Words prediction using Stupid Backoff algorithm

Leonardo Pinto Neves, Dario Delle Vedove

July 4, 2017

Contents

1 The problem: about words prediction

In a world where the digital communication is more and more important, technology has to help users to communicate in a fast and accurate way.
We all know that the oral and the written communications are mighty different in terms of flexibility, cost, time spent, reliability and, above all, significance. So, when we type a message on our mobile phone or on our personal computer with the keyboard, we surely spend more time and, moreover, we have to be careful that the contents of it could be interpreted in a different way with respect of that we had in mind. That is why it is important to have a system of words prediction on our devices.
More formally, Word Prediction is the problem of guessing which word is likely to continue a given initial text fragment.

A scientific approach leads us to use statistics in order to solve this problem, so we can identify words in noisy, ambiguous input and return the most likely ones.
It is a typical problem of the Natural Language Processing, that is a field of computer science concerned programming computers to analyze large natural language corpora, in order to solve problems like natural language understanding, natural language generation, connecting language and machine perception, managing human-computer dialog systems. Formerly, many language-processing tasks typically involved the direct hand coding of rules, which is not in general reliable and stable in order to look for natural language variation.
Statisitcal inference can be useful in order to learn some rules needed to allow prediction through the analysis of large corpora of typical real-world examples (a corpus is a set of documents, possibly with human or computer annotations), and assess the likelihood of various hypotheses:
• probability of word sequences;
• likelihood of words co-occurrence.

1.1 Some applications

Word prediction techniques have some immediate applications that can be easily imagined, as:
• accelerate the writing;
• reduce the effort needed to type;
• suggest the correct word with no misspellings. So, basically, we could sum up these facts by saying that Words Prediction makes easier the process of writing.
More specifically, to understand the importance of this technique, think to the Tegic T9. It is a successful system but its prediction is based on dictionary disambiguation (only according to last word). We would like something that is skilful at doing prediction according to the previous context.
Other possible applications are about:
• spelling checkers
• disabled users communication
• handwriting recognition
• word-sense disambiguation

2 Description of the data

In this project three Open Source Corpus from two different sources were used in order to built the N-grams language model:

- American National Corpus:
  - OANC: Contemporary American English
  - MASC: 19 genres of American English
- Swiftkey/Johns Hopkins University
  - Blogs
  - News
  - Twittes

Before being used, the corpus pass by a cleaning process where punctuation, number, and special chracteres are removed. The same process is also performed when a user types an input phrase. This corresponds to 2.049 files, more than 618 million words and 5 million sentences. The final corpus is a text file of 601Mb. The final corpus generate 5 different N-gram files, these data will then be used to perform the final step of the project.

Data	Size	# N-Grams
5-Grams	xxx.xx Mb	xxx.M
4-Grams	xxx.xx Mb	xxx.M
3-Grams	xxx.xx Mb	xxx.M
2-Grams	xxx.xx Mb	xxx.M
1-Grams	xxx.xx Mb	xxx.M

3 Test Environment

One of the goals of this project is to compare two different methods to generate the N-gram language model: centralized and distributed computation. To test the distributed computation it was created a *pseudo-distributed*, single-noded Hadoop cluster. The hardware used in both cases is described on the table below:

Computer Model	Asus N552VW-FY058T
Processor	i7 6700HQ 2.60GHz x 8
Memory	16Gb
O.S.	Ubuntu 16.04 LTS 64bits
Python	3.6.0
Anaconda	4.3.1 (64bits)
GNU Parallel	2.8.0
Java	1.8.0_131

4 N-Grams

Models that assign probabilities to sequences of words are called language models. The simplest model that assigns probabilities to sentences and sequences of words is the N-grams.
An N-gram is simply a sequence of N words: so, for exaple, a 2-gram (or bigram) is a two-word sequence of words like "hello how", "how are", or "are you", and a 3-gram (or trigram) is a three-word sequence of words like "hello how are", or "how are you".
We want to use N-gram models to estimate the probability of the last word of an N-gram given the previous words and to assign probabilities to entire sequences.
Let $w_1^L = (w_1, ..., w_L)$ denote a string of L tokens over a fixed vocabulary (and denote with $w_j^k = (w_j, ..., w_k)$ the substring of w_1^L from the j-th to the k-th token).
An N-gram language model assigns a probability to w_1^L according to the Bayes' rule:

$$\begin{aligned}\mathbb{P}(w_1^L) &= \mathbb{P}(w_1)\mathbb{P}(w_2|w_1)\mathbb{P}(w_3|w_1^2)...\mathbb{P}(w_L|w_1^{L-1}) \\ &= \prod_{k=1}^{L} \mathbb{P}(w_k|w_1^{k-1}) \approx \prod_{k=1}^{L} \mathbb{P}(w_k|w_{k-N+1}^{k-1})\end{aligned}$$

where in the last passage we made the Markov assumption according to which only the most recent $n-1$ tokens are relevant when predicting the next word.
The bigram model, for example, approximates the probability of a word given all the previous words $\mathbb{P}(w_n|w_1^{n-1})$ by using only the conditional probability of the preceding word $\mathbb{P}(w_n|w_{n-1})$.
The way in which we compute this probabilities in our algorithm will be explained later.

5 Stupid Backoff

The Stupid Backoff algorithm was introduced by a Google research team.
Usually, the probability assigned to a word to be predicted is

$$\mathbb{P}(w_i|w_{i-k+1}^{i-1}) = \begin{cases} \rho(w_{i-k+i}^i) & \text{if } w_{i-k+1}^i \text{ is found} \\ \\ \alpha(w_{i-k+1}^{i-1})\mathbb{P}(w_i|w_{i-k+2}^{i-1}) & \text{otherwise} \end{cases}$$

where $\rho(\cdot)$ are pre-computed probabilities and $\alpha(\cdot)$ are back-off weights.
The algorithm consists of a smoothing method that uses text-dependent backoff. The algorithm uses a sentence as input and try to find the n-gram corresponding to the last n words typed. If it finds a match, then it computes the score function showed below. In the case that the n-gram is not found, the algorithm uses the (n-1)-gram data to find the last n-1 words typed, penalizing the score by a coefficient α. The process continues until it reaches the unigrams, which corresponds to the most frequent words.
Stupid Backoff is a similar but simpler scheme that does not generate normalized probabilities. The main difference is that we use the relative frequencies (S is used instead of P to emphasize that these are not probabilities but **scores**):

$$S(w_i|w_{i-k+1}^{i-1}) = \begin{cases} \frac{f(w_{i-k+i}^i)}{f(w_{i-k+i}^{i-1})} & \text{if } f(w_{i-k+i}^i) > 0 \\ \\ \alpha S(w_i|w_{i-k+2}^{i-1}) & \text{otherwise} \end{cases}$$

where $f(\cdot)$ is the relative frequency of a k-grams with respect to all the others k-grams in the Corpus.
The scalar α is set in 0.4. This value was empirically chose by the authors.

6 Hadoop MapReduce

Hadoop is the most popular and powerful big data tool and provides world?s most reliable storage layer. It is an open-source software framework used for distributed storage and processing of dataset of big data using the MapReduce programming model. It consists of computer clusters built from commodity hardware.

MapReduce is the original framework for writing applications that process large amounts of structured and unstructured data stored in the Hadoop Distributed File System (HDFS), which divides the data into blocks and store distributedly.
As its name suggests, it is the combination of two different processing idioms: Map and Reduce. Basically, it runs following these steps:
1) One block is processed by one **mapper** at a time, which runs on all the nodes of the cluster and process the data blocks in parallel.
2) Mapper generates <key, value> pairs, partitioning the output (that is, grouping together the values with the same key) and then sorting them. In our case, keys are all the possible N-grams and the values are the counting of each of them.
3) Output of mapper is shuffled to **reducer** nodes.
4) The intermediate output is merged, sorted basing on the keys in different Mappers and finally provided as input to reduce phase (the shuffle and sort phases occur concurrently).
5) An input to a reducer is provided from all the mappers. Reducers, which run in parallel, aggregate the key value pairs, which are the final output. In our case, we will have the final amount of the N-grams counting.

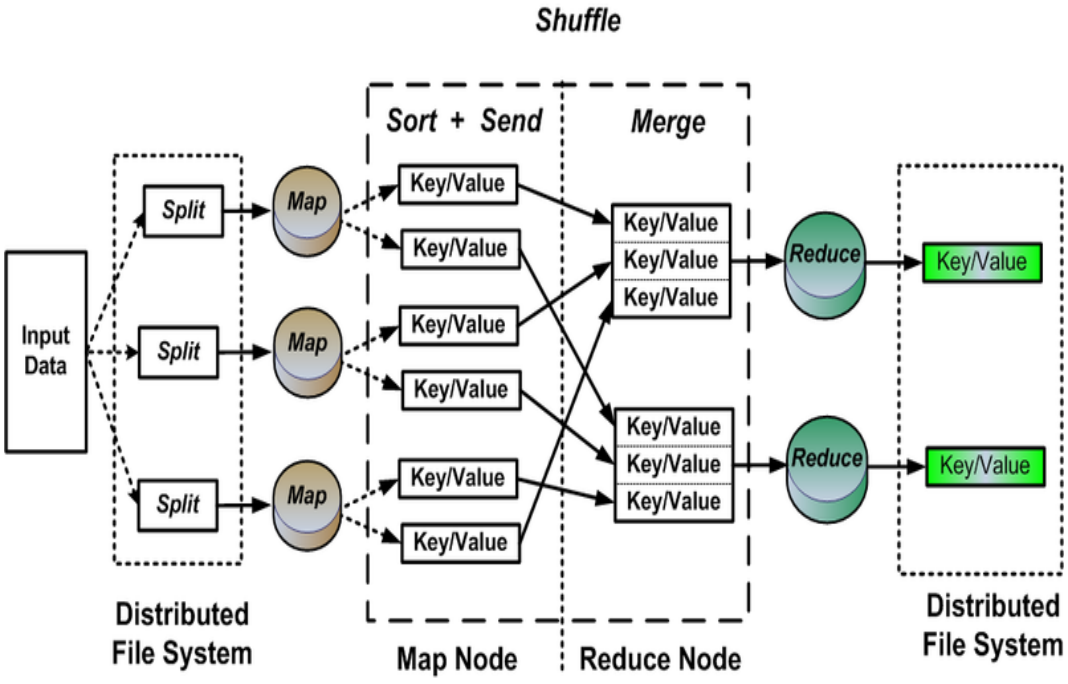
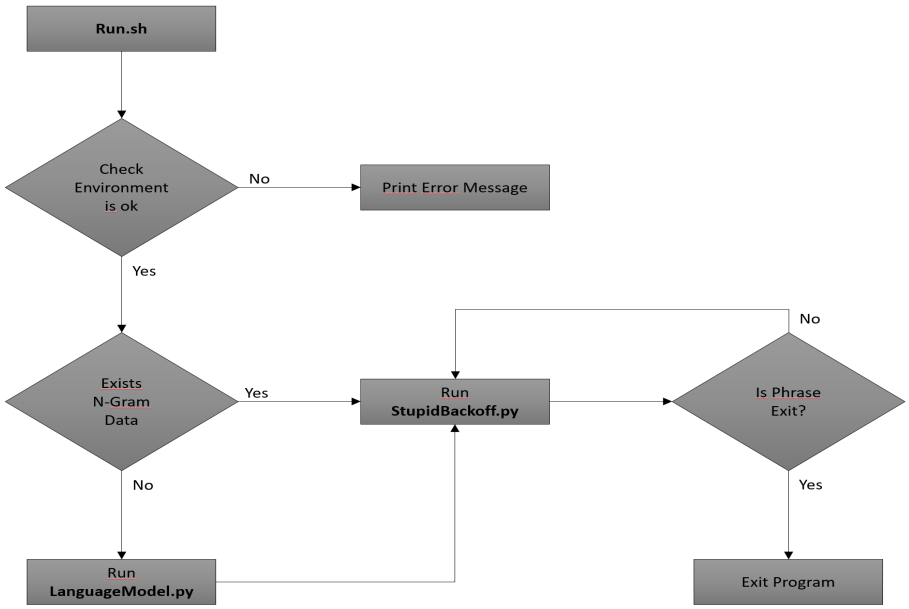


Figure 1: http://gppd-wiki.inf.ufrgs.br/index.php/File:Mr-dataflow-eng.png

7 The algorithm

In the following section we are going to analyze one by one all the classes present in our algorithm.



7.1 CreateCorpus

- **downloadFiles**
  It checks if the files we are going to base our predictions on are already dowloaded or not. If it is not the case, it downloads them in a compressed format from
  https://www.dropbox.com/s/hbmn0rbkujnxqrt/ANC_Corpora.tar.gz?dl=1.

- **extractFiles**
  After the download is completed, it checks if the text in the file is already extracted (from the compressed format) If it is not, it calls the method **extractFunction**.

- **extractFunction**
  It extracts files from the compressed format.

- **creteCorpus**
  It creates a new text file named **RawCorpus.txt**, if it does not exist yet, and it is filled with all the texts extracted in advance.

7.2 CleanCorpus

- **createTokens**
  Considering a line of text, it strips newlines, converts all the characters to lowercase, removes numbers and tokenizes the lines, that is it converts lines into strings vectors. Finally it calls the method **removePunctuation**.

- **removePunctuation**
  It removes punctuation from a token.

- **read_corpora**
  It reads the Raw Corpus file and returns all lines tokenized (by calling **createTokens**) and cleaned.

The constructor of this class, by calling the function **read_corpora** in order to read the file RawCorpus.txt, put all the tokens in the file Tokens.txt.

7.3 Mapper (of MapReduce)

- **tokens**
  It look for all the strings containing only letters through a token and put everything to the lowercase.

- **to_ngrams**
  It returns the memory location of the created N-grams, where the N is specified by *length*.

Mapper take a piece of text and line by line creates the N-grams, with N going from 1 to 5, and counts them (the counting is done already here in the Mapper in order not to overload the Reducer step).

7.4 Reducer (of MapReduce)

First we create a dictionary with a layout in accordance with that produced by the Mapper: for each line contained in the input file (which is the output of the Mapper), the keys are the N-grams and the values are their counting.

7.5 StupidBackoff

- **removePunctuation**
  It deletes everything except of alphabet characters.

- **createTokens**
  Considering a line of text, it strips newlines, converts all the characters to lowercase, removes numbers and tokenizes the lines, that is it converts lines into strings vectors. Finally it calls the method **removePunctuation**.

- **CleanPhrase**
  It cleans a phrase by calling **createTokens**. If the length of the tokenize phrase is greater or equal than 4, it keeps only the last four words.

- **train**
  For each line in the corpus

7.6 Train

It downloads and creates the Corups by **CreateCorpus**, then it cleans the data contained in a given path by **CleanCorpus**, reads by the method **read_corpora** and finally it interacts with the users asking for a sentence as input and then, through **StupidBackoff**, returns the three most likely words that may complete the input sentence.

8 Alternative solutions

The following can replace the MapReduce class. The main difference is that, while MapReduce approach generates N-Grams by splitting the original corpus of the text in many sub-texts, working with one of these chunks at time and aggregating all together just in the end of the procedure, the **NGrams** class make operations starting from the unitary corpus without splitting it.

8.1 NGrams

- **createNgrams**
  It creates a data frame with the columns labeled how we want (in our case we have five: "5Gram", "4Gram", "3Gram", "2Gram", "Candidate"). Then it goes through our collections of vectors TokensData and for each vector it creates all the possible N-grams, configuring them as words vectors. Finally, it counts the produced N-grams and the data frame produced shows all the N-grams with their counting.

- **dataModel**
  It opens the file containing the tokens and iteratively create N-grams calling the function **createNgrams**.

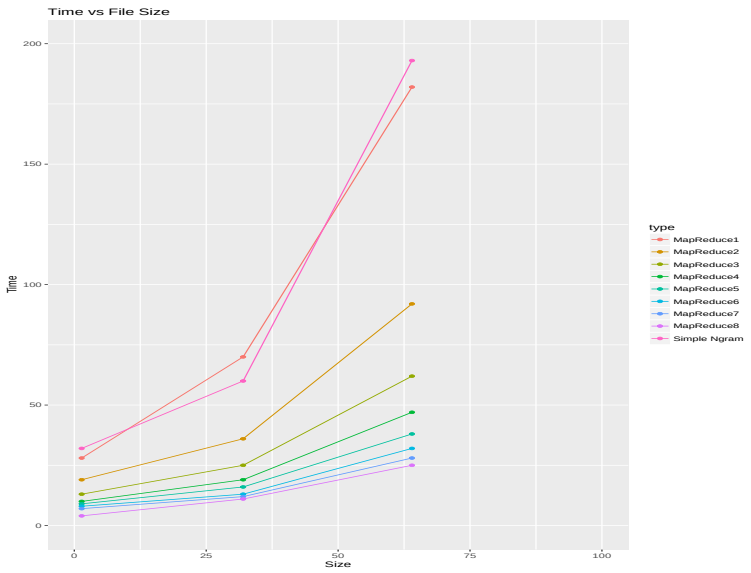
9 Complexity

9.1 Time Complexity

The time complexity of an algorithm quantifies the amount of time taken to run as a function of the length of the string representing the input. It is very complicated to compute a general formula for the time complexity with the big O notation, because we should consider that the input is one or more text files with different numbers of lines which in turn have different number of words, punctuation charachters and so on.
What we can do is to make some simulations with different sizes of input files and different methods (simple N-grams method or MapReduce method with many cores). What we get is reported in the following table:

Simulation	Start	Type	TokenSize	TotalWords	Cores	End	Time
1	06/17/2017 12:35:01	Simple Ngram	**1.4**	15200	1	06/18/2017 13:07:17	**00:32**
10	06/18/2017 12:26:05	Simple Ngram	**32**	486400	1	06/18/2017 13:27:01	**01:00**
19	06/18/2017 16:47:50	Simple Ngram	**64**	977360	1	06/18/2017 20:01:02	**03:13**
9	06/18/2017 11:53:43	MapReduce	**1.4**	15200	1	06/18/2017 12:26:05	**00:28**
18	06/18/2017 15:37:20	MapReduce	**32**	486400	1	06/18/2017 16:47:50	**01:10**
27	06/19/2017 01:27:45	MapReduce	**64**	977360	1	06/19/2017 04:30:15	**03:02**
8	06/18/2017 11:34:43	MapReduce	**1.4**	15200	2	06/18/2017 11:53:43	**00:19**
17	06/18/2017 15:00:50	MapReduce	**32**	486400	2	06/18/2017 15:37:20	**00:36**
26	06/18/2017 23:55:15	MapReduce	**64**	977360	2	06/19/2017 01:27:45	**01:32**
7	06/18/2017 11:21:13	MapReduce	**1.4**	15200	3	06/18/2017 11:34:43	**00:13**
16	06/18/2017 14:35:40	MapReduce	**32**	486400	3	06/18/2017 15:00:50	**00:25**
25	06/18/2017 22:52:45	MapReduce	**64**	977360	3	06/18/2017 23:55:15	**01:02**
6	06/18/2017 11:10:28	MapReduce	**1.4**	15200	4	06/18/2017 11:21:13	**00:10**
15	06/18/2017 14:16:10	MapReduce	**32**	486400	4	06/18/2017 14:35:40	**00:19**
24	06/18/2017 22:05:15	MapReduce	**64**	977360	4	06/18/2017 22:52:45	**00:47**
5	06/18/2017 11:01:22	MapReduce	**1.4**	15200	5	06/18/2017 11:10:28	**00:09**
14	06/18/2017 13:50:04	MapReduce	**32**	486400	5	06/18/2017 14:16:10	**00:16**
23	06/18/2017 21:26:45	MapReduce	**64**	977360	5	06/18/2017 22:05:15	**00:38**
4	06/18/2017 10:53:22	MapReduce	**1.4**	15200	6	06/18/2017 11:01:22	**00:08**
13	06/18/2017 13:48:14	MapReduce	**32**	486400	6	06/18/2017 13:50:04	**00:13**
22	06/18/2017 20:54:15	MapReduce	**64**	977360	6	06/18/2017 21:26:45	**00:32**
3	06/18/2017 10:46:10	MapReduce	**1.4**	15200	7	06/18/2017 10:53:22	**00:07**
12	06/18/2017 13:34:01	MapReduce	**32**	486400	7	06/18/2017 13:48:14	**00:12**
21	06/18/2017 20:26:02	MapReduce	**64**	977360	7	06/18/2017 20:54:15	**00:28**
2	06/18/2017 10:42:02	MapReduce	**1.4**	15200	8	06/18/2017 10:46:10	**00:04**
11	06/18/2017 13:27:01	MapReduce	**32**	486400	8	06/18/2017 13:34:01	**00:11**
20	06/18/2017 20:01:02	MapReduce	**64**	977360	8	06/18/2017 20:26:02	**00:25**

We can plot this results as follows (with the time expressed in **minutes** and the size in **megabytes**):



In the legend, the number after MapReduce corresponds to the number of cores used in that case. We see that the increasing of the time spent according to the size of the files seems not to be linear. This depends on the fact that we are considering also the initialization of the algorithm (downloading the files, extracting them from a compress format, writing a text file) and not only the part where the N-grams are produced.
What we notice for sure, anyway, is that the usage of different cores for the running of the algorithm is quiet as we were expecting: that is, the time needed for an operation with eight cores is $\frac{1}{8}$ with respect to the same operation done with one core.

9.2 Spatial Complexity

First of all we see that the output produced using the Simple N-grams method and the MapReduce method are the same as we expected.

Type	TokenSize	TotalWords	UnigramSize	BiGramSize	TrigramSize	4GramSize	5GramSize	OutputSize
Simple Ngram	**1.4**	15200	17	25	27	28	12	**109**
MapReduce	**1.4**	15200	17	25	27	28	12	**109**
Simple Ngram	**32**	486400	21	27	28	30	18	**124**
MapReduce	**32**	486400	21	27	28	30	18	**124**
Simple Ngram	**64**	977360	29	32	37	38	32	**168**
MapReduce	**64**	977360	29	32	37	38	32	**168**
MapReduce	**164.7**	2538400	197	218	255	298	302	**1270**

Anyway, also here is quiet hard to say something about the trend of the spatial complexity, because it depends on the input files: if the same sequence of words repeats very often, it will occupy the same memory used in the case it appears only once.

10 References

O. Tange (2011): GNU Parallel - The Command-Line Power Tool, ;login: The USENIX Magazine, February 2011:42-47.