# COMP 212 Spring 2024
# Homework 10

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 (where a particularl key can occur more than once in the sequence), 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 :   ('k * 'k -> order)          (* comparison *)
                     * ('a -> ('k * 'v) Seq.seq)   (* extractor *)
                     * ('v * 'v -> 'v)             (* combiner *)
                     * 'a Seq.seq
                     -> ('k,'v) Dict.dict
```

This first input is a comparison function ordering some type of keys `'k`. The second input is the *extractor*, which extracts a key-value sequence from a document, where a single document is represented by a type `'a`. The third input is the *combiner*, which is used to combine values (which have type `'v`) when the same key occurs multiple times in the extracted information. The fourth input is a sequence of documents. The output is a key-value dictionary consisting of all of the keys extracted from all of the documents, where the dictionary maps a key to the combination of all of its extracted values.

For example, for frequency counting, suppose where are given a sequence of documents, where each document is a single string. Then we can do

```
fun wordcount (docs : string Seq.seq) : (string,int) Dict.dict =
    extractcombine (String.compare,
                    fn doc => Seq.map (fn w => (w, 1), (words doc)),
                    Int.+,
                    docs)
```

Here we assume that the function `words` divides a string into words. The extractor pairs every word in a document with `1`, and the combiner is addition. The resulting function creates a dictionary with the frequencies of all of the words in all of the documents.

## 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 signature:

```
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 *)
```

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

```
        * '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 where type 'a mapreducable = 'a Seq.seq =
struct
    type 'a mapreducable = 'a Seq.seq
    fun mapreduce (l, e, n, s) = Seq.mapreduce (l, e, n, s)
end
```

The homework support code also defines a second instance `FileMR` that reads from a file:

```
structure FileMR : MAP_REDUCE
```

The idea is that `'a FileMR.mapreducable` reads values of type `'a` from a file, and the `mapreduce` function does a map-reduce on all of the values in a file.

For the remainder of the homework, we will assume that there is a

```
structure MR : MAP_REDUCE
```

defining a mapreducable type, which will be either `SeqMR` or `FileMR` from above, depending on which `CM` file is used to compile the project. You should use `MR.mapreduce` to refer to the mapreduce function for either sequences or files in your code. Like the two plane implementations in Barnes Hut, this will allow you to test the code using both sequences (for small tests written in the SML files) and using data read from a file.

## 1.2 Task

Your job is to implement the extract-combine pattern as a function with the following type (see `extractcombine.sig`):

```
signature EXTRACT_COMBINE =
sig

    val extractcombine :   ('k * 'k -> order)
                         * ('a -> ('k * 'v) Seq.seq)
                         * ('v * 'v -> 'v)
                         * 'a MR.mapreducable
                         -> ('k,'v) Dict.dict

end
```

**Task 1.1** (0 pts). Before the support code will load, you need to copy your solutions from HW9 into the appropriate spots in `dict.sml`, which is an implementation of dictionaries with some extra operations that you might find useful. If you didn't finish HW09, you can contact me for a reference implementation.

**Task 1.2** (25 pts). Implement this module in `extractcombine.sml`.

This should be an abstraction of your `frequencies` solution from HW9.

You do not need to write tests for `extractcombine`. You can test by running the word frequency counting code in `wordfreq.sml` after loading the `CM` file `sources-ec.cm`. For example:

```
- CM.make "sources-ec.cm";
- WordFreq.test();
```

**Task 1.3** (0 pts). Make sure `CM.make "sources-ec-file.cm";` also loads without errors — otherwise you are assuming that the data source `MR` is a sequence, and your extract-combine code won't also work for files. If it doesn't, rewrite your `extractcombine` so that the only operation it applies to the `'a MR.mapreduceable` is `MR.mapreduce`.

# 2 Text Classification

**See the section on testing below for which CM files to use to compile your code for this part.**

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").[2] 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.

For this assignment, we will be using a collection of Reuters news articles from the late 1990s that have been labeled with one of the above four categories.[3].

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

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

```
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...
ECAT      Nothing is sacred, it appears, in the British Treasury's...
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:

- 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 training documents:

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

- 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 (because log turns products into sums) 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

$$\ln \left( \frac{1}{\text{total number of unique words in all training documents}} \right)$$

Overall, we have the following algorithm:

1. Training: from the training examples, compute the quantities used above.
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.

## 2.2 Task

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

```
structure NaiveBayes :> NAIVE_BAYES_CLASSIFIER
```

where the signature `NAIVE_BAYES_CLASSIFIER` is in Figure 2.2.3.

The real exports of your classifier are:

```
signature CLASSIFIER =
sig

    type category = string

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

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

A category is just a string, a document is a sequence of words, and a `labeled_document` is a `document` (sequence of words) together with a category. 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 will retype the classifier with the `CLASSIFIER` signature given above to hide the helper functions.

### 2.2.1 Training

From the training data, you need to extract the following information:

1. for each category, the number of documents in that category
2. for each category, the total number of words in that category (counting duplicates)
3. for each category and word, the number of times that word occurs in documents with that category
4. the sequence of all categories
5. the total number of classified documents
6. the total number of *distinct* words used in all documents (i.e. don't count duplicates).

You will probably want to use multiple extract-combines to do this (my solution uses four or five).

**Task 2.1** (30 pts). Use `ExtractCombine.extractcombine` to define a function

```
type statistics =
      (category,int) Dict.dict
    * (category,int) Dict.dict
    * (category * string, int) Dict.dict
    * category Seq.seq
    * int
    * int

val gather : labeled_document MR.mapreducable -> statistics
```

Feel free to also use any functions from the `DICT` and `SEQUENCE` signatures.

### 2.2.2 Classifying

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

**Task 2.2** (20 pts). First, compute the log-probabilities of each category. Define a function

```
val possible_classifications : statistics * 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 2.3** (10 pts). Next, compute the best (maximum) category:

```
val classify : statistics * document -> category * real
```

The function can return the empty string and `Real.negInf` (negative infinity) if there are no possible classifications.

### 2.2.3 Putting it all together

**Task 2.4** (10 pts). Finally, define

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

**For efficiency, it is 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 function-that-returns-a-function that we could not do if the function took in a pair `labeled_document MR.mapreducable * document` as input. 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

    type category = string

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

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


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

    type statistics =
          (category,int) Dict.dict
        * (category,int) Dict.dict
        * (category * string, int) Dict.dict
        * category Seq.seq
        * int
        * int

    val gather : labeled_document MR.mapreducable -> statistics

    val possible_classifications : statistics * document -> (category * real) Seq.seq
    val classify : statistics * document -> category * real

end
```

Figure 1: Classifier signature

## 2.3 Testing

**Testing Each Function For these tests, you should compile your code with**

```
- CM.make "sources-classify-seq.cm";
```

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

```
val print_stats          : labeled_document MR.mapreducable -> unit

val print_stats_nofreqs  : labeled_document MR.mapreducable -> unit

val print_possibles      : labeled_document MR.mapreducable * labeled_document -> unit

val number_correct       : labeled_document MR.mapreducable
                         * labeled_document MR.mapreducable
                         -> int * int

val print_predictions    : labeled_document MR.mapreducable
                         * labeled_document MR.mapreducable
                         -> unit
```

The outputs look like this:

- Print stats

```
- TestSeq.print_stats TestSeq.dups_train;
Number of documents by category:
  ECAT 1
  GCAT 1
Number of words by category:
  ECAT 4
  GCAT 4
Frequencies:
  ECAT: price occured 2 times
  ECAT: stock occured 2 times
  GCAT: congress occured 2 times
  GCAT: court occured 2 times
All categories: ECAT GCAT
Total number of documents:2
Total number of distinct words:4
```

- Print possibles:

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

This shows the given category 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 the given category (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 category (what it was labeled with), the predicted category, and the document.

- Number correct:

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

For each document in a sequence, run the classifier, and compute the pair (number correctly classified, total number of documents).

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

**Task 2.5** (0 pts). The files

```
expected-outputs/stats.txt
expected-outputs/possibles.txt
expected-outputs/predictions.txt
expected-outputs/number-correct.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** **For these tests, you should compile your code with**

```
- CM.make "sources-classify-file.cm";
```

Now, you should run your classifier on some real data. Download `hw10data.tar` and unzip it to create a `data` directory inside your `hw10-handout` folder. We have provided files of three sizes:

```
Num docs   File

      65   RCV1.small_train.txt
       8   RCV1.small_test.txt

    7356   RCV1.medium_train.txt
     808   RCV1.medium_test.txt

   72398   RCV1.big_train.txt
   78899   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 10–20 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 2.6** (5 pts). In a comment at the bottom of `classify.sml`, report the percentage your classifier gets correct for each of the three sizes (small, medium, large). You can also use training/test of mismatched sizes to investigate how much additional training data helps—e.g. does using the big training data improve results on the medium test?