COMP 212 Spring 2015
Homework 09

In this homework, you will implement a parallel programming abstraction called *extract-combine*, which is the essence of cluster-programming techniques like Hadoop and Google's MapReduce.

You will use extract-combine to program a simple machine learning algorithm called *naïve Bayes classification*, and use this classification algorithm to automatically label news articles with categories (such as "econonmics" or "government/social"); another typical application is spam filtering, labeling an email with categories "spam" or "not spam". The first phase of classification is *training*, where word frequency counts are extracted from a collection of documents that have been labeled with categories. In the next phase, these frequencies are used to predict the category for unlabeled documents. Naïve Bayes classification is called "naïve" because it treats a document as a multiset of words, ignoring the order of the words and other dependencies. It is called "Bayes" because a fact about probabilities called *Bayes' rule* is the main component of the method.

1 Extract-combine

For many data analysis tasks, the input is a large collection of *documents* of some sort, which we call a *dataset*. Here, we will use a dataset whose documents are Reuters news articles:

```
England midfielder Paul Gascoigne stunned Hearts with two goals in 90 seconds...

The Stock Exchange of Singapore said on Tuesday that a technical fault...
```

One common pattern for processing such a dataset is to

1. *extract* some information from each document, and then
2. *combine* the extracted information from all documents

It is often convenient if the extraction phase produces a sequence of key-value pairs (with the keys not necessarily unique), and the combination phase combines these pairs into a dictionary.

For example, suppose we want to count the frequencies of each word in the following dataset, which has two documents:

```
this is is document 1

this is document 2
```

We would

1. Extract from each document the sequence of pairs $(w, 1)$ for each word in the document:

   ```
   ("this",1)
   ("is",1)
   ("is",1)
   ("document",1)
   ("1",1)

   ("this",1)
   ("is",1)
   ("document",1)
   ("2",1)
   ```

2. Combine these pairs into a dictionary mapping words to numbers, by adding the values associated with each word:

   ```
   "1" ~ 1
   "2" ~ 1
   "document" ~ 2
   "is" ~ 3
   "this" ~ 2
   ```

This suggests a higher-order function

```
    val extractcombine : ('a -> (key * 'v) Seq.seq)
                       -> ('v * 'v -> 'v)
                       -> 'a Seq.seq -> 'v dict
```

This first argument is the *extractor*, which extracts a key-value sequence from an `'a`. The second argument is the *combiner*, which is used to combine values when the same key occurs multiple times in the extracted information.

For example, for frequency counting, we have

```
val wordcount : string Seq.seq -> int dict =
    extractcombine (fn doc => Seq.map (fn w => (w, 1)) (words doc)) Int.+
```

Here we assume that `'a dict` is a type of dictionaries, whose keys are strings, and that the function `words` divides a string into words. The extractor pairs every word in the document with `1`, and the combiner is addition. The resulting function maps sequences of documents to a word-frequency dictionary.

1.1 Data source

For many applications of extract-combine, each individual entry in the input dataset is not very large, but the overall number of documents is big, so that the size of the entire dataset is megabytes or gigabytes or more.

The above type for `extractcombine` assumes a sequence (`Seq.seq`) of documents as an input. Since an `'a Seq.seq` is represented by a value in memory (RAM), this means that to use `extractcombine`, we first need to load the entire dataset as a value in memory. However, for many applications, the information being collected from the data is much smaller than the dataset itself. For example, when we count word frequencies, the output is proportional to the number of *unique* words in the documents, which will likely be much less than the total number of words in the documents. Thus, we should be able to process a collection of documents that is too big to fit in memory, as long as the number of unique words fits.[1] But to do this, we need to generalize `extractcombine` so that it does not require its input to be in memory.

At a first cut, what we want is another implementation of the `SEQUENCE` signature, whose values are stored as a file on disk instead of as a value in memory. However, for this problem, we will not need these disk-based sequences to support all of the sequence operations: in particular, we never need to *create* such a sequence (via `tabulate` or `map`), we only need to *consume* them. Indeed, the only operation we need is `mapreduce`, so we define the following type class:

```
signature MAP_REDUCE =
sig
    type 'a mapreducable

    (* Assume that
       - n is associative and commutative
       - e is a unit for n
       Then mapreduce l e n s computes the map-reduce of
       l e n on s *)
    val mapreduce :
           ('a -> 'b)      (* result for single element *)
        -> 'b              (* result for empty *)
        -> ('b * 'b -> 'b) (* merge results *)
        -> 'a mapreducable -> 'b
end
```

A type is *mapreducable* if it has a `mapreduce` function. The prototypical mapreducable type is `'a Seq.seq`, which is expressed by the following instance:

```
structure SeqMR : MAP_REDUCE =
struct
    type 'a mapreducable = 'a Seq.seq
```

[1] If the extracted information does not fit in memory, it too can be stored on disk, but we won't pursue that here.

```
    fun mapreduce l e n = Seq.mapreduce l e n
end
```

We also define a second instance `FileMR` that reads from a file:

```
structure FileMR : MAP_REDUCE =
struct

    type 'a mapreducable = TextIO.instream
                         * (string -> 'a) (* parse a line as an 'a *)

    fun mapreduce l e n (stream,parse) = ...
end
```

A `TextIO.instream` represents a way to read from a specific file on disk. The function `string -> 'a` "parses" each line of the file as a value of type `'a`. Then `mapreduce` applies `l` to each parsed value, using `e` for the end of the file, and combining results using `n`.[2]

1.2 Task

Your job is to implement the following functor:

```
signature EXTRACT_COMBINE =
sig
    structure MR : MAP_REDUCE

    structure D : DICT

    val extractcombine : ('a -> (D.Key.t * 'v) Seq.seq) (* keys are not nec unique *)
                       -> ('v * 'v -> 'v)
                       -> 'a MR.mapreducable
                       -> 'v D.dict

end

functor ExtractCombine (A : sig
                                structure Key : ORDERED
                                structure MR : MAP_REDUCE
                            end) : EXTRACT_COMBINE
```

That is, you are given

- an ordered type of keys (to be used in extraction)
- a mapreducable type (for the dataset)

[2]For efficiency, it actually starts with `e` and combines intermediate results into an accumulator using `n`, which is equivalent because `n` is assumed to be commutative.

and you must produce a dictionary (whose keys are given by the `Key` argument) and an `extractcombine` function (whose input is given by the `MR` argument)

Task 1.1 (25 pts). Implement this functor in `extractcombine.sml`.

Hints: Use the `Dict` functor (see `dict.sig`) to construct the dictionary. Note that the provided dictionary data structure has an operation

```
val merge  : ('v * 'v -> 'v) -> 'v dict * 'v dict -> 'v dict
```

which merges two dictionaries, according to the following rules:

- If `k` is in neither `d1` nor `d2`, then `k` is not in `merge c (d1, d2)`
- If `k` is associated with `v` in one dictionary, but `k` is not in the other, then `k` is associated with `v` in `merge c (d1, d2)`.
- If `k` is associated with `v1` in `d1` and with `v2` in `d2`, then `k` is associated with `c(v1,v2)` in `merge c (d1, d2)`.

You do not need to write tests for `extractcombine`. You can test by running the word frequency counting code in `wordfreq.sml`. For example:

```
   WordFreq.wordCounts_list (SeqUtils.seq ["hi there how are you", "ok how about you"])
== [("about",1),("are",1),("hi",1),("how",2),("ok",1),("there",1),("you",2)]
```

2 Text Classification

Before starting this task, in `sources.cm`, uncomment the lines for `classify.sig` and `classify.sml` and `testclassify.sml` so that the code for the next task will be load.

In this problem, you will implement a simple form of *statistical machine learning*—i.e. you will write a computer program that learns from examples. Machine learning techniques are ubiquitous these days; they play a role in web search, touchscreen input recognition, voice recognition, machine translation; sports and election and economic predictions, ... Our goal for this assignment is to categorize text documents in some way. Specifically, we will organize Reuters news articles into four categories used by Reuters: (`CCAT` = "corporate/industrial", `ECAT` = "economics", `MCAT` = "market", `GCAT` = "government/social").[3] Such a categorization might be used by a news aggregator site to automatically group articles by topic. The same idea can be used to implement spam filters (the categories are "spam" and "not spam"), to recognize what language a web page is in, and for many other purposes.

In (supervised) machine learning, the starting point is some *training examples*, which have been labeled with the correct outputs by some means (e.g. hand-labeled by a person). Based on the training examples, you build some sort of statistical model, which is then used to predict the output for new examples. The model's predictions are tested on some *test examples*, which are also labeled, but are not included in the training examples.

[3] These are the top level of a hierarchical categorization scheme; we will ignore all the lower levels.

For this assignment, we will be using a collection of Reuters news articles from the late 1990s that have been labeled with one or more of the above four categories[4].

```
MCAT       Worries about the U.S.-Iraq crisis faded on Asian markets...
CCAT       Sun Microsystems Inc president Ed Zander will sign on June 16...
GCAT       Israel's beleaguered leader Benjamin Netanyahu scored a...
ECAT       International trade through Florida totaled $26.9 billion...
CCAT,ECAT  Nothing is sacred, it appears, in the British Treasury's...
ECAT,GCAT  The following is a list of upcoming ballot initiatives...
```

The goal is to build a classifier that, given a new article, labels it with one of `MCAT` or `CCAT` or `ECAT` or `GCAT`.

We have provided training and test datasets, each with 70,000–80,000 labeled articles.

2.1 Naïve Bayesian classification

We write $P(A)$ for the probability of something, and $P(A \mid B)$ for the *conditional probability* of thing A assuming that B happened. One form of *Bayes' rule* is that,

$$P(A \mid B) \text{ is proportional to } P(A) \times P(B \mid A)$$

Bayes' rule is used to "flip" a conditional probability $P(A \mid B)$, expressing it in terms of $P(B \mid A)$ and some estimate of how likely A is outright.

For categorization, we would like to know

> how likely is it that a document has category C, given that the document words are $w_1, \ldots, w_n$?

because then we can choose the most likely category. Applying Bayes' rule with $A =$ "document has category C" and $B =$ "document has words $w_1, \ldots, w_n$?", the probability we want is proportional to

> how likely is a document to have category C? (without knowing anything about what is in the document)

and

> how likely is it for a document in category C to be the words w_1,...,w_n?

This is helpful because we can extract information from the training data that allows us to answer these questions:

[4]Lewis, Yang, Rose, and Li. RCV1: A New Benchmark Collection for Text Categorization Research. Journal of Machine Learning Research, 2004.

- To know how likely a document is to have category C, we count how many documents are in each category, and the total number of document/category pairs (where a document with k categories category counts k times towards the total):

$$P(\text{a document has category } C) = \frac{\text{number of documents with category } C}{\text{total number of categorized documents}}$$

- To know how likely a document in category C is to be $w_1, \ldots, w_n$, we make the naïve assumption that all words are independent, so that

$$P(\text{document in category } C \text{ is } w_1, \ldots, w_n) = \prod_{i=1}^{n} P(\text{word in category } C \text{ is } w_i)$$

  For the latter, we count the number of times each word occurs in each category, and the total number of words in each category:

$$P(\text{word in category } C \text{ is } w_i) = \frac{\text{number of times } w_i \text{ occurs in documents with category } C}{\text{number of words in documents with category } C}$$

Putting this all together, we get:

$$P(C \mid w_1, \ldots, w_n) = \frac{\text{number of docs in } C}{\text{total number of docs}} \cdot \prod_{i=1}^{n} \frac{\text{count of } w_i \text{ in } C}{\text{total number of words in } C}$$

One technical point is that the above probabilities often get too small to represent in a floating point number. Because the natural log function is monotone, it is equivalent[5] to compute:

$$\ln P(C \mid w_1, \ldots, w_n) = \ln \frac{\text{number of docs in } C}{\text{total number of docs}} + \sum_{i=1}^{n} \ln \frac{\text{count of } w_i \text{ in } C}{\text{total number of words in } C}$$

and then take the max of these. This keeps the numbers in a range that is easier to represent.

Another point is what to do about a word that occurs in the test document that does not appear in the training documents (so we have no idea how likely it is). There are several possibilities, but one that works well is to say that its probability is

$$\frac{1}{\text{total number of unique words in all test documents}}$$

Overall, we have the following algorithm:

1. Training: from the training examples, compute the quantities used on the right-hand side of the above equation.

2. Classifying: compute $\ln P(C \mid w_1, \ldots, w_n)$ for each category C, and then classify with the category C that maximizes this probability.

The assumption behind this algorithm is that we have a representative collection of training examples, so that the distribution of words the training examples will predict the categories for test examples.

[5] remember that log turns products into sums

2.2 Task

In `classify.sml`, your job is to implement

```
functor Classify (ClassSpec : sig
                                  structure Category : ORDERED
                                  val default_category : Category.t

                                  structure Dataset : MAP_REDUCE
                              end) : NAIVE_BAYES_CLASSIFIER =
```

where the signature `NAIVE_BAYES_CLASSIFIER` is in Figure 2.2.3. A client of your classifier will supply

- an ordered type of categories
- a default category (to be reported if there is no better information)
- a mapreducable type for the training dataset

The real exports of your classifier are:

```
signature CLASSIFIER =
sig
    structure Dataset : MAP_REDUCE

    type category

    type document = string Seq.seq
    type labeled_document = category Seq.seq * document

    val train_classifier : labeled_document Dataset.mapreducable
                         -> document -> (category * real)
end
```

where the `Dataset` and `category` types are the ones the client provided, a document is a sequence of words, and a `labeled_document` is a document together with a sequence of categories. The main export is the `train_classifier` function, which is given a training dataset of labeled documents, and produces a classifying function.

However, to make it easier to test the intermediate stages of the computation, your classifier will also export the intermediate functions described in the `NAIVE_BAYES_CLASSIFIER` signature. For everyone but the tester, we would retype the classifier with the `CLASSIFIER` signature given above.

2.2.1 Training

From the training dataset, you need to compute:

1. for each category, the count of documents in that category
2. for each category, for each word, the count of that word in documents with that category
3. the sequence of all categories
4. the total number of classified documents
5. the total number of distinct words in all documents
6. for each category, the total number of words in that category

While we could use separate extract-combines for each of these, processing the training dataset is expensive, so we would like to only traverse it once. The final four things can be computed from the first two, so we will divide training into

- an extract-combine that computes items 1 and 2
- a post-processing phase that computes items 3–6

To compute 1 and 2 in one pass, we will compute a dictionary mapping each category to (1) a count of the documents in that category together and (2) a dictionary mapping words to their frequencies in documents with that category. This is represented by the following type:

```
type counts =   int (* number of documents with that category *)
              * int WordDict.dict (* frequencies of words in docs with that category *)
```

Task 2.1 (3 pts). Define a dictionary structure `WordDict` whose keys are strings (the module `StringLt` provides an `ORDERED` instance for strings).

Task 2.2 (4 pts). Use the functor `ExtractCombine` to make an extract-combine instance whose keys are categories and whose mapreducable type is `Dataset`. Define `CatDict` to be the dictionary from this extract-combine instance.

Task 2.3 (11 pts). Define a function

```
val count_by_category : labeled_document Dataset.mapreducable
                      -> counts CatDict.dict
```

Now, we post-process the output of `count_by_category` to compute items 3–6. This is represented by the following type:

```
type postprocess_data =
      category Seq.seq (* list of categories (no duplicates) *)
    * int              (* total number of categorized training documents *)
    * int              (* total number of words *)
    * int CatDict.dict (* how many words in each category? *)
```

Task 2.4 (20 pts). Define

```
val postprocess : counts CatDict.dict -> postprocess_data
```

You will find many of the functions in the `DICT` and `SEQUENCE` signatures to be helpful.

2.2.2 Classifying

Now we use the above data to classify a given `document`.

Task 2.5 (20 pts). First, compute the log-probabilities of each category. Define a function

```
val possible_classifications : counts CatDict.dict -> postprocess_data
                             -> document -> (category * real) Seq.seq
```

that maps a document $w_1, \ldots, w_n$ to the sequence of all pairs

$$(C, \ln P(C \mid w_1, \ldots, w_n))$$

for every category C. The `Math.ln` function computes the natural log.

Task 2.6 (10 pts). Next, compute the best (maximum) category:

```
val classify : counts CatDict.dict -> postprocess_data
             -> document -> category * real
```

The function should return the default category and `Real.negInf` (negative infinity) if there are no possible classifications.

2.2.3 Putting it all together

Task 2.7 (10 pts). Finally, define

```
val train_classifier : labeled_document Dataset.mapreducable
                     -> document -> (category * real)
```

For efficiency, it is very important that this function does all processing of the training data is done before the `document -> category * real` function is returned. This is something we can do with a curried function that we cannot do with tuples of arguments! Training will sometimes take minutes, and you definitely do not want to retrain the classifier for every document that you classify.

```
signature NAIVE_BAYES_CLASSIFIER =
sig
    structure Dataset : MAP_REDUCE

    type category

    type labeled_document = category Seq.seq * string Seq.seq
    type document = string Seq.seq

    val train_classifier : labeled_document Dataset.mapreducable
                         -> document -> (category * real)


    (* ------------------------------------------------------------- *)
    (* internal components that are exported only for testing *)

    structure WordDict : DICT
    structure CatDict : DICT


    type counts =   int (* number of documents with that category *)
                  * int WordDict.dict (* frequencies of words in docs with that category *)

    val count_by_category : labeled_document Dataset.mapreducable -> counts CatDict.dict

    type postprocess_data =
          category Seq.seq (* list of categories (no duplicates) *)
        * int              (* total number of categorized training documents *)
        * int              (* total number of words *)
        * int CatDict.dict (* how many words in each category? *)

    val postprocess : counts CatDict.dict -> postprocess_data


    val possible_classifications : counts CatDict.dict -> postprocess_data
                                 -> document -> (category * real) Seq.seq

    val classify : counts CatDict.dict -> postprocess_data
                 -> document -> category * real

end
```

Figure 1: Classifier signature

2.3 Testing

Testing Each Function We have provided test functions for printing out each stage of the computation:

```
print_counts_by_category : labeled_document Dataset.mapreducable -> unit
print_postprocess        : labeled_document Dataset.mapreducable -> unit
print_possibles          : labeled_document Dataset.mapreducable -> labeled_document
                           -> unit
number_correct           : labeled_document Dataset.mapreducable
                           -> labeled_document Dataset.mapreducable -> int * int
print_predictions        : labeled_document Dataset.mapreducable
                           -> labeled_document Dataset.mapreducable -> unit
```

The outputs look like this:

- Print counts by category:

```
- TestSeq.print_counts_by_category TestSeq.dups_train;
Category: ECAT
  Num docs:1
    price 2
    stock 2
Category: GCAT
  Num docs:1
    congress 2
    court 2
```

For each category, we have the number of documents in that category, and for each word, the count of that word in that category.

- Print postprocess:

```
- TestSeq.print_counts_by_category TestSeq.dups_train;
All categories: ECAT GCAT
Total number of documents:2
Total number of distinct words:4
Number of words by category:
  ECAT 4
  GCAT 4
```

This shows all tbe categories, the total numbers of documents and distinct words, and for each category, the number of words in that category.

- Print possibles:

```
- TestSeq.print_possibles TestSeq.dups_train TestSeq.doc4;
Given Categories: ECAT
Scores:
  ECAT ~2.77258872224
  GCAT ~3.4657359028
```

This shows the given categories of the document (i.e. the ones it was labeled with), and the scores the classifier returned for each category. The classifier got things right when the maximum score is one of the given categories (note that the numbers are negative, so maximum is closest to 0).

- Print predictions:

```
- TestSeq.print_predictions TestSeq.dups_train TestSeq.docs14;
CORRECT
Given Categories: ECAT
Predicted: ECAT
stock

CORRECT
Given Categories: GCAT
Predicted: GCAT
congress

CORRECT
Given Categories: GCAT
Predicted: GCAT
court fell

CORRECT
Given Categories: ECAT
Predicted: ECAT
stock ticker
```

For each document in a sequence, print the given categories (what it was labeled with), the predicted category, and the document.

- Number correct:

```
- TestSeq.number_correct TestSeq.simplest_train TestSeq.docs14;
val it = (3,4) : int * int
```

We have also provided very small documents in the `TestSeq` module in `testclassify.sml`.

Task 2.8 (0 pts). The files

```
expected-outputs/count-by-category.txt
expected-outputs/number-correct.txt
expected-outputs/possibles.txt
expected-outputs/postprocess.txt
expected-outputs/print-predictions.txt
```

show many ways to call to these test functions and what they should output. Make sure your code matches these outputs (up to floating point rounding errors).

Evaluating your classifier Now, you should run your classifier on some real data. Download `hw09data.tgz` and unzip it to create a `data` directory inside your `hw09-handout` folder. We have provided files of three sizes:

```
Num docs   File

      68   RCV1.small_train.txt
       8   RCV1.small_test.txt

    7356   RCV1.medium_train.txt
     818   RCV1.medium_test.txt

   72398   RCV1.big_train.txt
   80435   RCV1.big_test.txt
```

You should evaluate your classifier by counting the number it gets correct, such as:

```
TestFile.number_correct (TestFile.open_file "data/RCV1.small_train.txt")
                        (TestFile.open_file "data/RCV1.small_test.txt");
```

Each run should take no more than a few minutes.

The small test is small enough that you can print out the results to look at them:

```
TestFile.print_predictions (TestFile.open_file "data/RCV1.small_train.txt")
                           (TestFile.open_file "data/RCV1.small_test.txt");
```

Task 2.9 (5 pts). Report the percentage your classifier gets correct for each of the three sizes. You should also use training/test of mismatched sizes to investigate how much additional training data helps—e.g. does using the medium training data improve results on the small test?