Tri-Gram Language Model and Byte-Pair Encodding Language Identification and Similarity

Jean Lucien Randrianantenaina
Applied Mathematics, Machine Learning and Artificial Intelligence
Stellenbosch University
161 Decanting Building, Hammanshand Road.
Email: lucien.randrianantenaina@aims-cameroon.org

***Abstract*—There are many languages on the earth. It is not always easy to recognize them especially when they are similar as some language is a derivation of other languages. Given, a text document with an unknown language, how would we know in which language was it written? Language Model can be used for this purpose of identification.**

I. Introduction

II. Data Preprocessing: text normalization

To build the language identification we are given five non-normalized corpus with the following language:

- Afrikaans (af)
- English (en)
- Deutch (nl)
- Xhosa (xh)
- Zulu (zl)

Each of them with a normalized validation data to check and tune some hyperparameter of our model. Lastly, we have a validation set to measure the final performance of our language identification.

Thus to normalize the training data to match the form validation set we apply the following operation:

1) Each paragraph is splited by detecting where we have '.', '!' or '?' followed by space and Capital letter. Then, each sentence is placed on a single line in the normalized file data,
2) Remove leading and trailing spaces,
3) Replace all:
   - diacritics into its normal form,
   - digits by 0,
4) Expand all abbreviation and acronyms by inserting a space,
5) The text is then transformed into a lower case.

To perform these operations we use the `re` package of python for regular expression and `unicodedata` for diacritics.

Once the data is ready, we can use them to build our language model.

III. Language Modelling

This section describes the how we build the character trigram language model, how we evaluate and generate text from it. The model will use the following vocabulary:

$$\mathcal{V} = \{<, >, \texttt{spaces}, \texttt{0}, \texttt{a}, \texttt{b}, \ldots, \texttt{z}\} \tag{1}$$

which are obtained from the normalized corpus, where $<$ indicate the beginning of a sentence and $>$ its end.

A. Trigram model

With a Markov assumption, given a sequence of two characters (w_{t-2}, w_{t-1}), a trigram model predicts the probability of a third word w_t given the previous two by:

$$P(w_t|w_{t-2:t-1}) = \frac{P(w_{t-2:t})}{P(w_{t-2:t-1})} \tag{2}$$

and the likelihood of a given sentence is defined by:

$$P(w_{1:T}) = \prod_{i=1}^{T} P(w_i|w_{i-2}w_{i-1}) \tag{3}$$

However, we do not have access to this true probability, as we cannot have all the possible trigrams in our training set. Therefore we have to estimate it, using maximum likelihood estimation.

B. Probability estimation

Since $P(w_t|w_{t-2:t-1})$, is not accessible we substitute it by its maximum likelihood estimation defined by:

$$P_{\text{MLE}}(w_t|w_{t-2:t-1}) = \frac{C(w_{t-2:t})}{C(w_{t-2:t-1})} \tag{4}$$

Where $C(w_{t-2:t})$ is the count of the trigram $(w_{t-2:t})$ in the training data, and $C(w_{t-2:t-1})$ is the count of the bigram $(w_{t-2:t-1})$. Once again, some trigram will not be present, leading to a zero count, and then a zero probability for a particular sentence which does not make sense.

To handle the zero count trigram, we can use various smoothing technics, here we use the add-k smoothing which consist of adding a fractional count when computing the probability as follows:

$$P_{\text{Add}-\text{k}}(w_t|w_{t-2:t-1}) = \frac{C(w_{t-2:t}) + k}{C(w_{t-2:t-1}) + k|\mathcal{V}|} \tag{5}$$

When $k = 1$ it is called Laplace smoothing, which is the default value in our implementation. The value of k is then tuned on the validation set using a grid search, and consider the one that maximize the likelihood.

Using Laplace-smoothing is fine for our language identification purpose, but it can move to punch mass to the unseen trigram, and this is one of the reasons that we tune it.

Another alternative is to use a mixture of several models, by using the probability defined by:

$$P_{\text{INT}}(w_t|w_{t-2:t-1}) = \lambda_i P(w_i|w_{t-2:t-1}) + \lambda_2 P(w_t|w_{t-1}) + \lambda_3 P(w_t) \tag{6}$$

This technic is called Interpolation smoothing, the value of λ_i can be tuned as well, and it must sum to one.

Both of these technics are suitable for our problem, as we aim to predict in wich language is used in a given sentence.

C. Model evaluation and hyperparameter tunning

Now, with the basis of the model, we need a metric to measure how good it is. The most common metric in NLP is the perplexity, defined by the probability of the data assigned by the language model, normalized by the nmber of words:

$$PP(W) = P(w_{1:T})^{-\frac{1}{T}} = \sqrt[T]{\frac{1}{P(w_{1:T})}} \tag{7}$$

where a lower perplexity reflects a better model. In practice we use the log probability, since the perplexity can be computed

$$PP(W) = P(w_{1:T})^{1/T} = 2^{-\frac{1}{N}\log_2 P(w_{1:T})} with: \tag{8}$$

In our implementation we use the natural log, so the two is replaced by e. Additionally, minimizing the perplexity is equivalent to maximizing the probability, and we use this fact to optimize our hyperparameter.

D. Text Generation

With the hyperparameter-tuned models, we can now generate text character by character. The code was designed to generate text from nothing or from a specific starting text. The text generation process is as follows:

- With a starting text: Split the text into a list of characters and add the starting marker '¡' at the beginning.
- From nothing: We start with the starting sentence, then we use the bigram model to form the first two characters, i.e., $(<, \bullet)$.
  Repeat the until we meet the ending markers $>$:
  - Compute all $p(x|w_{t-2}w_{t-1})$, for all $x \in \mathcal{V}$
  - Normalize these probabilities to ensure they form a valid probability distribution.
  - Use `np.random.multinomial` to sample the next character according to the normalized probabilities.

Note that the process described above is also used for the bigram model at the beginning if we do not specify any starting character. The probability is computed according to the specific smoothing method.

Another utility of this metric is to look how similar a language A on language B. We can perform that by consider a corpus from A and compute the perplexity on the language model B, and vice versa. If the two languages are similar, it should yield lower perplexity on both sides, otherwise it l be high.

IV. Language identification

this section described how we use our set of trigram models to identify the language for a given sentence by doing the following steps:

- Compute the perplexity of the sentence using each of the trained language models.
- The predicted label (language) is then determined by the model with the lowest perplexity.

Then we apply that on the whole test set in order to determine the accuracy of our language identification. The accuracy is computed by counting the number of correct language.

V. Byte-Pair Encoding and Language Similarity

We have already mentioned that the perplexity can be used to analyse the similarity of two languages. Here, we compare the vocabulary generated by the Byte-Pair Encoding (BPE) algorithm to see how similar they are.

The BPE is an algorithm that allows automatic subword token learning from a given training corpus by following the process bellow:

1) Initialization:
   - Initialize the tokens at the character level.
   - Create the initial vocabulary $\mathcal{V}$ using the unique characters from the tokens.
2) Repeat for k times:
   - Find the most frequent pair of adjacent tokens t_L, t_R in the corpus.
   - Create a new token $t_{new} = t_L + t_R$ (merge the pair).
   - Add the new token t_{new} to the vocabulary $\mathcal{V}$.
   - Replace all occurrences of t_L, t_R with t_{new} in the corpus.
   - Record the merge into the history.

The resulting vocabulary $\mathcal{V}$ will have $k + |\text{Initial Characters}|$ tokens after the k iterations.

If the algorithm is run for a very large number of merges, the vocabulary may contain complete words from the training corpus, which may not be the most useful representation.

Therefore, the appropriate number of merges should be chosen to obtain a compact set of sub-word units that can effectively represent the original text corpus. This allows the model to handle rare and out-of-vocabulary words by decomposing them into these learned sub-word units.

VI. Result and Discussion

VII. Conclusion

References

[1] Herman Kamper. *NLP817*. https://www.kamperh.com/nlp817/, 2022–2024.

[2] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. 3rd edition, Stanford University and University of Colorado at Boulder, 2024.

[3] *Compare languages - online genetic proximity calculator*. http://www.elinguistics.net/Compare_Languages.aspx, 2023.