

3. (Produce word-count replies.)

   (a) Concatenate C' and temporary file 2, sorting them by word, so that the word-count records from C' precede the requests, from temporary file 2.

   (b) Stream through the resulting file as follows:

   - If a line contains a tuple $(w, \mathit{wordCounts}_w)$, save that record in memory. (It will be small, containing one count for each possible label y.)
   - If a line contains a tuple (w, i), then print a message $(i, w, \mathit{wordCounts}_w)$ to temporary file 3.

   (c) Sort temporary file 3 by document id i.

   (d) Stream through temporary file 3, accumulating together all the sequentially contiguous tuples $(i, w, \mathit{wordCounts}_w)$ for a single document i, and saving them in memory. When this sequentially contiguous block is finished, output a message $(i, \hat{y})$ where $\hat{y}$ is the predicted class for i.

Table 5: Testing a Naive Bayes classifier that does not fit in memory.

Of the streaming operations, we used two types: ones which require as input a set of key-value pairs that have been sorted by key, which we called *reduce* operations, and ones which did not. Let us give a name to the more general streaming algorithms and call them *map* operations.

In Hadoop, every algorithm is expressed as a sequence of these sorts of operations. In fact, Hadoop algorithms are more restricted: they consist of a series of *map-reduce* steps.

Definition 2 (Map-reduce step) *A map-reduce step consists of one streaming map algorithm, which must output key-value pairs; followed by a sort; followed by a streaming reduce algorithm.*

Either the reduce step or the map step can be trivial steps, which just copy the input to the output, so it's clear that anything we could implement before can still be implemented in this map-reduce framework.

The map-reduce framework has an important advantage: it can be implemented in parallel, using a cluster of fairly inexpensive components. Hadoop uses a distributed file system called the *Hadoop file system* (HDFS), in which large files are distributed across many machines. To the user, a large file looks like a directory containing many subfiles, called *shards*, such that appending the shards together would reconstruct the original file. In fact, however, the shards are stored on different machines in the cluster. Usually all data HDFS is stored in key-value pairs, and some hash of the key is used to *partition* the pair, i.e., to assign it to a particular machine in the cluster.

To run a map-reduce job on Hadoop, first the map step is performed, in parallel, by running the map process independently on different shards. The map outputs are then re-partitioned by key, and then sorted.[5] A reduce process is then run, and the results are, once again, repartitioned.

Map-reduce jobs are usually written in Java using a particular API. Alternatively, you can use *Hadoop streaming*,[6] which allows you to perform map-reduce operations where the map and reduce processes are arbitrary shell programs, such as the ones discussed here. In other words, *the implementations discussed here can be ported directly to Hadoop streaming*, with few changes to the streaming methods. The major difference will be in how to startup the jobs, and how to "shard" the original input files.

[5]In fact, the sort is not guaranteed to be complete: the only guarantee is that all the pairs for a particular key will be saved contiguously in the file.

[6]http://hadoop.apache.org/docs/r1.2.1/streaming.html

There is one other difference in Hadoop: while it's possible to use the optimization of Version 3 of the algorithm in Hadoop, it's awkward to use in streaming Hadoop. A better approach is to use Hadoop *combiners*, which I will not discuss here.

5 Other stream-and-sort tasks

There are a number of other common tasks that can easily be performed with a stream-and-sort pattern (or, if you prefer, a single map-reduce step). One is building an *inverted index*. Imagine you have a corpus like the one below:

Doc id	Tokens
d001	an aardvark looks funny
d002	i love playing zooville
d003	dmitri martin is funny an cool
d004	i found an aardvark in zynga zooville
...	...

An inverted index associates a token with all the documents that contain it, for instance:

Token	Doc Ids
aardvark	d001, d004
an	d001, d003, d004
cool	d002
i	d002, d004
is	d003
dmitri	d003
found	d004
funny	d001, d003
in	d004
looks	d001
love	d002
martin	d003
zooville	d002, d004
zynga	d004

The algorithm for doing this, which is very similar to the algorithm of Table 2, is shown in Table 6. The code needed to replace a sequence of key-value pairs with the same key with a (key,list) pair is shown in Figure 4.

Building an inverted index

1. Inputs: a corpus of documents.
2. (Generate messages.) For each document $d = (i, (w_1, \ldots, w_{n_i}))$, where i is the document id,
   (a) For each word w_j at position j in d:
      i. Output the key-value pair (w_j, i)
3. (Sort.) Sort the key-value pairs by key (i.e., by the word w_j).
4. (Accumulate indices.) Stream through the key-value pairs and, for each *contiguous* run of pairs $(w, i_1), \ldots, (w, i_m)$ that have the *same key* w, output a single key-value pair (w, L) where L is the list $i_1, \ldots, i_m$.

Table 6: Building an inverted index

```
•prevKey = Null
• sumForPrevKey = 0
• For each (event, delta) in input:
    • If event==prevKey
        • sumForPrevKey += delta
    • Else
        • OutputPrevKey()
        • prevKey = event
        • sumForPrevKey = delta
• OutputPrevKey()

define OutputPrevKey():
• If PrevKey!=Null
    • print PrevKey,sumForPrevKey
```

```
• prevKey = Null
• docsForPrevKey = [ ]
• For each (word, id) in input:
    • If word==prevKey
        • docsForPrevKey.append(id)
    • Else
        • OutputPrevKey()
        • prevKey = word
        • docsForPrevKey = [id]
• OutputPrevKey()

define OutputPrevKey():
• If PrevKey!=Null
    • print PrevKey, docsForPrevKey
```

Figure 4: On the left, reducing to the sum of all values associated with a key. On the right, reducing to a list of all values associated with a key

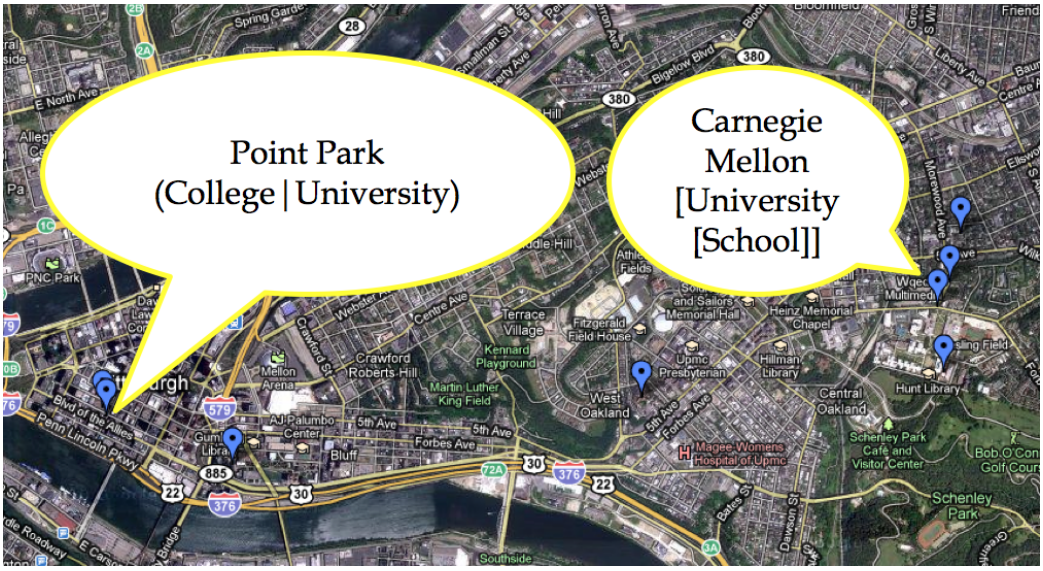

Figure 5: Locations of universities in Pittsburgh from a noisy DB

Another very different task that can be solved with a map-reduce is approximately normalizing place names. In a previous project I wrote some code to clean up a noisy DB of place names. A common problem was that there were near-duplicates: for example "Point Park College" and "Point Park Universirty" were both in the database, and located a block or two apart (as shown in Figure 5). The collection of names was too large to fit in memory, so I used the following process.

1. First, stream through the collection DB, and extract pairs consisting of a name x and a location z (the location was latitude-longitude coordinates, i.e., two-dimensional.) For each name, output all word n-grams that approximate that name, and pair each n-gram with the location z. For example, from ("Point Park College", z_1), you might output the pairs ("Point Park College", z_1), ("Point Park", z_1), and ("Park College", z_1), if the n-grams were for only $n = 2$ and $n = 3$.

2. Sort these key-value pairs by key (the word n-gram).

3. Accumulate contiguous key-value pairs for the same n-gram, by adding all the z's into a single Gaussian—i.e., accumulate the sufficient statis-

tics for the mean latitude, mean longitude, and variance of the Gaussian.

4. Finally, scan through these and discard n-grams associated with Gaussians with high variance. This removes n-grams (like, say, "Park College") which are ambiguous, and can refer to many different places across the country, but retains ones (like "Point Park" or "Carnegie Mellon") which are always paired with nearby locations.