Instituto Politécnico de Portalegre
Escola Superior de Tecnologia e Gestão
Departamento de Tecnologias
Licenciatura em Engenharia Informática

Relatório de Projecto

A Natural Language Processing Toolkit
Language modeling for next word prediction

Apresentado por
João Miguel Príncipe Fé

Orientador
Valentim Realinho

Setembro de 2021

Abstract

This work presents a small natural language processing (NLP) toolkit developed in C, with a Python binding interface. The ultimate goal of this work is to develop a writing assistance system for augmentative and alternative communication (AAC), which is used by people that are incapable of using the common forms of communication (speech and writing). From the various types of AAC, this work focuses on pictogram-based AAC, which can be used by people of every age, literacy and cognitive levels, and with motor diseases. A fundamental component of a writing assistance system is the next word (or pictogram) suggestion/prediction. This work focuses on this part. To this end, the state-of-the-art work in language modeling is reviewed and a small toolkit for next word prediction is developed. The carried experiments show the potentiality of the developed toolkit to be integrated in a writing assistance system for AAC, but also in any NLP application where next word prediction is needed.

Keywords: augmentative and alternative communication, writing assistance, language modeling, natural language processing, word prediction

Resumo

Este trabalho apresenta o desenvolvimento de um pequeno *toolkit* de processamento de linguagem natural desenvolvido (NLP) em C, com uma interface em Python. A criação deste *toolkit* tem como objetivo final a implementação de um sistema de assistência de escrita no âmbito da comunicação aumentativa e alternativa (AAC), utilizada por pessoas incapazes de utilizar as formas mais comuns de comunicação – a fala e a escrita. Das várias linguagens de ACC, este trabalho foca-se nas baseadas em pictogramas, que por poderem ser utilizadas por pessoas de várias idades, tipos de literacia e desenvolvimento cognitivo, e problemas motores, são as mais abrangentes. Uma parte fundamental num sistema de assistência de escrita é a sugestão/predição da próxima palavra (ou pictograma, na linguagem em particular). É nesta parte que o trabalho se foca. Nesse sentido, é estudado o estado da arte na área da modelação de linguagem e é desenvolvido um pequeno *toolkit* para predição de palavras. Os testes realizados demonstram a potencialidade do *toolkit* desenvolvido para ser integrado num sistema de assistência de escrita no âmbito da AAC, mas também em qualquer sistema de NLP que necessite de fazer predição de palavras.

Palavras-chave: comunicação alternativa e aumentada, escrita assistida, modelação de linguagem, predição de palavras, processamento de linguagem natural

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Communication is a key element in animals' life. It has greater importance for any social animal, but for humans, it is probably even more important since without it *Homo sapiens* could not be the species that maintains the most complex network of social relationships among all species (which in turn allowed it to evolve in the way it did). For most human beings, communication such as speech or writing is a given, however, some suffer from serious communication difficulties, due to some kind of disease. One should have no doubt about the importance of assisting those persons, not only from an individual point of view but also from a social point of view, considering, e.g., the knowledge that Stephan Hawking gave us all through its augmentative and alternative communication (AAC) device. While in the past it was very difficult to assist these people, today, the ubiquity of computers in our daily lives and the current-century progress in areas such as machine learning and natural language processing (NLP) (with the former influencing the latter), have opened new doors to the development of superior AAC systems. This is the ultimate goal of this work.

There are different kinds of AAC, but two main branches can be identified: gestural and graphic. The gestural requires motor skills, being as such not an option for people with motor diseases. Within the graphic type are the systems based on images, pictograms, words, and letters. Considering factors such as young age and illiteracy, the AAC systems based on pictograms can be considered as the most encompassing type of system. Although broader, this system can impose serious challenges to fluent conversations since the set of choosable pictograms is very wide and the users need to select the right ones manually, sometimes through non-friendly input interfaces (which becomes particularly difficult for users with strong motor diseases). Considering this problem, it would be of greater importance to have an intelligent pictogram predictor system that would suggest pictograms given the current sequence of pictograms that the user already chose during the current conversation.

Considering the availability of resources in the area of NLP, this work has focused on the task of next *word* prediction (NWP) instead of *pictogram*. However, it is expected that

much of the work done here can be translated to the specific task of pictogram prediction. Predicting the next word given some context sentence is tightly related to language modeling, a sub-field of NLP with applications in several areas, such as machine translation, speech recognition, handwriting recognition, and others. A statistical language model is a queryable data structure from which a probability can be assigned to any given sentence. A sentence can be seen as a particular sample of a language; thus, if some sentence is commonly used within a language, a good language model for that language should assign it a relatively high probability. Considering language models, the task of NWP given some sentence context $\boldsymbol{w} = w_1, \ldots, w_L$ and some vocabulary $\mathcal{V}$, can be seen as the task of finding the word $w_{L+1} \in \mathcal{V}$ that maximizes the probability $P(\boldsymbol{w}, w_{L+1})$. Independently of targeted language (Portuguese, English, sign language, pictogram-based, etc.), there are different types of language models. The most widely studied over the literature are the n-gram models, which estimate the probability of a given sentence by the chain product of the conditioned probabilities but conditioning the probabilities only on the last $n-1$ words, i.e. $P(\boldsymbol{w}) = \prod_{i=1}^{L} P(w_i|w_1, \ldots, w_{i-1}) \approx \prod_{i=1}^{L} P(w_i|w_{i-n+1}, \ldots, w_{i-1})$. In the last decade, neural network based language models gain a lot of attention, with the achievement of performances superior to n-gram models in several applications. The simplest types of these models have a fixed number of input units capable of processing only the last n context words, however, the state of the art models are recursive neural networks that receive one word at a time but are able to capture long-range relations within a sentence due to their memory units.

This work had the goal of applying both types of language models to the application of NWP, however, the n-grams models were sufficient to spent the summer in the computer. The general concept of language modeling is presented in Chapter 2, with especial focus on n-gram models on Section 2.1. Within this section two key subjects of n-grams models, smoothing techniques and indexing data structures, are surveyed in 2.1.1 and 2.1.2, respectively. Considering the particular application of NWP, some possible improvements to n-gram models are given in 2.1.3. In Chapter 3 a detailed description of the developed work is presented, with Section 3.1 giving an overview of the work, Section 3.2 introducing a widely use language model format file, Section 3.3 describing the applied preprocessing of used text corpus, Section 3.4 describing and analyzing the implemented indexing data structure, and Section 3.5 showing a sample usage of the developed toolkit. A set of experiments is presented in Chapter 4, validating the hypothesis that n-gram language models can be used for NWP. Finally, Chapter 5 concludes the work, with a critical view and outline of ideas for future work.

Chapter 2

Language Modeling

Language modeling is the NLP sub-field which aims to develop good language models, from which a useful probabilities can be assigned to any given sentence. A good language model should assign a greater probability to sentence $\boldsymbol{w}_A$ than to sentence $\boldsymbol{w}_B$ if $\boldsymbol{w}_A$ is more commonly used within the considered language than $\boldsymbol{w}_B$. The way a language model estimates these probabilities depends on the considered type of language model. The following section will describe how the probabilities can be estimated for n-grams, but before getting into the details of n-gram language models, it is important to understand why consider language models in first place since the goal is to predict the next word given some context. Firstly, the task of NWP is, by the nature of the natural language, subject to stochasticity; not even the humans (which are the experts in natural language) can predict every word someone is going to say. As such, NWP should be approached in a probabilistic way (as language modeling is). Secondly, if one develop some system that can make the NWP $\max_w P(\boldsymbol{w}, w)$ for every possible context sentence $\boldsymbol{w}$, it is very likely that that same system can also estimate $P(\boldsymbol{w}, w)$ for any w.[1] But estimating probabilities for every $\boldsymbol{w}, w$ is exactly what language models do, whereby it can be seen how language modeling and NWP are interchangeable concepts. However, when searching the literature for NWP few work is found (and what appear refer to language modeling) in comparison with what exist around language modeling. For this reason, the research in this work focuses on language modeling as a mean to achieve NWP. This is also how keyboard decoders that use NWP are lined up [9][18].

[1] If it does not, how does it know the $\max_w P(\boldsymbol{w}, w)$?

2.1 n-gram language models

Given some sentence $\boldsymbol{w} = w_1, \ldots, w_L = w_1^L$, by the chain rule of probability, its probability $P(\boldsymbol{w})$ can be calculate as

$$\begin{aligned} P(\boldsymbol{w}) &= P(w_1^{L-1}) \times P(w_L \mid w_1^{L-1}) \\ &= P(w_1) \times P(w_2 \mid w_1) \times \cdots \times P(w_L \mid w_1^{L-1}) \\ &= \prod_{i=1}^{L} P(w_i \mid w_1^{i-1}). \end{aligned}$$

However, this calculation is not feasible in practice since the complexity in regards of space to store the conditioned probabilities would be $O(|\mathcal{V}|^L)$. To solve this problem, n-gram models make an $(n-1)^{\text{th}}$ order Markov assumption by assuming that $P(w_i \mid w_1^{i-1}) \approx P(w_i \mid w_{i-n+1}^{i-1})$, yielding to simpler estimation of $P(\boldsymbol{w})$ defined as

$$P(\boldsymbol{w}) \approx \prod_{i=1}^{L} P(w_i \mid w_{i-n+1}^{i-1}),$$

that reduces the complexity to $O(|\mathcal{V}|^n)$. Considering that in any natural language the vocabulary $\mathcal{V}$ contains typically hundreds of thousands to millions of words, this makes a huge difference in absolute values.

The natural way to estimate these conditioned probabilities is by counting the occurrences of the n-grams on the training corpus:

$$P(w_i \mid w_{i-n+1}^{i-1}) = \frac{c(w_{i-n+1}^{i})}{c(w_{i-n+1}^{i-1})}, \tag{2.1}$$

where $c(\boldsymbol{w})$ gives the number of occurrences of $\boldsymbol{w}$ on the training corpus. In this way, the resulting probabilities estimations are the maximum likelihood (ML) estimations on the training data, and this type of estimation will be referred as P_{ML} in this work. While being a simple way of estimating, the resulting model might suffer from two problems: high bias (underfitting) and high variance (overfitting). High bias might occur when n is too small, being the model unable of understanding any medium-long range dependency within a sentence, whereas a high variance occur when n is large and the estimations are made from a not so big corpus.[2] In fact, both problems can occur simultaneously since (1) natural language is full of long-range dependencies that cannot be captured using a particular n, while (2) the training corpus may not be sufficiently large to contain all language phenomena well distributed for the same n, resulting in imprecise probabilities

[2]The number of probability estimations increase exponentially with n, but a corpus is finite. Thus, when n increases, the number of samples available in the corpus for each probability estimation decreases exponentially. If a probabilities are estimated from very few samples, it is unreliable that these probabilities will be suitable for any other corpus.

estimations [4].

While there is probably no way of avoiding high bias when using a small n, there are techniques to minimize the data sparsity problem that occurs when is n large. The most considered technique in the literature is the *smoothing* technique, which tries to make the estimated probabilities more uniform, decreasing the high probabilities and increasing the low. There are many types of smoothing techniques, each one with its own motivation. Chen and Goodman present some of these techniques in an extensive empirical study [3]. In the following subsections, several smoothing techniques are presented, from the simplest to the state-of-the-art smoothing technique, the modified Kneser-Key.

2.1.1 Smoothing techniques

Additive smoothing

The simplest type of smoothing is the additive smoothing [16, 17, 13, 12]. This adds a constant value α to each count, where typically $0 < \alpha < 1$:

$$P_{\text{add}}(w_i|w_{i-n+1}^{i-1}) = \frac{\alpha + c(w_{i-n+1}^{i})}{\alpha|\mathcal{V}| + c(w_{i-n+1}^{i-1})}.$$

In this way there is no zero probabilities as might happen with ML estimations and the probability distribution becomes more smoother than ML.[3] To understand more concretely how probabilities are modified with this smoothing, it can be stated that:

$$\begin{cases} P_{\text{add}}(\boldsymbol{w}) < P_{\text{ML}}(\boldsymbol{w}) & \text{if } P_{\text{ML}}(\boldsymbol{w}) > \frac{1}{|\mathcal{V}|} \\ P_{\text{add}}(\boldsymbol{w}) > P_{\text{ML}}(\boldsymbol{w}) & \text{if } P_{\text{ML}}(\boldsymbol{w}) < \frac{1}{|\mathcal{V}|} \\ P_{\text{add}}(\boldsymbol{w}) = P_{\text{ML}}(\boldsymbol{w}) & \text{if } P_{\text{ML}}(\boldsymbol{w}) = \frac{1}{|\mathcal{V}|} \end{cases}$$

Good-Turing

Imagine someone is fishing and caught 10 carp, 3 perch, 2 whitefish, 1 trout, 1 salmon, 1 eel (totaling 18 fish).[4] How likely is that the next fish to be caught is of trout species? One might say 1/18. How likely is that next species is new (e.g., catfish or bass)? One might consider the number of species that have only appeared once (which is 3) and say 3/18. But assuming this, the probability that next species is trout needs to be less than 1/18 (otherwise, the sum of all probabilities would be $1 + 3/18$). This example shows the problem of ML estimations: the unseen events are not considered, with all probability mass being divided by the seen events. The example can be easily translated to the n-

[3] Zero probabilities are probably not a problem in next word prediction, but in other applications they are. A speech recognition system, for example, might never translate some speech to some text if that text never appeared on the training corpus even if there no noise in the speech.

[4] The slides from where this scenario was taken say its authorship is from Josh Goodman.

gram language models, where the fish fished are the n-grams seen in the training corpus and the unseen fish are the n-grams unseen in the training corpus. J. Good, derives (through non-simple mathematics) a smoothed estimator to answer the questions above, whose simple form is defined as

$$P_{\text{GD}}(w_{i-n+1}^{i}) = \begin{cases} \dfrac{N_1}{|C_n|} & \text{if } c(w_{i-n+1}^{i}) = 0 \\ \dfrac{r+1}{|C_n|}\dfrac{N_{r+1}}{N_r} & \text{otherwise} \end{cases}, \tag{2.2}$$

where $|C_n|$ is the total number of n-grams in the corpus, and N_r is the number of *distinct* n-grams that appeared exactly r times on the training corpus [11].

Eq. 2.2 was derived theoretically but is actually an approximation and J. Good advocates that for better estimations N_r values should be themselves smoothed before by other smoothing technique. Nonetheless the conditions under which these smoothing is applicable are not restrictive and the larger N_1 the more applicable are the results [11]. This smoothing technique has influenced several other techniques.

Interpolation vs. Backoff

Two different ways of estimating the probability of an n-gram were given above, but both ignore (because they were not developed specifically for language models) the fact that an n-gram frequency might has an important correlation with its $(n-1)$-gram context frequency. Thus, if the estimated n-gram frequency is unreliable (i.e., there are few counts of the n-gram on the training corpus) it might be useful to improve it with the frequency values of the $(n-1)$-gram, and even of $(n-2)$-gram, $(n-3)$-gram, ..., and 1-gram.

There two main approaches for combining lower-order n-gram frequencies. One is *linear interpolation*, which can be described as:

$$P_{\text{inter}}(w_i|w_{i-n+1}^{i-1}) = \rho(w_i|w_{i-n+1}^{i-1}) + \lambda P_{\text{inter}}(w_i|w_{i-n+2}^{i-1}), \tag{2.3}$$

where λ is some weight that gives importance to the lower-order probability estimations; and $\rho(w_i|w_{i-n+1}^{i-1})$ is some probability estimation for $w_i|w_{i-n+1}^{i-1}$, e.g., $P_{\text{ML}}(w_i|w_{i-n+1}^{i-1})$. To end the recursion, it can be defined that $P_{\text{inter}}(w_i|\varnothing) = \rho(w_i|\varnothing) = \rho(w_i)$. The parameter λ is typically defined so that the probabilities sum to 1. It can nonetheless be optimized considering some metric. It is not mandatory to have a single λ for all contexts w_{i-n+2}^{i-1}, and in fact, that yields to poor performance. On the contrary, a different λ for each existing w_{i-n+2}^{i-1} does not worth the effort (it would take time to be estimated, take extra space within the model data structure, and the final consequence could just be a training data overfitting). Bahl, Jelinek, and Mercer suggest estimating one λ for each set of n-grams that occur exactly r times on the training corpus [1].

The other approach is *backoff*, which instead of always considering lower-order n-gram frequencies, just backoffs to lower-order when there is no occurrence of the n-gram:

$$P_{\text{backoff}}(w_i|w_{i-n+1}^{i-1}) = \begin{cases} \rho(w_i|w_{i-n+1}^{i-1}) & \text{if } c(w_{i-n+1}^{i}) > 0 \\ \lambda P_{\text{backoff}}(w_i|w_{i-n+2}^{i-1}) & \text{if } c(w_{i-n+1}^{i}) = 0 \end{cases} \tag{2.4}$$

Absolute discounting

The absolute discounting was originally presented as an interpolation smoothing technique that subtracts a fixed discount $0 < D < 1$ to the number of occurrences of an n-gram [19], so that in Eq. 2.3

$$\rho(w_{i-n+1}^{i}) = \frac{\max\{c(w_{i-n+1}^{i}) - D, 0\}}{c(w_{i-n+1}^{i-1})}$$

and to ensure that the distribution sum to 1

$$\lambda = \frac{D}{c(w_{i-n+1}^{i-1})}|w_i : c(w_{i-n+1}^{i}) > 0|,$$

being its complete definition:

$$\begin{aligned} P_{\text{abs}}(w_{i-n+1}^{i}) = & \frac{\max\{c(w_{i-n+1}^{i}) - D, 0\}}{c(w_{i-n+1}^{i-1})} + \\ & \frac{D}{c(w_{i-n+1}^{i-1})}|\{w_i : c(w_{i-n+1}^{i}) > 0\}|P_{\text{abs}}(w_{i-n+1}^{i-1}). \end{aligned}$$

Ney and Essen [20], arrived lately to a estimation of

$$D = \frac{n_1}{n_1 + 2n_2}.$$

The concept of absolute discounting was integrated in several other smoothing developed later, mainly due to its simplicity. But a Church and Gale had already shown that Good-Turing discounts become fixed when $r \geq 3$, which brings even more value to absolute discounting.

Modified Kneser-Ney

One common problem of existing backoff methods, identified by Kneser and Ney, was that the backing off to lower-order probability estimation could lead to a higher probability estimation of the full n-gram than it should [14]. To understand why this is the case, consider the word *dollars* which is very frequent in the Wall-Street-Journal corpus but occurs almost exclusively after numbers.[5] It is very unlikely that the word *dollars*

[5]Example from [14].

will occur after a word that has not been observed, say *José*. However, the estimated probability $\rho(dollars|José)$ will be relatively high because the model backoffs to *dollars* and $\rho(dollars)$ itself is relatively high.

To solve this problem, Kneser and Ney [14] proposed a new absolute discounting based estimator, defined as

$$P_{\text{KN}}(w_i|w_{i-n+1}^{i-1}) = \begin{cases} \dfrac{\max\{c(w_{i-n+1}^{i}) - D, 0\}}{c(w_{i-n+1}^{i-1})} & \text{if } c(w_{i-n+1}^{i}) > 0 \\ \lambda \dfrac{N_{1+}(\bullet, w_{i-n+2}^{i})}{N_{1+}(\bullet, w_{i-n+2}^{i-1}, \bullet)} & \text{if } c(w_{i-n+1}^{i}) = 0 \end{cases},$$

where $\bullet$ is a placeholder for *any-word* and $N_{1+}(\bullet \boldsymbol{w})$ gives the number of distinct words that precede $\boldsymbol{w}$, so that

$$N_{1+}(\bullet, w_{i-n+2}^{i}) = |\{w_{i-n+2} : c(w_{i-n+1}^{i}) > 0\}| \quad \text{and}$$
$$N_{1+}(\bullet, w_{i-n+2}^{i-1}, \bullet) = |\{(w_{i-n+2}, w_i) : c(w_{i-n+1}^{i}) > 0\}| = \sum_{w_i} N_{1+}(\bullet, w_{i-n+2}^{i}).$$

Later, Chen and Goodman made some modifications to this method [3]. Firstly, they used a linear interpolation version of the Kneser and Ney, i.e.,

$$P_{\text{mKN}}(w_i|w_{i-n+1}^{i-1}) = \frac{c(w_{i-n+1}^{i}) - D(c(w_{w-n+1}^{i}))}{c(w_{i-n+1}^{i-1})} + \lambda P_{\text{mKN}}(w_i|w_{i-n+2}^{i-1}). \tag{2.5}$$

This version is fully recursive as opposite to the original version, which was designed for bigrams (even though a recursive version of it would be straightforward). As referred at the final paragraph of 2.1.1, Good-Turing discounts become fixed when $r \geq 3$, but when $r \leq 3$ they tend to differ considerably. In line with this, another difference is the use of a discount $D(c)$ that depends on the number occurrences of w_{i-n+1}^{i} in the training corpus:

$$D(c) = \begin{cases} 0 & \text{if } c = 0 \\ D_1 & \text{if } c = 1 \\ D_2 & \text{if } c = 2 \\ D_{3+} & \text{if } c \geq 3 \end{cases}.$$

Lastly, the λ coefficient ensures that the probabilities sum to 1, while giving more or less importance to the lower-order distribution if more or less distinct words procede the lower-order n-gram, respectively, in the training corpus (in similar thought to Kneser and Ney):

$$\lambda = \frac{D_1 N_1(w_{i-n+1}^{i-1}, \bullet) + D_2 N_2(w_{i-n+1}^{i-1}, \bullet) + D_{3+1} N_{3+}(w_{i-n+1}^{i-1}, \bullet)}{c(w_{i-n+1}^{i-1})}.$$

The estimated optimal values for D_1, D_2, and D_{3+} are

$$D_1 = 1 - 2Y\frac{n_2}{n_1}$$
$$D_2 = 2 - 3Y\frac{n_3}{n_2}$$
$$D_{3+} = 3 - 4Y\frac{n_4}{n_3},$$

where $Y = \frac{n_1}{n_1+2n_2}$ [3].

While somewhat complex and challenging to build language model using it, this became the most widely used smoothing technique in language models.

2.1.2 Indexing and querying data structures

In the previous section it was discussed how to achieve the most accurate and trustable probability estimations. However, that by itself is not enough if there is no efficient data structure capable of mapping a large set of (possibly large-order) n-grams to their respective conditioned probabilities, so that a powerful language model can exist. Not only do the conditioned probabilities of seen n-grams need to be available, but also the backoff coefficients that allow estimating probabilities for unseen n-grams. This backoff coefficient refers to the λ parameter in the equations of the previous section.[6] While some estimators use a constant λ, the state-of-the-art modified Kneser-Ney uses a (possibly) different λ for each n-gram. Note that, differently from the probability values, the backoff values do not need to be saved for the higher-order n-grams, such that for a language model of order N, only the l-grams need to have an associated λ, where $l \in \{1, \ldots, N-1\}$. Together, probability and backoff values are also referred as *satellite values*; and developing a data structure capable of handling the satellite values of every n-gram in a training corpus is not a memory friendly challenge, especially when N is large. To give a sense of the data scale, the training corpus used in this work, the Corpus de Extractos de Textos Electrónicos MCT/Público[7] (CETEMP), (which can be considered small when compared to many available English corpus) contains approximately 1, 15, and 63 millions of distinct 1-grams, 2-grams, and 3-grams, respectively, and the number keeps increasing exponentially with n (until the limited size of the corpus allows it). In this section, two data structures (and several of their variants) capable of index millions of n-grams together with their satellite values will be explored.

The two main data structures used to map n-grams to their satellite values are *hash tables* and *tries*. Hash tables are used when query (i.e., obtain the probability of a given

[6]The term *backoff coefficient* might feel strange when talking about interpolation methods, but note that interpolation methods backoff implicitly to lower-order frequencies for unseen n-grams. (For unseen n-grams, Eq. 2.3 and Eq. 2.4 yield the same result.)

[7]https://www.linguateca.pt/CETEMPublico/

n-gram) speed is more important than memory consumption. In this scheme, there is one (giant) hash table for each order, totaling N hash tables. For each order l, on each entry of the l^{th} hash table are saved the l-gram key, probability and backoff coefficient; and for the order N, on each entry of the N-grams hash table only the key and probability value are saved (since, as already referred, the backoff values do not need to be saved for N-grams). Considering that normal training corpus tend to have millions of distinct n-grams, it is common to use 64-bit hash keys to avoid hash collisions[8]. A popular hash function for the job is MurmurHash[9], which, while computationally simple, yields considerably well-distributed hashes. To resolve bucket collisions in hash tables (which should not be confused with hash collisions) it is necessary to have more buckets than entries, so that whenever the bucket corresponding to some hash is already occupied, an empty one can be found easily (through some probing technique, such as linear probing). The ration of buckets to entries is controlled by a space multiplier $m > 1$. Hash tables allow for constant time (in relation to the number of entries) when querying for the probability of a given n-gram, however this type of model is not the most memory efficient since it is necessary to keep 64-bit keys and compression techniques can not be applied because of the random nature of the data structure.

A trie [8], or prefix tree, is a tree data structure used to index compounded keys in situations where keys tend to share some of their sub-parts. This fits well for indexing n-grams, considering that each n-gram is formed by other lower-order n-grams, which are typically shared between multiple n-grams. To understand how tries structure the information consider, as an example, a vocabulary $\mathcal{V} = \{\texttt{A}, \texttt{B}, \texttt{C}\}$ and the corpus `ABBCACABAB`, from which several n-grams can be counted, and see the trie representation in Figure 2.1. This example trie models up to 3-order n-grams, but each of its nodes contains only one vocabulary symbol as key. To find some n-gram, say the 3-gram `ABB`, the trie is depth-first searched by `A`, then `B`, and then by `B`, reaching the level 3 of the trie, where it can see that the trigram `ABB` count is 1. While this example considers only n-gram counts, the nodes can obviously keep any necessary satellite values, such as conditioned probabilities and backoff coefficients. Even though the figure shows that each node have a set of pointers to its children, in practice this is unfeasible (and not necessary). Instead, several approaches have been developed to implement these tries more efficiently, as it will be described next. (The toolkits referred next also have especial hash table implementations, but the focus will be on tries since they are more memory efficient, and thus more suitable for mobile devices).

The SRILM toolkit [24] implements the trie data structure in two distinct ways. The default one uses one hash table per node from where the 64-bit children pointers can

[8]Considering a completely uniform-random hash function and using 64-bit keys, a collision is only expected after hashing a total of $\sqrt{2^{64}} = 2^{32}$ distinct n-grams. Search for *birthday problem* for extra details.

[9]2http://sites.google.com/site/murmurhash/

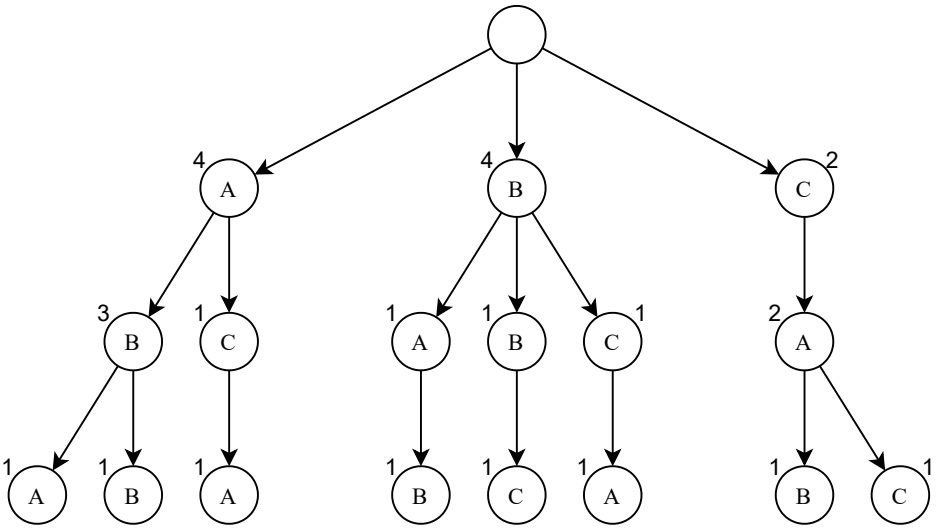

Figure 2.1: Trie for an example corpus `ABBCACABAB`. The number beside each node represents its counts in the corpus. For example, the unigram `A` appears 4 times and the bigram `AB` 3 times.

be obtained. The compact version uses sorted arrays instead of hash tables within each node, requiring binary search to find a child but saving some memory since hash keys are not stored nor the extra buckets necessary for hash tables. Even though, it still keeps the 64-bit pointers to the children, just like shown in Figure 2.1, being thus an inefficient representation. Both implementations use a pure C struct to represent each node, with keys and satellite values being saved in normal data types, such as `int` and `float`, forbidding any kind of compression, quantization[10], or bit packing. In the end, the resulting models are simple and fast (special the one that uses hash tables), but require too much memory.

The IRSTLM project [7] was developed with more focus on memory usage and it overcomes SRILM in that regard. In their trie implementation, each node saves only one pointer to an array wherein all its children are (in opposition to SRILM, where each node had one pointer for each child). During the build process these arrays are allocated (and reallocated if necessary) individually, but after every n-gram in the corpus is read, the arrays belonging to the same order are collapsed in a giant array, being the final trie represented by N sorted arrays. The pointer previously mentioned, is actually the start index of the children in the next-order array. Table 2.1.2 show how the N arrays would be populated for the example of Figure 2.1. The arrays are sorted by complete key, which is clear for the first-order array; the second-order array starts with `B` because its complete key is `ABB`, and is followed by `C` which has the key `AC`, and so on. The `index` field of element `A` in the first-order array (with value 0) indicates where its children nodes start in the second-order array. To know the index where its children end, one can see

[10]Quantization is the process of mapping values from a set of some size to values from a smaller set, reducing their memory size but loosing precision.

the children of the adjacent node, `B`, start: since `B`'s children start at index 2, the `A`'s children end at index1. To know where the last elements' children end (e.g., the end index of the `C`'s children), it is necessary to add some a dummy element at the end of each array (except the last order array, since its elements do not have children). These dummy elements have no key nor satellite values, just the index value, which for the first-order array is 6 because `C`'s children start at index 5 but `C` has only one child node (differently, `CA` has two children nodes starting at index 4, which requires the dummy element index in the second array to be $4 + 2 = 6$). To improve the memory efficiency of the solution, IRSTLM additionally offers a compressed version, where 8-bit quantization is applied to the probability and backoff values. To reduce the query and loading times, a hash table based caching mechanism and binary file with memory mapping are used, respectively. The empirical results shown memory savings over SRILM of 64% and 76% when not using and using quantization, respectively, on SITACAD corpus [7].

1	key	A	B	C	X				
	count	4	4	2	X				
	index	0	2	5	6				
2	key	B	C	A	B	C	A	X	
	count	3	1	1	1	1	2	X	
	index	0	1	1	2	3	4	6	
3	key	A	B	A	B	C	A	B	C
	count	1	1	1	1	1	1	1	1

Table 2.1: Implementation of trie from Figure 2.1 using sorted arrays. Entries marked with X are the *dummy* entries that are put at the end of the arrays so that the entries at their left can obtain the children end index.

In 2011, BerkeleyLM [21], using a considerably different trie structure, improved both query time and memory consumption over IRSTLM trie. BerkeleyLM also uses N sorted arrays, however, these are not sorted by key, but by the last part of the key (i.e., by the symbol/word/gram each entry holds). This means that the entries with the same word are all placed consecutively in the array. Taking advantage of this, the words are not explicitly saved in the entries, but rather in an auxiliary array, where there is an entry for each word indicating the range of the main array that corresponds to that word. Also, words are bit packed and satellite values are rank encoded. For CETEMPúblico, for example, having words bit packed would mean that each word could be uniquely represented using just $\lceil \log(1 \times 10^6) \rceil = 20$ bits.[11] Having satellite values rank encoded means to define a set of distinct probabilities and set of distinct backoff values that were calculated from the corpus and sort these sets by descending order of frequency, so that the codes of the most common satellite values are the lower ones, which can then be exploited by the variable-length encoding compression technique that is applied in the compressed version

[11] $\log x$ should be interpreted in this work as binary logarithm of x unless stated otherwise.

of BerkeleyLM. Variable-length encoding the complete arrays would prevent using binary search on them, as such the compression is done in blocks of 128 bytes. While saving memory, it becomes slower than the normal version due to uncompression overhead. The experiments on Web1T corpus [2], shown a memory reduction over the not quantized version of IRSTLM of 82% and 74% when using the compressed and normal version, respectively [7].

In the same year, the KenLM toolkit was published [10], overcoming every existing toolkit, including BerkeleyLM. The trie structure is the same as in IRSTLM (based on sorted arrays, where each node points to the children array range), but here the words and indexes are bit packed, which considerably reduces the memory size. The greater novelty was that, by knowing the range of the word identifiers, they used interpolation search to search the arrays instead of binary search, reducing the expected search time on array A to $O(\log\log|A|)$.[12] In the experiments they did not compare their trie against the non-compressed version of BerkeleyLM, saying that the source code was not released, but the comparison between compressed versions of both toolkits shown that KenLM memory usage was only slightly small, taking about 92% of the BerkeleyLM trie memory. While it have not bring great memory improvements, the structure simplicity inherited from IRSTLM, the use of interpolation search, and the developed *stateful queries* mechanism[13], have resulted in query time speeds at least two times faster than any other toolkit. This toolkit has become the most used n-gram language modeling toolkit in the past decade, being integrated into the Moses, cdec, and Joshua, and other translation systems.

Much more recently, the toolkit developed by Pibiri and Venturini had beat considerably the KenLM in regards of memory usage, through the implementation of Elias-Fano encoded tries [6, 5, 22]. However, the Pibiri and Venturini work was not considered in the present work. Instead, the trie implemented in this work is based on KenLM due to its simplicity and (even so) efficiency. The developed trie and the overall developed toolkit are detailed described in Chapter 3.

2.1.3 Improving language models

The choice of a good smoothing technique and indexing data structure is crucial to have a capable language model. However, depending on the context application and training corpus, the preprocessing of the training might be such or more important than the choice of smoothing technique or indexing data structure. There are several preprocessing techniques that can be applied to the corpus [15], but considering the target application

[12] Interpolation search is similar to what humans do when searching for a word in a dictionary. If they want to find the meaning of word *ambition*, they will open the dictionary in the first pages (because they know that A is the first letter in the alphabet), not in the middle as binary search would do.

[13] The stateful queries (also presented in BerkeleyLM as *scrolling queries*) are mainly useful in translation systems, where several n-grams with similar contexts need to be evaluated. To avoid repeated trie traversals, the model keeps track of the last query context.

of NWP and the use of journal corpus, the most relevant are removal of punctuation, capitalization (i.e., put all letters in lower case or all in upper case), and special characters (this is indeed a problem of the used corpus, where there are many citations wrapped by the symbols « and »). Considering the same corpus but the application of next pictogram prediction, stemming or lemmatization could also be very important techniques. Both stemming and lemmatization convert the words in their basic form (e.g., *studying* is replaced with *study*), which helps to reduce the vocabulary size without necessarily loose the sentence meaning.

Finally, there is a technique that is essential to every tree health: pruning. In real world, if trees are not pruned, their branches break. In computer world, if trees are not pruned, the memory breaks. In language models, pruning is a very important technique considering the fact that many n-grams only appear once. n-grams with few counts not only occupy considerable space, but their probability estimations are also very unreliable, as already discussed. Particularly, in the application of NWP, there is much less value in the n-grams with few counts than in a normal language model that has to be capable of assigning a good probability to any given sentence (including the ones made of uncommon n-grams). Considering this, a fundamental step in the development of successful language model for the task of NWP, is the application o pruning. It is not easy to say in advance exactly what n-grams should be pruned: if the 3-grams that appear only once, all n-grams that appear only once or twice, etc. The right choice is found empirically, considering the trade-off between memory usage and accuracy.

Chapter 3

Implementation

It should be disclosed that it was not part of the initial plans to spend several weeks developing any type of indexing data structure. However, known NLP libraries, such NLTK[1] or SpaCy[2], do not offer any out-of-the-box NWP module, not even language modeling. Most of these libraries offer several useful features, such as text preprocessing functionalities, named entity recognition, part-of-speech tagging, and others, but all these are just utilities to facilitate the development of specific and concrete intelligent systems. Considering out-of-the-box solutions, the closest to NWP, or language modeling to be accurate, is probably the OpenAI GPT-2[3] language model provided by Transformers[4], but using this to achieve NWP would be overkilling and unfeasible in practice. At the same time, the NWP capabilities of some known virtual keyboards are closed-source. Lastly, the toolkits mentioned in the previous chapter, while open-source, can not be used for NWP since they implement reverse tries (see 3.4.2). Because of these reasons (and also the advantages of having your own solution), it was decided to develop the small toolkit for NWP presented here. In the next section, an overview of all the developed work is presented, and in the sections that follow it, concrete descriptions of the different parts are presented.

3.1 Overview

The majority of the project consists of a C library that implements a trie data structure based on the trie implementations of IRSTLM/KenLM. This library is prepared to build/index language model tries from ARPA files (more on this later). Once created, the trie can be queried to assign probabilities to sentences, or to give NWP given some context sentence. The trie itself can be saved to disk in binary format as is in memory,

[1] https://www.nltk.org/
[2] https://spacy.io/
[3] https://github.com/openai/gpt-2
[4] https://github.com/huggingface/transformers

which allows it to be later quickly loaded into memory. Because the indexing procedure takes time, an extra executable that creates tries from ARPA files is also offered. This can be used as a terminal program by users (it offers `--help` documentation) or be instantiated by other programs, which can then kill it whenever and if desired. (This happens in the Python library when users abort the build procedure. When it happens, only the indexing process is terminated, while Python keeps running normally).

The remaining of the project consists of a Python *binding* to the C library and a SpaCy *component* for NWP. The role of the Python binding is to make the developed C library usable in pure Python. It is available for installation on PyPI, but only with wheels for Linux platforms.[5] The SpaCy component depends on the Python binding and is available in Github.[6] SpaCy makes use of the pipeline design pattern to process the text. A pipeline contains a sorted list of components that process the text sequentially, where each component adds attributes to the `Doc` object[7] that can be used by the following components. Thus, the goal of the component is to make the NWP feature available as an add-on for any system developed on SpaCy.

3.2 ARPA format

The ARPA file format is designed to save n-gram backoff language models in pure text. While there is no official specification, the format is so simple and suitable to many smoothing techniques that every toolkit understands it. The header section starts with the keyword `\data\` on the first line and the following lines indicate the number of distinct n-grams (one line per order). The remaining content of the file is the body section, which is divided into N subsections (one for each order), where the probability and backoff values for each n-gram are listed.

The ARPA format is useful because it represents language models in a toolkit-independent way, but it can not be used directly as in-memory indexing data structures since it takes too much space (because it is pure text) and has no structure that allows for efficient queries. The process of generating the ARPA files is commonly referred to as *estimation*. The estimation procedure, where a huge raw corpus is processed, is itself time demanding and several toolkits have developed their own estimation procedure (e.g., KenLM published its estimation procedure in 2013 [10], and Pibiri and Venturini in 2019 [23]). However, estimation should not be confused with indexing, since the latter is the procedure that generates an in-memory data structure that can efficiently be queried by other systems. Figure 3.1 shows the language model creation workflow, clarifying the distinction between estimation and indexing. The developed library in this work does not implement

[5] https://pypi.org/project/ngram-lm
[6] https://github.com/joaompfe/word-prediction
[7] The `Doc` object wraps every text unit in SpaCy, containing extra information/attributes about it.

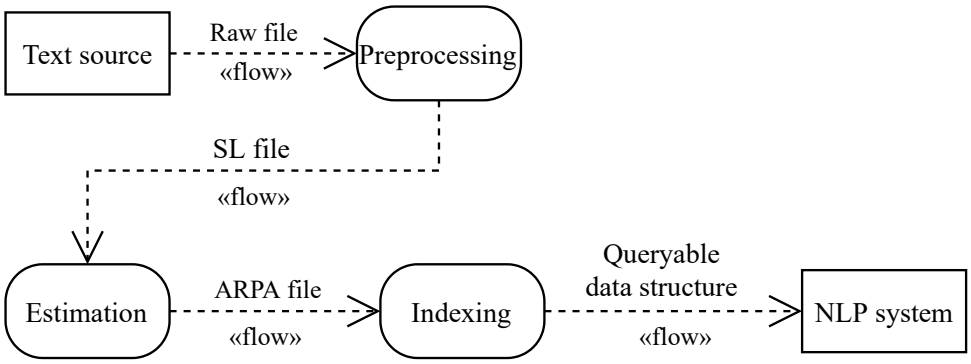

Figure 3.1: Language model creation, from raw text to a queryable data structure that can be used by an NLP system.

any estimation procedure. The preprocessing and indexing steps will be detailed in the following sections.

3.3 CETEMP preprocessing

The corpus used in this work is available for downloading at Linguateca website.[8] It contains more than 7 million sentences and more than 900 hundred unique tokens. In its original form, it seems like XML, but it does not strictly conform with XML format. As such, a first script was developed to convert the raw format to valid XML.[9] From the valid XML, two other scripts can be used if desired: one that converts it to JSON and another that converts it to a non-standard format that places one *sentence per line* and separates each token by one space. The latter format will be referred to in this work as SL format and, while non-standard, is used as input by many ARPA generators. The corpus in SL format was divided in three parts: *train* part that contains approximately 96% of the complete corpus, and *dev* and *test* parts that contain approximately 2% of the corpus each. (This division was done initially having in mind the need to empirically optimize the hyperparameters of the neural language models on the dev corpus. Considering that no neural language model was developed, the dev set was not used.) Then, from the train set, a first train ARPA was generate using the estimation procedure available in KenLM toolkit [10], which uses the modified Kneser-Ney smoothing by default, and a second was generated using the SRILM `ngram-count` program and the `cnt2arpa` script, yielding to ML estimated probabilities. These ARPA files were then used to build the trie-based language models (see next section).

Because all this preprocessing takes time (from minutes to hours, depending on the used machine), all these files and some ready-to-use models are available online for down-

[8] https://www.linguateca.pt/CETEMPublico

[9] All CETEMP preprocessing scripts are available at https://github.com/joaompfe/word-prediction/tree/master/scripts/cetemp

load through commands offered by the developed library.

3.4 Trie implementation in detail

The trie implementation of the developed toolkit is highly based on the KenLM trie implementation (which is itself based on IRSTLM trie). As KenLM, the word keys are bit packed, but no compression or quantization of the satellite values is offered. Considering that this trie targets NWP and not probability assignment, the backoff values are not stored since they are constant for n-grams with equal context and consequently, only the conditioned probabilities on that context matter when choosing the maximum probability word. It is important to note that this does not mean that using modified Kneser-Ney smoothing or maximum likelihood estimations lead always to equal NWPs. This is not true and Chapter 4 shows it empirically. While the trie structure was based on KenLM, the indexing procedure was not and any similarity is coincidence. The procedure, which is rather simple, is described below.

3.4.1 Indexing procedure

For simplicity, let $u_n = N_{1+}(w_{i-n+1}^{i})$ be the number of distinct n-grams. Then the indexing procedure used in the developed toolkit is as follows:

1. Read from the ARPA file header the number of distinct n-grams, u_n, for each $n = 1, \ldots, N$;

2. Create the vocabulary lookup array V by reading the 1-gram section of the ARPA file and hashing each 1-gram, pushing the hash value into the array. After that, sort V by hash value. With that, the id of each word is the position of its hash value in V. Each id can be uniquely represented in $\lceil \log |V| \rceil$;

3. Considering the number of distinct 1-grams from step (1), allocate the required space for the 1-gram array A_1 [$u_1(\lceil \log u_2 \rceil + 32)$ bits, where $\lceil \log u_2 \rceil$ is for the index and 32 for the probability, for each of the u_1 entries], and reread the 1-gram section, placing each 1-gram/word probability at $A_1[i]$, where i is the word id (which can be determined by binary searching the lookup array V);

4. For each $n = 2, \ldots, N-1$ do:

   (a) Allocate the required space for A_n $(u_n(\lceil \log u_1 \rceil + \lceil \log u_{n+1} \rceil + 32)$ bits, where $\lceil \log u_1 \rceil$ is for the word id, $\lceil \log u_{n+1} \rceil$ for the index, and 32 for the probability, for each of the u_1 entries);

(b) Read the n-gram section of the ARPA file, pushing into A_n a *temporary* representation of each n-gram $\boldsymbol{w}$, which is composed by the following triplet: (i) the probability value, (ii) the word w_n id (which can be determined by searching the lookup array), and (iii) the index of w_1^{n-1} in the previously build array A_{n-1} (which can be determined by depth-first searching the trie, i.e., searching for w_i on A_i, from $i = 1$ to $i = n - 1$). (This is called temporary representation because w_1^{n-1} indexes are kept only because of steps (c) and (d), being overwritten afterwards in the next iteration of (4), as explained in step (d));

(c) Sort A_n by index of w_1^{n-1} and by w_n id for equal w_1^{n-1} indexes;

(d) Having A_n well sorted, the A_{n-1} pointer indexes to A_n can now be filled, being the previously saved indexes of w_1^{n-2} in A_{n-1} overwritten by these pointer indexes. At this moment, A_{n-1} assumes its final form.

5. Allocate space for A_N (which requires $u_N(\lceil \log u_1 \rceil + \lceil \log u_{N-1} \rceil + 32)$ bits, where $\lceil \log u_1 \rceil$ is for the word id, 32 for the probability, and $\lceil \log u_{N-1} \rceil$ is only required for the temporary representation, i.e., w_1^{N-1} indexes, for each of the u_N entries) and read the N-gram section of the ARPA file, saving the temporary representation of each N-gram as in step (4b). Sort A_N as in step (4c) and fill the pointer indexes of A_{N-1} as in step (4d). With A_{N-1} well sorted, A_N can be reduced by removing the unnecessary w_1^{N-1} indexes.

Considering that sorting is done using the quick sort algorithm, the time complexity of the procedure is as follows. Step (1) takes $O(N)$; step (2) takes $O(|V| \log |V|)$; step (3) takes $O(|V| \log |V|)$; step (4), takes $\sum_{n=2}^{N-1} O(|u_n| \log |u_n|)$; and step (5) takes $O(|u_N| \log |u_N|)$. Considering these complexities and assuming $u_N > u_i$ for $i = 1, \ldots, N - 1$, the overall procedure complexity can be resumed as $O(N|u_N| \log |u_N|)$.

The memory complexity is the same as the complexity of the final structure: $O(N|u_N|)$. More concretely, both require

$$u_1(64+\text{size}(\boldsymbol{w}))+u_1(\lceil \log u_2 \rceil+32)+\sum_{n=2}^{N-1} u_n(\lceil \log u_1 \rceil+\lceil \log u_{n+1} \rceil+32)+u_N(\lceil \log u_1 \rceil+\mu+32)$$

bits, where size($\boldsymbol{w}$) gives the average number of bits required to represent the words text, and $\mu = 0$ for the final trie structure and $\mu = \lceil \log |u_{N-1}| \rceil$ for the indexing procedure. Considering $N = 3$, a trie built from the unpruned train ARPA would take approximately 557MB without the words text and 572MB with the words text.

3.4.2 Next word prediction query

The NWP query procedure can be divided in three steps: validate the context n-gram, find the context in the trie by depth-first search, and search from the respective n-gram

children the one with the highest conditioned probability. The context is passed to the procedure as a list of strings, so the first step is to convert it into a list of word ids by searching the vocabulary lookup array. If a gram of the n-gram is not found in the vocabulary, there is a backoff to the grams following the non-found gram, i.e., if the j^{th} gram is not found, the n-gram w_{n-i+1}^{i} is reduced to $w_{n-i+1+j}^{i}$. After that, the trie can be depth-first searched. If at any level of the trie the search word is not found, there is a backoff to the lower-order n-gram, with the search for this restarting from the trie root. When the final word of the n-gram is found, its children are iterated to find the maximum conditioned probability entry. From the time complexity point of view, this last step is the most expensive one, since the search is done in linear time in relation to the number of children. This is so because the entries are sorted by key and not by probability value, which impedes the use of binary search.

An especial note should be said about the second step. As mentioned, whenever a word is not found during the trie search, the search for the lower-order n-gram needs from the root, which means that in the worst-case $N, N-1, \ldots, 1$ binary searches are carried out. However, there is a different type of trie, *reverse* trie, that avoids this by saving the n-grams in the reverse order. To understand this, consider an example 3-gram `ABC`. In a reverse trie, the symbol/word `C` would be saved on the first level array, `B` on the second, and `A` on the third. The first node (where `C` is saved) would contain `C`'s probability, the second node (where `B` is saved) would contain the conditioned probability $P(\texttt{C} \mid \texttt{B})$, and the third node (where `A` is saved) would contain the conditioned probability $P(\texttt{C} \mid \texttt{AB})$. Now, to calculate $P(\texttt{C} \mid \texttt{AB})$ the trie is depth-first searched by `C`, then `B`, and then `A`, totaling 3 binary searches. However, imagining that `A` was not found in the `B`'s children, $P(\texttt{C} \mid \texttt{AB})$ would be calculated by backing off to $P(\texttt{C} \mid \texttt{B})$, whose value is immediately available in the `B`'s node, being thus not necessary to research the trie from the root. This means that in the worst case, N binary searches are carried out. While reverse tries improve query times, they can not be used for NWP. Finding the NWP given some context in a reverse trie would imply searching the full trie.

The NWP procedure described focused only on the query of only one prediction, but the developed toolkit has implemented an extra procedure to allow the query of k predictions. This procedure presents extra challenges, which start when k is greater than the number of children (which might also happen for the single prediction query when there are no children, but the solution is as simple as backing off to the lower order n-gram and search for the highest probability). When that happens, all the available children are selected as predictions, and the search for the remaining predictions happens in the lower order n-gram children. A special challenge exists at this moment since the highest probability words in these children might be already selected from the higher-order n-gram children. As such, whenever the search happens on children of lower-order n-grams, these need to be filtered according to the already selected predictions. This

challenge does not increase the time complexity nonetheless. Note however that, because a reverse trie is not used, when there is a backoff to a lower order n-gram, the trie needs to be searched from the root again. Also, when k is less than the number of children, the children entries are first sorted by probability value, and then the top-k are selected. Both facts mean that querying k predictions might become an expensive procedure if k is large.

3.5 Python API usage example

Here a sample usage of the developed Python binding and SpaCy component is given. Consult the online documentation for installation guidance and up-to-date documentation.[10]

3.5.1 Python binding

Given an ARPA file, a trie can be built like this:

```
from ngram_lm.trie import build
lm_order = 3
arpa_path = "/path/to/arpa-file"
out_path = "/the/desired/output/path/where/trie/will/be/saved"
build(lm_order, arpa_path, out_path)
```

Then, the trie can be loaded into memory and queried for NWPs:

```
from ngram_lm.trie import Trie
t = Trie(out_path)
context = ["ele", "foi"]
n_predictions = 5
predictions = t.next_word_predictions(context, n_predictions)
for p in predictions:
    print(p)
```

3.5.2 SpaCy component

Create a text-processing pipeline, e.g.:

```
from spacy.lang.pt import Portuguese
nlp = Portuguese()
```

[10] https://joaompfe.github.io/ngram-lm for Python binding and https://joaompfe.github.io/word-prediction for SpaCy component.

Create a pipeline predictor component and add it to the pipe:

```
from word_prediction.nwp import TrigramLmWordPredictor
from word_prediction.trie import Trie

@Portuguese.factory("next_word_predictor")
def create_next_word_predictor(nlp, name):
    order = 3
    t = Trie(order, "models/train-mkn-trie") # this model will only be
        available if you download the pre-built models
    nwp_pipe_component = TrigramLmWordPredictor(t)
    return nwp_pipe_component

nlp.add_pipe("next_word_predictor")
```

Process some sentences and see the NWPs:

```
sentences = ["seria muito", "demasiado para", "sem que", "com muita"]
for s in sentences:
    doc = nlp(s)
    print("'%s' NWP is: '%s'" % (s, doc._.nwp))
```

Chapter 4

Experiments

Language models are not usually an application in themselves. Instead, they are typically a component of larger systems (e.g., a virtual keyboard or speech recognition system). In the end, what matters is the performance of the application where they are embedded. When these applications are evaluated with respect to some metric, the language model they embed is evaluated *extrinsically*. Nonetheless, it might be useful to evaluate the quality of models without considering any particular application scenario. To do so, *intrinsic* tests, such as held-out likelihood or perplexity (which is just a different representation of the former), are used. These tests might differ in the way the final result is presented, but they all evaluate the model based on the same metric: the probabilities that the model assigns to the n-grams of a given test corpus. Typically, the best models assign higher probabilities to the sentences in the corpus, but the best models in intrinsic tests might not be the best in the real application (it is actually very simple to have a perfect model in intrinsic tests which is useless for any application: one that assigns always 1 to every sentence.). As such, whenever the models are to be used in some application, it is the application that should be evaluated [4].

To prove the concept that a good next word predictor can be achieved through n-grams (or 3-grams, more concretely) language models, several 3-order language models were generated from the training corpus and, then, each model was queried for NWPs using each bigram in the test corpus as prediction context. In total, each model was queried 3 538 930 times, returning $k = 10$ predictions sorted by descending order of probability for each time. The first tested models are two unpruned language models, one that uses ML estimations and the other that uses modified Kneser-Ney smoothing. The results are shown in Table 4.1, showing that for approximately 27% of the contexts, the prediction #1 of both models correctly matches the real next word. Also, considering the top 3 predictions, the models correctly predict the next word approximately $(27 + 9 + 6)\%$ of the time (3 is the typical number of predictions/suggestions given by virtual keyboards in mobile phones). Considering the complete set of 10 predictions, the next word is contained in that set 61% of the time, for both models.

	1	2	3	4	5	6	7	8	9	10	[1-10]
ML	27.42	**9.76**	**6.23**	4.45	3.36	2.68	2.21	1.88	1.64	1.43	**61.06**
mKN	**27.46**	9.66	6.22	**4.46**	**3.37**	**2.69**	**2.22**	1.88	**1.65**	**1.44**	61.04

Table 4.1: Accuracy (in percentage) of the k^{th} prediction when using ML estimations and mKN smoothing.

The accuracy achieved by both models is very good. (If one does not think the results are good, remember that the task of NWP is stochastic and not even the specialists in natural language, the humans, can correctly predict the next word 100% of the time.) However, both models take approximately 572MB, which means they can not be used on any mobile device due to memory restrictions. Nonetheless, these results show the importance of extrinsic tests since both ML and mKN models achieved very similar performances (with ML even surpassing mKN when considering the complete set of 10 predictions), while it is widely known that mKN has much superior performances than ML in intrinsic tests done in English corpus [3].

Considering the current mobile memory restrictions, two other pruned models were tested, both using mKN smoothing. One was pruned by removing every n-gram that occurred only once (which will be denoted by P1), and the other by removing every n-gram that occurred only three or fewer times in the training corpus (which will be denoted by P3). With that, the models' size was reduced from 572MB to 165MB and 72MB for P1 and P3, respectively. The results are shown in Table 4.2. In comparison with the unpruned models, there is a loss of precision for the #1 prediction of about 20% and 28% for P1 and P3, respectively. Considering the set of the top 3 predictions, the loss is about 19% and 27% for P1 and P3, respectively. Considering the complete set of 10 predictions, the loss is about 18% and 26% for P1 and P3, respectively. There is a considerable difference in accuracy between the unpruned and pruned models, but much higher is the difference between their sizes. Particularly, P3 size is almost one order lower than the unpruned versions while its accuracy is only about 27% worse. While 72MB is still expensive for mobile devices, it should be noted that no corpus preprocessing was done to remove capitalization, punctuation, or another type of noise, meaning that the models' vocabulary is much larger than it needs, and consequently the models size, as well.

	1	2	3	4	5	6	7	8	9	10	[1-10]
P1	21.81	8.11	5.29	3.79	2.84	2.26	1.91	1.57	1.37	1.20	50.16
P3	19.71	7.40	4.83	3.45	2.58	2.05	1.68	1.42	1.25	1.11	45.48

Table 4.2: Accuracy (in percentage) of the k^{th} prediction for pruned models.

Chapter 5

Conclusion

The ultimate goal of this work was to develop a pictogram predictor for AAC systems. To this end, the state-of-the-art work in language modeling was reviewed, with smoothing techniques and indexing data structures surveyed. Considering the lack of solutions to next word prediction, a small toolkit was developed, implementing a trie data structure based on the one used by KenLM. After this, several experiments were carried out for the specific task of next word prediction. In total, four different language models were tested, achieving good accuracy but occupying too much memory when considering the mobile hardware restrictions. Nonetheless, especially considering the most pruned model, it was shown that n-gram models and the developed toolkit are viable options for next word prediction.

There are, nonetheless, several weaknesses and shortcomings of the developed toolkit that should be discussed. Currently, there is no hash collision detection during the vocabulary building procedure. Detecting this is trivial (just check whether the array entry is empty or not), but resolving a hash collision is not trivial since the array has a fixed length. Also, there is no cache mechanism to avoid unnecessary trie traversals when equal contexts are given. Considering the memory restrictions of mobile devices, compression or quantization could be very important. As future work, the implementation of an Elias-Fano trie as done by Pibiri and Venturini or the implementation of quantization should be studied. Currently, the indexing procedure is done completely in-memory and is rather naive. In the future, the currently used indexing procedure should be redesigned. Considering the NWP query time complexity (especially the linearity in relation to the context' children), it might be possible to design a completely new data structure targeting NWP to reduce this complexity.

Regardless of these weaknesses, there are two general paths for future work (not mutually exclusive). The first is to fulfill the ultimate goal by mapping the work done to the particular case of pictogram prediction. The second is to keep developing the library to offer a ready-made solution for next word prediction or even general language modeling through a high-level language (e.g., Python) interface. When researching solutions for

next word prediction, it was founded that ready-made solutions for it are lacking, and while there are solutions for general language modeling, these are typically only offered through a C++ interface.

Bibliography

[1] Lalit R Bahl, Frederick Jelinek, and Robert L Mercer. "A maximum likelihood approach to continuous speech recognition". In: *IEEE transactions on pattern analysis and machine intelligence* 2 (1983), pp. 179–190.

[2] Thorsten Brants and Alex Franz. "Google web1t 5-gram corpus, version 1". In: *Linguistic Data Consortium, Philadelphia, Catalog Number LDC2006T13* (2006).

[3] Stanley F Chen and Joshua Goodman. "An empirical study of smoothing techniques for language modeling". In: *Computer Speech & Language* 13.4 (1999), pp. 359–394.

[4] Jacob Eisenstein. *Natural language processing*. 2018.

[5] Peter Elias. "Efficient storage and retrieval by content and address of static files". In: *Journal of the ACM (JACM)* 21.2 (1974), pp. 246–260.

[6] Robert Mario Fano. *On the number of bits required to implement an associative memory*. Massachusetts Institute of Technology, Project MAC, 1971.

[7] Marcello Federico, Nicola Bertoldi, and Mauro Cettolo. "IRSTLM: an open source toolkit for handling large scale language models". In: *Ninth Annual Conference of the International Speech Communication Association*. 2008.

[8] Edward Fredkin. "Memory Trie". In: *Communications of the ACM* 3.9 (1960), pp. 490–499.

[9] Andrew Hard et al. "Federated learning for mobile keyboard prediction". In: *arXiv preprint arXiv:1811.03604* (2018).

[10] Kenneth Heafield et al. "Scalable modified Kneser-Ney language model estimation". In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 2013, pp. 690–696.

[11] I. J. Good. "the Population Frequencies of Species and the Estimation of Population Parafeters". In: *Biometrika* 40.3/4 (1953), pp. 237–264.

[12] Harrold Jeffreys. *Theory of probability 2nd ed.* 1948.

[13] William Ernest Johnson. "Probability: The deductive and inductive problems". In: *Mind* 41.164 (1932), pp. 409–423.

[14] Reinhard Kneser and Hermann Ney. "Improved backing-off for m-gram language modeling". In: *1995 international conference on acoustics, speech, and signal processing*. Vol. 1. IEEE. 1995, pp. 181–184.

[15] Kamran Kowsari et al. "Text classification algorithms: A survey". In: *Information* 10.4 (2019), p. 150.

[16] Pierre-Simon Laplace. *Philosophical essay on probabilities*. Trans. by Andrew I Dale. Springer, 1995.

[17] George James Lidstone. "Note on the general case of the Bayes-Laplace formula for inductive or a posteriori probabilities". In: *Transactions of the Faculty of Actuaries* 8.182-192 (1920), p. 13.

[18] Sharmila Mani et al. "Real-time optimized n-gram for mobile devices". In: *2019 IEEE 13th International Conference on Semantic Computing (ICSC)*. IEEE. 2019, pp. 87–92.

[19] Hermann Ney and Ute Essen. "On smoothing techniques for bigram-based natural language modelling". In: (1991), pp. 825–828.

[20] Hermann Ney and Ute Essen. "On smoothing techniques for bigram-based natural language modelling". In: *Proceedings of the IEEE International Conference on Acoustics* Speech and Signal Processing (1994), pp. 1–4.

[21] Adam Pauls and Dan Klein. "Faster and smaller n-gram language models". In: *Proceedings of the 49th annual meeting of the Association for Computational Linguistics: Human Language Technologies*. 2011, pp. 258–267.

[22] Giulio Ermanno Pibiri and Rossano Venturini. "Efficient data structures for massive n-gram datasets". In: *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2017, pp. 615–624.

[23] Giulio Ermanno Pibiri and Rossano Venturini. "Handling massive n-gram datasets efficiently". In: *ACM Transactions on Information Systems (TOIS)* 37.2 (2019), pp. 1–41.

[24] Andreas Stolcke. "SRILM - An extensible language modeling toolkit". In: *7th International Conference on Spoken Language Processing, ICSLP 2002* (2002), pp. 901–904.