Text prediction app documentation - Model construction

Ivy Woo

17 February 2021

1 Theoretical model

A simple n-gram model is employed.

At a high level, an n-gram model predicts the next word of some text chunk based on the last $n-1$ words of the chunk, with the prediction being the one appearing next with the highest probability. That is, denote by V the space of vocabulary of the model, $w_{k-n+1}, w_{k-n+2}, \dots, w_{k-1} =: w_{k-n+1} : w_{k-1}$ the last $n-1$ words of some text chunk, W the random variable representing the next word, and $w \in V$ the next word predicted by the model, then

$$w = \arg\max_{v \in V} \mathbb{P}(W = v | w_{k-n+1} : w_{k-1}).$$

See e.g. Jurafsky and Martin (2009) for a detailed explanation on the model.

In the Shiny app a 4-gram model is used. That is, the next word is predicted by looking at the last three words (at most) of some given text chunk.

1.1 Non-existent n-grams

What happens when the app user inputs some sequence of words which does not exist in the model? This app uses an approach similar to a back-off model but with additional tweak.

Suppose we use an n-gram model to predict the k-th word. In a simple back-off model, when certain $(n-1)$-gram, say some $w_{k-n+1} : w_{k-1}$, to be looked up does not exist in the model, the first word is to be dropped so that the model next search for the $(n-2)$-gram, i.e. $w_{k-n+2} : w_{k-1}$. This process iterates until looking at the last one word w_{k-1} for prediction, if this fails then the unigram is to be looked up, that is, returning a prediction which is simply the word appearing most frequently in the dataset, without drawing information from the user's input at all.

In our model, the grams are looked up iteratively in the following order:

1. First check if the last word w_{k-1} exists in the vocabulary or not. If yes, continue with point 2; else jump to point 6.
2. Look up the complete $(n-1)$-gram $w_{k-n+1} : w_{k-1}$.
3. Replace the first word by the unknown token <unk>, and look up the tuple (<unk>,$w_{k-n+2} : w_{k-1}$).

4. Remove the first word and look up $w_{k-n+2} : w_{k-1}$.
5. Repeat points 3 and 4 until the last word w_{k-1} is looked up.
6. Replace the last word w_{k-1} by <`unk`>, then loop over points 2 to 5 again.

Points 1,3 and 6 are additional to a back-off model. The intuition is that, looking up the tuples (<`unk`>,$w_{k-n+2} : w_{k-1}$) and ($w_{k-n+2} : w_{k-1}$) are different, since the information of whether there exists a (unknown) word at the beginning can make a different to the prediction. Further, the approach here implicitly implies a different weighting is especially given to the last word w_{k-1}.

2 Implementation - dictionary tree

The principle of the text prediction app is designed to output prediction(s) of the next word as the user inputs words in the typing box. As such, minimizing online computation time is crucial to the app's usability.

In view of the huge amount of data (refer to the dataset documentation), to boost the computation efficiency, a nested dictionary in form of a tree, which we call a *dictionary tree*, is built in attempt to achieve a computational complexity of $\mathcal{O}(1)$.

In the employed dictionary tree, the root node (in the zero-th layer) has keys which correspond to the $(n-1)$-th word. Given some keys, the values, aka. the children of the root node, are other dictionaries (in the first layer). The key of the first-layer dictionaries are the set of all tuples $\{((n-k+1) : (n-2)), ((n-k+2) : (n-2)), \ldots, (n-2)\}$-th words, given that the $(n-1)$-th word is the key which leads to their parent. The children of a first-layer dictionary are list objects, in which the names of the list items are the n-th word and the values of the objects are the corresponding counts of the tuple $((n-k+1) : n)$-th words.

A 4-gram model is used in the Shiny app. we use two examples below to illustrate the structure of the dictionary tree as well as how a prediction is looked up.

Example 2.1 Suppose we use the 4-gram model and ask the app to predict the next three word of the sentence *"Today I want a bowl of"*. The 4-gram model takes only the last three words of the given text (at most), that is *"a bowl of"* here.

Now refer to Figure 1 for a sketch of an exemplary dictionary tree. To get the next word prediction, we traverse through the tree by reading the phrase "a bowl of" backwards: first read the last word "of", input this as the first key and get the first layer dictionary, then input the remaining words "a bowl" as the second key to obtain the resulting list. This list contains all the possible next words with their counts of occurrence in the dataset being sorted.

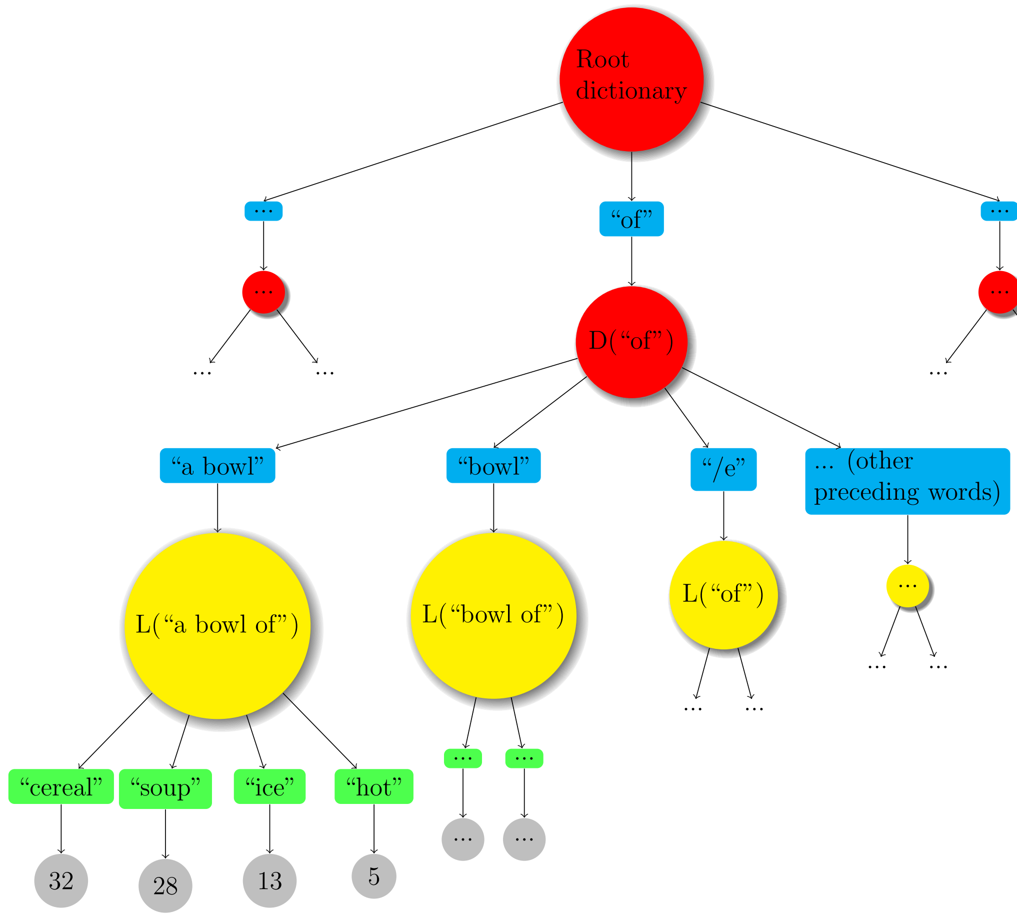

Figure 1: Sketch of an exemplary dictionary tree. Red: dictionary[D]; blue: dictionary key; yellow: list[L]; green: name of list item; grey: list item (counts).

The tree shows that there are four possibilities for the next word from the dataset. Since we want three predictions, the three with the highest number of occurrence will be output in order, which are "cereal", "soup" and "ice".

Example 2.2 Continuing from the previous example, suppose now there is no element (or less than 3 elements as asked) in the list "a bowl of". Now the app will try the following options in order, until 3 elements are collected:

1. Replace the first word in the 3-element tuple as the ⟨`unk`⟩ token and look for the corresponding list, i.e. search for the list "⟨`unk`⟩ bowl of".
2. Omit the first word, i.e. search for the list "bowl of".
3. Replace the fist word by ⟨`unk`⟩ i.e. search for the list "⟨`unk`⟩ of".

4. Omit the first word and search for the list for “of”, which is marked by the key “/e” [1] under the dictionary “of”.

5. Replace the last word by <unk> and include back the two words previously removed, that is, now search for the list “a bowl <unk>”.

6. Repeat points 1 to 4, with the last word kept being <unk>.

Using this iteration, the last list to be searched is <unk>, which will guarantee the return of some results if there is no/insufficient match from all the previous lookups, although such output will probably fit poorly to the input text.

To the best of the author’s knowledge, given the huge amount of data in this project and R’s limited support on dictionary, the dictionary tree described above is the most efficient structure that can be deployed smoothly, in the sense that

- a one-time linear scan on the dataset suffices to record all n-grams for all n’s desired due to the tree’s nested structure, with no any further processing required,
- n can be set however large as long as the machine has sufficient RAM to build the tree,
- user can demand from the tree at most as many predictions of the next word as what have presented in the dataset, and
- computation time is constant.

Remark 2.3 In the appendix we sketch another design of dictionary tree, which is a further nested version with each key being exactly just one word so that, alongside with the advantages listed above, has a much more logical structure and coding can be made more easily. Despite being structurally more desirable, this alternative can unfortunately not be built smoothly in R because of memory issues and the huge amount of data involved (unless using only a very small subset of the dataset). In case of coding in other languages which provide better support on dictionary, such as C++, this alternative would likely be preferred.

3 Deployment on Shinyapp.io

The last project requirement is to deploy the app on Shinyapp.io so that it can be accessed through internet. Currently, this project makes use of a free account, which is bound to a 1-GB RAM limitation for running the app. Therefore, it is unfortunately not possible to directly load the dictionary tree onto the platform.

[1]For empty string.

To reduce the memory requirement for running the app (with the trade-off of a longer computation time), a data table (using the `data.table` library), which is well-known for its search function with complexity $\mathcal{O}(\log n)$ through binary search base subset, is constructed to summarize the necessary information in the tree. The resulting data table has two columns: column 1 contains the n-grams with the last word removed; column 2 contains the top k predictions for the string in column 1 in the respective row, which are concatenated as one string to save space. Table 1 depicts the form of the data table with reference to the examples in the previous section.

...	...
a bowl of	cereal/soup/ice
...	...

Table 1: A sketch of the data table employed on the Shinyapp.io platform.

The final data table employed on Shinyapp.io is constructed using only 50% of the dataset. Further, all n-grams which have appeared only once in the data are removed due to their low reliability. The top $k = 5$ predictions are stored. This approach results in a data table with around 2 million rows weighting just over 200MB in R , which can be processed easily on the platform.

References

Jurafsky, D. and Martin, J. H. (2009). *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Appendix

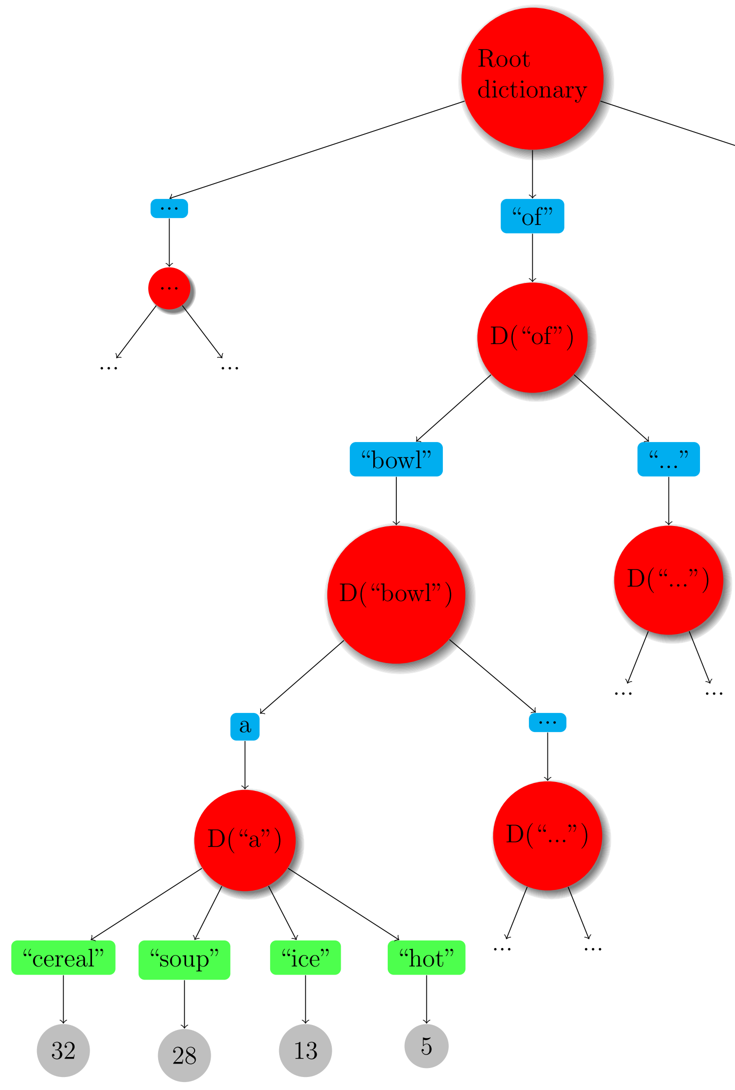

Figure 2: Sketch of an alternative dictionary tree. Red: dictionary[D]; blue: dictionary key; green: name of vector element; grey: vector element (counts).