Text Generation using a Markov Chain ⋆, ⋆⋆

Topics

- Strings, Lists, Hash tables, Arrays
- A little bit of Format and Arg

Introduction

The goal of this exercise suite is to decompose an existing text corpus, in order to recompose sentences by randomly collating original word sequences with the same appearance probability.

For this, we will use a markov chain style method. We will construct a markov chain from an input text, whose states are the words, and transitions are quantified by the probability for a word to be the predecessor of another.

For instance, if we examine *"I am a man and my dog is a good dog and a good dog makes a good man."*, delimiting it with *"START"* and *"STOP"* to identify proper beginning and end of sentences, we end up with the following chain, that can then be walked to generate new text that ressembles the input.

"START" → *"I"* (100%)
"I" → *"am"* (100%)
"am" → *"a"* (100%)
"a" → *"man"* (25%), *"good"* (75%)
"man" → *"and"* (50%), *"STOP"* (50%)
"and" → *"my"* (33%), *"a"* (67%)
"my" → *"dog"* (100%)
"dog" → *"is"* (33%), *"and"* (33%), *"makes"* (34%)
"good" → *"dog"* (66%), *"man"* (34%)
"is" → *"a"* (100%)
"makes" → *"a"* (100%)

Exercise 1 – Basic Version

Our first goal will be to build such a table and generate sentences from it, quick and dirty style, using lists and their predefined operators. Below is an example output.

```
0: START I am a good man STOP
1: START I am a good dog is a good dog and my dog and my dog is a man and
   my dog and a man STOP
2: START I am a good dog is a man and my dog makes a good man STOP
3: START I am a good dog makes a good dog is a good dog and a good dog
   makes a good dog is a man STOP
4: START I am a good dog and a man and a good dog and a good man and a good
   dog is a good dog is a good man and a man STOP
5: START I am a good dog and a good dog and my dog is a man STOP
6: START I am a man STOP
7: START I am a good dog is a good dog is a good dog and my dog is a man
   STOP
8: START I am a good man STOP
9: START I am a good dog makes a good dog and a good dog is a good dog is a
```

```
    good man and my dog is a good dog and my dog and a good man and a good
    dog is a good man STOP
10: START I am a man and my dog and my dog is a good dog and a good dog
    makes a man and my dog is a good man and my dog and my dog and my dog
    makes a man STOP
```

Question 1.1 – Write a function `words_of_file : string -> string list` that takes a file name and returns the list of its words. As a first approximation, you can consider roman letter sequences as words and everything else as separators.

Question 1.2 – Write `build_table : string list -> (string * string list) list` that associates each word present in the input text to all its possible successors (so that we can access the successors using `List.assoc`). If a successor appears several times, simply add all the duplicates, so that we respect the probability distribution implicitly.

Question 1.3 – Write the random generation function `walk : (string * string list) list -> string -> string -> string list` which takes a table, a given starting word and stops the generation when a given final word is encountered. We assume that the request is well formed for the table, meaning that there is always a path in the table between the given start and stop.

Question 1.4 – Write a small main program that takes an input text file as parameter and generates and prints 20 sentences. In order to ensure consistency, wrap the input text with fake tokens *"START"* and *"STOP"*, that you also pass to the `walk` function.

You can use the almighty `Format` module to pretty print the text as in the example output.

Exercise 2 – Performance Improvements

The previous prototype was quick to write, but does not scale. Let us improve it so that we can index a full book, or even several ones.

Question 2.1 – If we want to input a corpus bigger than a few dozens of lines, we have to build the table with a more appropriate data structure than an associative list. Update the `build_table` function so it uses a hash table internally, without changing its type. You can use `Hashtbl.fold` to dump the internal table to a list. Experience the performance boost by running the generator before and after on some big ebook.

Question 2.2 – Now, generation should be the bottleneck. You can enhance the performance doing two things: (1) dropping the associative list completely and (2) using arrays for successors instead of lists.

In order to keep all those things hidden, and to leave space for potential future improvements, we will hide all the implementation behind a function type.

Update `build_table` so that it returns a lookup function of type `(string -> string)` that does the searching and the random choosing.

Question 2.3 – Update `walk` accordingly.

Exercise 3 – Quality Improvements

When given books as input, our generator generates a lot of garbage. We will now enhance it to use the structure of its input in order to filter the results.

Question 3.1 – First, let us improve our trivial input splitting function. Update `words_of_file` to return a list of sentences instead of a single big succession of words.

A good first approximation is to consider end-of-word dots and skipped lines as sentence terminators, and capitals just following dots as sentence starters.

Question 3.2 – In order to modify as few things as possible, simply collate all sentences using the *"START"* and *"STOP"* tokens to separate them, before calling `build_table`.

Question 3.3 – Another way to enhance the output is to help our brain filter the results. An idea is to colorize the words with respect to the input text they have been extracted from.

Update the main program to take a list of inputs, and instead of simply building a big list of words to pass to `build_table`, build a list of pairs, associating each word to a number corresponding to the order of its originating source on the command line. You can associate files starting from 1, reserving 0 for the fake *"START"* and *"STOP"* tokens.

Question 3.4 – Update `build_table` so that the returned lookup function returns a pair of the selected successor word and its origination. The selection in the table must not take in consideration the color, so the lookup function still only takes a string as input.

Question 3.5 – Use these numbers in the main program to print ANSI escape sequences. In OCaml, escaped character codes are in decimal, so you must use `\027` to start escape sequences.

Exercise 4 – Prefix Matching

A drastical way of improving the results is, instead of considering the possible successors of each word, to consider the ones of each prefix of a given length *N*.

These examples have been generated from a mix of Arsène Lupin, a collection of sandwich recipes, and the OCaml Manual, using this algorithm with a prefix length set to 2.

- *And then, was she impressed with the g option to ocamlc during the past few years.*
- *No sound could be freely redistributed.*
- *The file currently being parsed by the compiler will be your humble servant.*
- *Devanne exclaimed with much gusto: Ah monsieur, you can observe the entire loaf, butter each slice a layer of the greatest integer less than or equal than the usual slices for sandwiches.*
- *The buttered side of the constrained signature.*
- *Dead Dead he repeated, to himself; they are carefully cooked, rejecting the skins; work this together.*
- *Sherlock Holmes is the source code for a moment.*
- *Readers unfamiliar with lex and yacc can report that expression of intense agony.*

Question 4.1 – Write `split_prefix : int -> (string * int) list -> string list * (string * int)` That takes an input list of colored tokens, and returns the uncolored prefix of the given length *N* and its successor token. The function should raise `Not_found` when the end of list is reached.

Question 4.2 – Update `build_table` to take the prefix length *N* and return a lookup function that takes a prefix of type `string list` as input. Use `split_prefix` to deconstruct the list until it raises `Not_found`.

Question 4.3 – Update `walk` so that is takes an input prefix sequence of length *N* instead of a single word, and stops when the accumulated sentence ends with a stop sequence of length *N*.

Question 4.4 – Instead of *"START"* and *"STOP"*, we will simply use repeated sequences of length N of these same tokens. Write a utility `repeat n elt` that builds a list containing `n` times `elt`, and update the main program.

Question 4.5 – Use the `Arg` module to parse a `-length` option to set the prefix length.

Text Generation using a Markov Chain

OLUTIONS – SOLUTIONS – SOLUTIONS – SOLUTION

Solution to question 1.1

One in functional style:

```
let words_of_file fname =
  let isletter c = match c with
    | '0'-'9' | 'a'..'z' | 'A'..'Z' | '\128'..'\255' -> true
    | _ -> false in
  let rec read fp acc =
    match input_line fp with
    | line ->
      let acc = split line acc 0 0 in
      read fp acc
    | exception End_of_file ->
      List.rev acc
  and split line acc s i =
    if i < String.length line && isletter line.[i] then
      split line acc s (succ i)
    else if s < String.length line then
      let acc =
        if s <> i then
          String.sub line s (i - s) :: acc
        else acc in
      split line acc (succ i) (succ i)
    else acc in
  let fp = open_in fname in
  let words = read fp [] in
  close_in fp ;
  words
```

One in imperative style:

```
let words_of_file fname =
  let fp = open_in fname in
  let words = ref [] in
  (try
     while true do
       let line = input_line fp in
       let rec eat s i =
         if i < String.length line then
           match line.[i] with
           | '0'..'9' | 'a'..'z' | 'A'..'Z' | '\128'..'\255'-> eat s (succ i)
           | _ -> cut s i
         else cut s i
       and cut s i =
         if s < String.length line then begin
           if s <> i then
```

```
            words := String.sub line s (i - s) :: !words ;
          eat (succ i) (succ i)
        end
      in eat 0 0
    done
  with End_of_file -> ()) ;
  close_in fp ;
  List.rev !words
```

Solution to question 1.2

```
let build_table phrase =
  let update table word next =
    match table with
    | [] -> [ (word, [ next ]) ]
    | (w, l) :: ws ->
      if w = word then
        (w, next :: l) :: ws
      else
        (w, l) :: update ws word next in
  let rec build phrase =
    match phrase with
    | [] | [ _ ] -> []
    | word :: (next ::_ as rest) ->
      let table = build rest in
      update table word next in
  build phrase
```

Solution to question 1.3

```
let walk table start stop =
  let rec walk index =
    if index = stop then
      [ stop ]
    else
      let choices = List.assoc index table in
      let next = List.nth choices (Random.int (List.length choices)) in
      index :: walk next in
  walk start
```

Solution to question 1.4

```
let () =
  Random.self_init () ;
  match Array.to_list Sys.argv with
  | [ exe ; file ] ->
    let start = "START" and stop = "STOP" in
    let words = [ start ] @ words_of_file file @ [ stop ] in
    let table = build_table words in
    for i = 0 to 20 do
      let sentence = walk table start stop in
      Format.printf "%␣3d:␣@[" i ;
      List.iter (Format.printf "%s@␣") sentence ;
      Format.printf "@]@."
    done
```

```
  | exe :: _ ->
    Format.printf "Usage: %s <input text file>\n" exe ;
    exit 1
  | [] ->
    exit 1
```

Solution to question 2.1

```
let build_table phrase =
  let table = Hashtbl.create 1000 in
  let update word next =
    match Hashtbl.find table word with
    | succs -> succs := next :: !succs
    | exception Not_found -> Hashtbl.add table word (ref [ next ]) in
  let rec build phrase =
    match phrase with
    | [] | [ _ ] -> ()
    | word :: (next ::_ as rest) ->
      update word next ;
      build rest in
  build phrase ;
  Hashtbl.fold (fun word succs r -> (word, !succs) :: r) table []
```

Solution to question 2.2

```
let build_table phrase =
  let table = Hashtbl.create 1000 in
  let update word next =
    match Hashtbl.find table word with
    | succs -> succs := next :: !succs
    | exception Not_found -> Hashtbl.add table word (ref [ next ]) in
  let rec build phrase =
    match phrase with
    | [] | [ _ ] -> ()
    | word :: (next ::_ as rest) ->
      update word next ;
      build rest in
  build phrase ;
  let final_table = Hashtbl.create 1000 in
  Hashtbl.iter
    (fun word succs -> Hashtbl.add final_table word (Array.of_list !succs))
    table ;
  let lookup word =
    let succs = Hashtbl.find final_table word in
    succs.(Random.int (Array.length succs)) in
  lookup
```

Solution to question 2.3

```
let walk table start stop =
  let rec walk index =
    if index = stop then
      [ stop ]
    else
      let next = table index in
```

```
      index :: walk next in
  walk start
```

Solution to question 3.1

```
let words_of_file fname =
  let isletter c = match c with
    | '0'..'9' | 'a'..'z' | 'A'..'Z' | '\128'..'\255'
    | '.' | '\'' | ';' | ',' | '-' | '_' | ':'
    | '(' | ')' | '[' | ']' | '{' | '}' -> true
    | _ -> false in
  let empty =
    ([], []) in
  let grab (cur, acc) =
    if cur = [] then acc else List.rev cur :: acc in
  let push word (cur, acc) =
    match word.[0], word.[String.length word - 1], cur with
    | _, '.', _ -> ([], grab (word :: cur, acc))
    | 'A'..'Z', _, [] -> ([ word ], acc)
    | _, _, [] -> ([], acc)
    | _ -> (word :: cur, acc) in
  let rec read fp acc =
    match input_line fp with
    | line ->
      let acc = split line acc 0 0 in
      read fp acc
    | exception End_of_file ->
      grab acc
  and split line acc s i =
    if i < String.length line && isletter line.[i] then
      split line acc s (succ i)
    else if s < String.length line then
      let acc =
        if s <> i then
          push (String.sub line s (i - s)) acc
        else acc in
      split line acc (succ i) (succ i)
    else acc in
  let fp = open_in fname in
  let sentences = read fp empty in
  close_in fp ;
  sentences
```

Solution to question 3.2

```
...
  | [ exe ; file ] ->
    let start = "START" and stop = "STOP" in
    let words =
      List.fold_right
        (fun words acc -> [ start ] @ words @ [ stop ] @ acc)
        (words_of_file file) [] in
    let table = build_table words in
...
```

Solution to question 3.3

```
...
  | exe :: files ->
    let start = "START", 0 and stop = "STOP", 0  in
    let words =
      List.mapi (fun i file ->
          List.fold_right
            (fun words acc ->
               let words = List.map (fun w -> w, succ i) words in
               [ start ] @ words @ [ stop ] @ acc)
            (words_of_file file) [])
        files
      |> List.flatten in
    let table = build_table words in
...
```

Solution to question 3.4

```
let walk table start stop =
  let rec walk index =
    if index = stop then
      [ stop ]
    else
      let next = table (fst index) in
      index :: walk next in
  walk start
```

Solution to question 3.5

```
...
    for i = 0 to 20 do
      let sentence = walk table start stop in
      Format.printf "%␣3d:␣@[" i ;
      List.iter (fun (word, color) ->
          let color_es = Format.asprintf "\027[%dm" (30 + color) in
          let reset_es = "\027[0m" in
          Format.printf "@<0>%s%s@<0>%s@␣" color_es word reset_es)
        sentence ;
      Format.printf "@]@."
    done
...
```

Solution to question 4.1

```
let split_prefix length l =
  let rec split_prefix n acc l = match (n, l) with
    | 0, next :: _ -> List.rev acc, next
    | _, (next, _) :: rest -> split_prefix (pred n) (next :: acc) rest
    | _ -> raise Not_found
  in split_prefix length [] l
```

Solution to question 4.2

```
let build_table length phrase =
  let table = Hashtbl.create 1000 in
```

```
  let update word next =
    match Hashtbl.find table word with
    | succs -> succs := next :: !succs
    | exception Not_found -> Hashtbl.add table word (ref [ next ]) in
  let rec build phrase =
    match phrase with
    | [] | [ _ ] -> ()
    | _ :: rest ->
      match split_prefix length phrase with
      | prefix, next ->
        update prefix next ;
        build rest
      | exception Not_found -> () in
  build phrase ;
  let final_table = Hashtbl.create 1000 in
  Hashtbl.iter
    (fun word succs -> Hashtbl.add final_table word (Array.of_list !succs))
    table ;
  let lookup word =
    let succs = Hashtbl.find final_table word in
    succs.(Random.int (Array.length succs)) in
  lookup
```

Solution to question 4.3

```
let walk table start stop =
  let roll l w =
    List.tl l @ [ w ] in
  let rec walk index =
    if index = stop then
      []
    else
      let next = table (List.map fst index) in
      next :: walk (roll index next) in
  start @ walk start
```

Solution to question 4.4

```
let () =
  Random.self_init () ;
  let length = 2 in
  match Array.to_list Sys.argv with
  | [ exe ] ->
    Format.printf "Usage: %s <input text file>\n" exe ;
    exit 1
  | exe :: files ->
    let rec repeat n elt =
      if n = 0 then [] else elt :: repeat (pred n) elt in
    let start = repeat length ("START", 0)
    and stop = repeat length ("STOP", 0 )in
    let words =
      List.mapi (fun i file ->
          List.fold_right
            (fun words acc ->
               let words = List.map (fun w -> w, succ i) words in
```

```
               start @ words @ stop @ acc)
           (words_of_file file) [])
       files
     |> List.flatten in
   let table = build_table length words in
...
```

Solution to question 4.5

```
let () =
  Random.self_init () ;
  let length = ref 2 in
  let rem = ref [] in
  let exe = Filename.basename Sys.executable_name in
  let usage = exe ^ " [options] <text file> <text file> ..." in
  Arg.parse
    [ "-length", Arg.Set_int length, "set the length of the prefix" ]
    (fun arg -> rem := arg :: !rem)
    usage;
  match List.rev !rem with
  | [] ->
    Format.printf "Usage: %s\n" usage ;
    exit 1
  | files ->
    let rec repeat n elt =
      if n = 0 then [] else elt :: repeat (pred n) elt in
    let start = repeat !length ("START", 0)
    and stop = repeat !length ("STOP", 0 )in
    let words =
      List.mapi (fun i file ->
          List.fold_right
            (fun words acc ->
               let words = List.map (fun w -> w, succ i) words in
               start @ words @ stop @ acc)
            (words_of_file file) [])
        files
      |> List.flatten in
    let table = build_table !length words in
    for i = 0 to 20 do
      let sentence = walk table start stop in
      Format.printf "%3d: @[" i ;
      List.iter (fun (word, color) ->
          if color <> 0 then
            let color_es = Format.asprintf "\027[%dm" (30 + color) in
            let reset_es = "\027[0m" in
            Format.printf "@<0>%s%s@<0>%s@ " color_es word reset_es)
        sentence ;
      Format.printf "@]@."
    done
```