

o[B]abble Technical Report

George Moe, William Drew, Injil Muhammad
CS 51
Harvard College

May 2018

1 Abstract

o[B]abble is a Markov chain chatbot that can emulate the personality of a text by training on the structure of the corpus. It is implemented in a multi-agent chatroom that features conversations between bots. First, we examine the design of o[B]abble and its engineering considerations before giving detail on its key algorithms and analyzing the efficacy of the final product. We close with a reflection of the project and some proposals for future work.

2 Code Overview

o[B]abble is comprised of five key components:

1. **token.ml** - Defines a token datatype and provides tools for working with tokens
2. **markov.ml** - Defines a Markov chain datatype and provides tools for working with Markov chain objects
3. **parser.ml** - Provides a utility that parses input text files for training Markov chains on corpuses

4. **model.ml** - Defines a Markov model object and provides functions for training Markov chain models and generating responses
5. **obabble.ml** & **obabble_chatroom.ml** - Environments to interact with Markov chain models

Figure 1 illustrates how these components connect together at a high level to make a working chatbot.

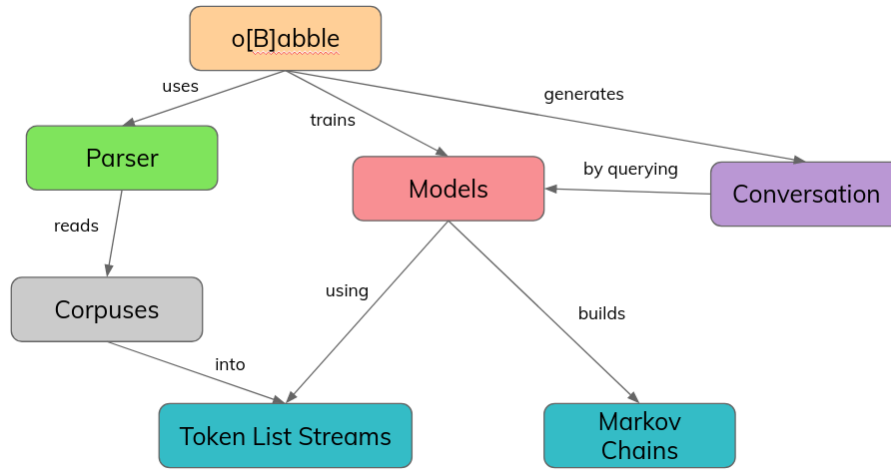


Figure 1: Web illustrating the high-level interconnection between components. Teal represents datatypes, red represents classes, green represents utilities.

2.1 token.ml

token.ml contains definitions and tools for working with tokens and converting them to and from strings.

```

type token =
  | Start
  | Word of string
  | End ;;

```

```

val tokenize : string -> string list ;;
val grammarize : token -> string ;;
val token_to_string : token -> string ;;
val token_list_to_string : token list -> string ;;
val token_list_to_string_list : token list -> string list ;;
val string_to_token_list : string -> token list ;;
val string_list_to_token_list : string list -> token list ;;

```

2.2 markov.ml

markov.ml exposes a module which defines the (abstracted) **mchain** datatype and tools to work with it.

```
module type MARKOVCHAIN =
  sig
    exception BadChain of string

    type mchain

    val empty : unit -> mchain
    val add : mchain -> token -> token -> unit
    val bake : mchain -> unit
    val token_list : mchain -> token -> token list
    val token_totals : mchain -> token -> int
    val query : mchain -> token -> token -> (int * int) option
    val roll : mchain -> token -> token option
    val size : mchain -> int * int
    val load : string -> mchain
    val save : mchain -> string -> unit
  end ;;

module MarkovChain : MARKOVCHAIN ;;

type mchain = {
  mutable baked : bool;
  chain : (token, (token_tree * (token, int) Hashtbl.t)) Hashtbl.t;
  totals : (token, int) Hashtbl.t
}
```

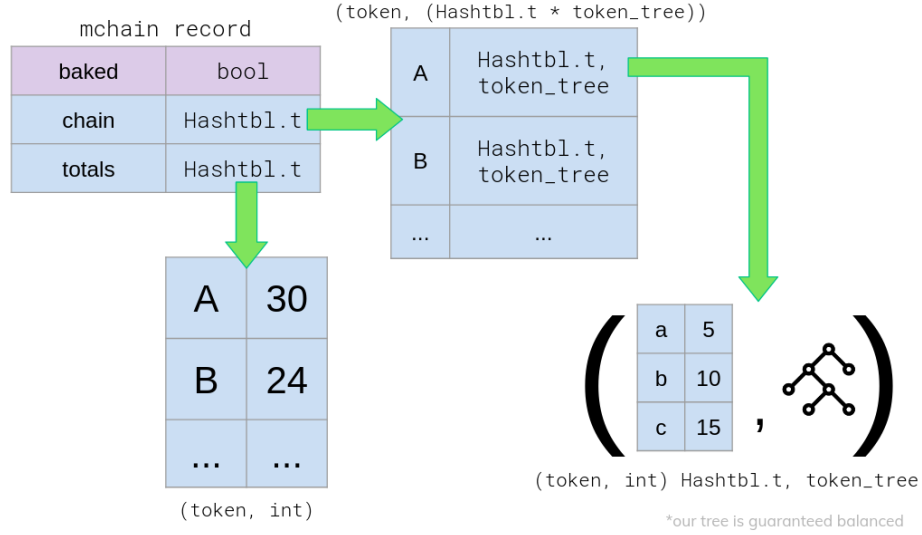
The **mchain** datatype stores the structure of the Markov chain describing the training corpus. It keeps counts of the occurrences of token $a \rightarrow b$ transitions, as well as the total counts for a , allowing for the probability of token $a \rightarrow b$ transition to be calculated by dividing the occurrence count over the total count. It also stores a token lookup tree that is baked after training and used to support fast and true probability token selections needed by text generation. This is described in further detail in the next section.

Figure 2 provides a visualization of the **mchain** datatype.

2.3 parser.ml

parser.ml provides a utility to parse raw training data from a file into a **token list Stream.t**. This token list stream is used in **model.ml** to train the Markov chain models.

```
val get_stream : string -> token list Stream.t
```

Figure 2: Illustration of the `mchain` datatype.

2.4 model.ml

`model.ml` defines the `model` class which represents a chatbot. It exposes methods which allow the chatbot's Markov chain models to be trained using parser token list streams. It also exposes methods to interact with the chatbot to generate output, as well as to save and load the chatbot to and from file.

This was designed to be a class in order to elegantly support our multi-agent chatroom; with a `model` class, we can simply instantiate a model for each desired chatbot and train each model with a corpus corresponding to the chatbot's target personality.

```
type model_t = {chains : mchain; assoc : mchain}  ;;
```

```
class type model_class_t =
  object
    method name : string
    method train : int -> token list Stream.t -> unit
    method save : string -> unit
    method load : string -> bool
    method chains : mchain
    method assoc : mchain
    method query : token list -> int -> int -> float -> token list option
    method set_debug : bool -> unit
  end  ;;
```

```
class model : string -> model_class_t  ;;
```

2.5 obabble.ml and obabble_classroom.ml

These are the main frameworks within which interactions with the chatbots take place. They set up the environment for the chatbots as well handle user-experience and updating the environment in each run-loop.

3 Algorithm Detail

3.1 True Probability Rolls

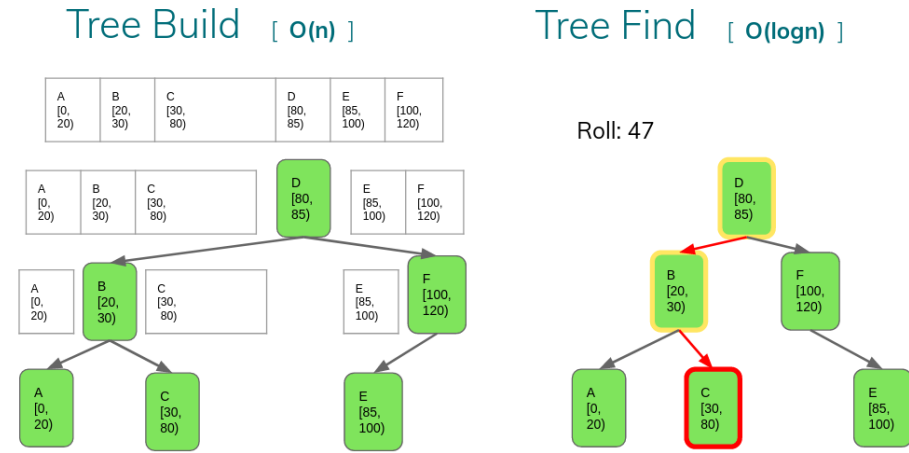


Figure 3: Illustration of token lookup trees: how they are built in a way that guarantees balance, and how they are queried quickly while remaining true to probabilities. The numbers below the token name represent the interval of the token, whose size is proportional to the token’s probability.

In order to randomly generate sentences, we need a way to get a random token with weighted probability, given a seed token. This is accomplished by assigning tokens intervals on a number line, rolling a random index, and selecting the token that contains that index.

To implement this efficiently, we create a balanced binary search tree with nodes representing tokens and their intervals. Random token fetch is accomplished by searching the tree for a token whose interval contains the index. Because this tree is guaranteed to be balanced, this takes $O(\log n)$ time. This process is illustrated on the right side of Figure 3.

Balance can be guaranteed by the way the tree is generated: given a list of tokens (unweighted, with each token appearing exactly once), we select the middle token to be the head of the tree. This ensures that both halves of the tree will have equal numbers of nodes up to a difference of 1 node. This head selection repeats recursively for both halves, guaranteeing that no subtree will

more than one node than its complement subtree. In this way, the entire tree is balanced. The left side of Figure 3 illustrates this process.

3.2 Ranking Responses

$$R(i, k) = \frac{f_a c_i, a_k}{\log t_a(c_i)} \cdot \frac{1}{\log(t_a(a_k) - f_a(a_k, c_i))} \quad (1)$$

$$K(i) = \frac{f_c(a_i, a_{i+1})}{\log t_c(a_i)} \quad (2)$$

$$S = \frac{\sum R(i, k)}{\log |\vec{c}|} \cdot \frac{\sum K(i)}{|\vec{a}|} \quad (3)$$

Figure 4: Equations used in scoring heuristic. $f_a(a, b)$ and $f_c(a, b)$ give the counts for the transition $a \rightarrow b$ in the association mchain and word chain mchain, respectively, and $t_a(a)$ and $t_c(a)$ give the total appearances of token a in the respective mchains.

Our goal in this project is to have the chatbot closely model true conversations, so came up with an algorithm to score different response candidates. We scored keeping in mind two axes: relevance (how related a response is to what was previously said) and coherence (do the string of words produced when read in order sound like actual speech).

To find relevance, we based our algorithm on TF-IDF, or term frequency inverse document frequency, but took it a step further. When we look at a term for context, we see how often this term appears in the same sentence as an “answer” term, and then we divide this number by the logarithm of the number of times this context term shows up overall. We divide by the logarithm of the number of times this context term appears overall to penalize answers that are used frequently overall more than words that are used frequently in tandem with another particular context word, but not too much as to discount very common words from showing up in responses. They are common words for a reason. Essentially, we were trying to create as precise a method as possible for delineating how to select some words over other to craft the most relevant strings of tokens possible.

To calculate the coherence of phrases, we look only at the string of answer terms being put together. Essentially, we look at the frequency of one answer term appearing in the same sentence with another answer term, and as we mentioned earlier, we are going from word to word. To normalize the frequency of one word appearing with another, we divide by the logarithm of the amount of time that word occurs overall, again because we want to penalize words that are used frequently overall more than words that are used frequently in tandem with another particular word.

An important thing to note is we tested all of these algorithms empirically to see which one produced the responses which sounded the most coherent and the most relevant, so trial and error was a huge part of arriving at these idealized algorithms to model true conversations.

Our final score consists of our relevance score divided by the log of the length of the context phrase multiplied by the coherence score divided by length of the answer phrase. We do this to ensure that both coherence and relevance are attributes that are measured per word and not related to the length of any one of the responses. The product of relevance and coherence produces a score that helps us select responses in a way that models actual conversations.

3.3 Multi-Agent Interaction

Multi-Agent Interaction is enabled by **models as classes** and **multithreading**. Because our models are classes, we can quickly instantiate different chatbots trained with different personalities. These are run in their own threads in parallel so that they can converse with each other. Whatever they say gets appended to the conversation history, which is used to rank future responses to be relevant to the conversation.

In order to ensure that there are no race conditions (e.g. interleaved output, interleaved history) each thread must acquire a single mutex to update the history and write to the output.

4 Corpus Detail

This corpus contains a large metadata-rich collection of fictional conversations extracted from raw movie scripts:¹

- 220,579 conversational exchanges between 10,292 pairs of movie characters
- involves 9,035 characters from 617 movies
- in total 304,713 utterances
- movie metadata included:
 - genres
 - release year
 - IMDB rating
 - number of IMDB votes
 - IMDB rating
- character metadata included:
 - gender (for 3,774 characters)
 - position on movie credits (3,321 characters)

¹*taken from http://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html

5 Results

The single-agent **obabble.ml** interface was satisfactorily capable of supporting back-and-forth-style conversation between the user and chatbot. Although largely nonsensical, the chatbot’s responses to the user’s queries were often acceptable, and more importantly, interesting. Figure 5 shows an image of a sample conversation with the chatbot.



```
Begin a conversation:
I: hello friends!
I> things.

I: What is your favorite fruit?
I> dr.

I: what's that?
I> yes, honey! before you turned you.

I: what?
I> milo.

I: hello?
I> here.

I: bye?
I> yes.

I: ok
I> what?

I: why?
I> i want?

I: computer science is fun!
I> yeah.
```

Figure 5: Sample one-on-one conversation with the single chatbot in **obabble.ml**.

Evidently, it seems that the chatbot favors short responses, which is not quite in line with the average expected response in the movie line corpus. We find that this is due to a overly-aggressive length penalty in the computation of the coherence subscore. Changing this from a linear penalty to a log penalty reduces the aggressiveness due to length and creates much more colorful responses.

obabble_chatroom.ml successfully placed multiple chatbots in conversations with one another as well as the user. Although once again nonsensical, the responses correctly mimicked an English-like appearance. Additionally, due to the distinct personalities of the chatbots, many of the phrases were outright hilarious. Figure 6 shows a sample conversation in the multi-agent chatroom.

Note that the response lengths are longer the figure. For this sample, we modified the scoring heuristic to use only a log-length penalty as described, as well as increased the upper-limit on generation length.


```

[YOU]> hello all my friends!
[trump]> "as usual mr. - cleveland, and more fake news nbc just saw photos of the trans-pacific partners
hip is a failed us in 2016. """,
[tinman]> oh, a palpitation.
[catwoman]> at night. i realize there's nothing you three lives together. i'm really angry.
[hal]> well, but not at the base, and burn them off to do a great pleasure to trouble.
[magneto]> and they will return home -- they will return as a test. every word i wouldn't really have sp
oken will provide for?
[terminator]> we are targeted for success now why you at 2:14 a geometric rate. no. guns and explosives
have chemicals, john.
[shieber]> "nearexclusive association, you are now this, az in american institutions endorse the cited s
tudy computer chess. libraries to be it on #liblab and @klsmith4906 yes it.
[shieber]> @hgmorrison @ktkeith @oajoe $60 is not an #oa policies guide ... which a model oa jrnal = $900
. that's the presidency, using latex.
[catwoman]> y'mean frame him into what, fake who you think you're quick. it's too high for my, lately. t
rying to you lost.
[trump]> "success is doing, and i'll show you! ",
[magneto]> enough for me. every word i could make some radical changes. i know. still trying to change t
he poor soul who comes to do.
[trump]> ""@bubble709 : @realdonaldtrump on 9/15 is disheartened. ", rather @realdonaldtrump he was no
one of pelosi and love the civil war terrorism. ",
[hal]> thank you very pleasant. i can and all i have been disconnected in the council.
[tinman]> oh, dear, oh!
[terminator]> it can form complex machines. i'm a metal. the future somehow. don't have chemicals, but i
t must be back.

```

Figure 6: Sample multi-agent conversation in **obabble_chatroom.ml**.

6 Conclusion

Overall, we as the o[B]abble team are very satisfied with our project's performance! In the course of our work, we learned more about language representation, designing online algorithms and data structures, and even touched upon resource-sharing in a multi-threaded environment.

Possible future extensions to our project include adding support for n-grams instead of just bi-grams, as well as incorporating larger-scale Markov chains that represent transitions between general topics or states within an overarching conversation. We feel that adding these nuances can only improve the potency of the generated text.

We find it incredible that such a succinct codebase and theory of operation can create such colorful and complex textual output. All in all, this was an incredibly fun and satisfying project, and we are pleased to present o[B]abble as a fitting capstone to our CS51 experience.