Natural Language Processing: Assignment 1

Johannes Gontrum

1 Tokenization

Tokenization is one of the fundamental steps in building a NLP pipeline, since higher level tasks like parsing rely on properly segmented text. Therefore it is very important to use a robust and precise yet fast tokenizer. Since regular expressions are internally compiled into very fast finite state automata (see Sipser (2006)), they are a frequently used tool for this task.

1.1 Building the regular expression

Regular expressions become harder to read and debug with increasing length. For that reason I developed my expression as individual parts that are afterwards combined with the *OR* operator (|) to form the final regular expression. If the sub expressions overlap and would (partially) match the same string, their order matters. Therefore I start with the most specific cases like hard-coded common abbreviations (e.g. *Ltd.*) and gradually move towards general cases like single symbols (e.g. *%, $, &*) and eventually tokens containing alphanumeric characters (\w+):

```
Common Abbrev.|Suffixes|Multi-Symbols|Numbers|Abbrev.|Symbols|General
```

As an example, I want to segment the string *8.14%* into the tokens *8.14* and *%*. The initial expression \w+ would result in *8* and *14*, splitting the number and ignoring the percentage character and dot. I introduce a new expression to capture numbers with a delimiter as a special case (\d+\.\d+) and add the percentage character and the dot to the list of symbols.

The most challenging part was to 'stem' negation suffixes, so that *couldn't* would be tokenized as *could* and *n't*, since \w+ would match *couldn*. Intuitively, one would add *n't* to the beginning of the regular expression as a special case. However, this is not sufficient since the *n* in *n't* was already captured before. To solve this problem, I added a *negative lookahead* to the general case expression. This way, we move character by character through the text checking if the following string does not match a *'t*:[1]

$$\texttt{\textbackslash w+} \quad \Rightarrow \quad \texttt{(?:\textbackslash w(?!'t))+}$$

The final regular expression including sentence boundary detection with detailed explanations can be found in Figure 1.

1.2 Evaluation

My complete regular expression archived a precision of 1.0 and a recall of 0.99 on the development data, only failing recognize a new sentence after *Ph.D.*.

However, since the expression was build to perform as good as possible on the given development data, I expect a worse performance on general text. There are multiple parts that will not generalize well: For example, the list of common abbreviations is far from being complete: November (*Nov.*) might be captured, but not any other month, and abbreviations for units or companies are missing as well. Furthermore, in several cases I use `[A-Z]` to specifically capture the letters A to Z. Non-English characters like *Ä* will not be recognized. However, these describe infrequent cases and I payed attention to add more cases to the hard-coded lists than necessary for the first task. Therefore I do not expect my performance to fall below a precision/recall of 0.95.

After testing on the new test set, I see that I predicted correctly: Precision and recall are 0.995, each matching $924/928$ cases. As expected, there are cases of undersplitting with abbreviations that were not included in my expression: *Inc.* and *etc.*, also I forgot to add brackets to my list of symbols. This highlights a disadvantage of hard-coding words into a tokenizer: If one does not formulate a generalizing rule, such a list will never be complete. On the other hand, it is sometimes not possible to distinguish one-word abbreviations ending with a period from the end of a sentence—at least not with a basic tool like regular expressions.

[1] Adding a `?:` prevents the creation of a capture group.

2 Language Modelling

2.1 Maximum Likelihood Estimation

Maximum Likelihood Estimation (MLE) is defined in Equation 1 and 2 in Figure 2 (see Manning et al. (1999)). Given the number of training instances N, the probability of an n-gram is the number of occurrences of the sequence of tokens divided by N. In an existing context $w_1...w_{n-1}$ we utilize Equation 2 to calculate the probability of the following word w_n by dividing the occurrences of the whole sequence by the counts of the preceding $n-1$ words. Thus we *maximize* the parameters of our model, namely the probability that an n-gram occurs, to give its occurrences in the training corpus the highest probability.

However, applying MLE to natural language processing has major drawbacks: MLE assigns a zero probability to all unseen events. Not only do we fail to get the probability of an unseen word or n-gram: A single zero probability prevents us from obtaining the likelihood of a longer text, since it is calculated by multiplying the probabilities of all the n-grams it contains.

2.2 Smoothing for MLE

Smoothing is an approach to handle unseen words and n-grams by redistributing the probability mass of the language model to include events that were unknown during the training.

The most basic form of smoothing is called *Laplace smoothing*. During the training a count of 1 is added to every event and also to potentially unknown tokens and n-grams. Events that are seen once during the training therefore receive a count of 2, while 1 will be assigned to all unseen ones. When computing the probabilities, the additional counts also have to be added to the denominator so that the probability of all unknown events will be $\frac{1}{2N}$, where N is the number of events. Laplace smoothing can also be adapted to use other values instead of 1, as seen in the experiments shown in Figures 3 and 4.

A more advanced smoothing technique are *back-off* models: One trains models for all n-grams with $n \leq 1$ and tries to use the best one available. As an example, let us assume we train a 4-gram model on our Sherlock Holmes corpus and evaluate it on the test set *His Last Bow*. When we encounter the 4-gram ”elementary class of deduction”, we find that it has a zero probability in our model, as it does not exist in the training corpus. We then reduce n by one and try the 3-gram ”class of deduction”. It also has a probability of 0, so we move to the bi-gram and discover a non-zero probability for ”of deduction”. This value will be used, but is lowered by multiplying it with a penalty, as it is not as accurate as a 3 or 4-gram.

The *Kneser-Ney* technique that is used in the provided `SRILM` tool also improves the handling of unigrams: Normally, very frequent unigrams would have the highest probability, though some of them might only occur in narrow contexts. *Kneser-Ney* discounting therefore prefers unigrams that are frequent, but also used in various contexts (see Jurafsky and Martin (2009)).

2.3 Evaluation

I am evaluating the language models trained on the collection of Sherlock Holmes texts on the test sets *LASTBOW* (other Sherlock Holmes stories), *LOSTWORLD* (other stories by the same author) and *OTHERS* (stories from the same time and place, but written by other authors). To show the differences in performance, I compare the two mentioned smoothing methods and vary n to $1 \leq n \leq 9$. The results listed in Figure 3 clearly show that the naive *Laplace* smoothing performs drastically worse than the advanced *Kneser-Ney* back-off smoothing.

As expected, *LASTBOW* always performs best, since it is very close to the training domain. However, I am surprised by the similarity of the other two sets. A reason for this might be the amount of out-of-vocabulary (OOV) tokens. While 3.35% tokens in *LASTBOW* are OOV, *LOSTWORLD* contains 5.92% and *OTHERS* 6.57%. Another cause might be a problem in the experiment setup: While comparing both Sherlock Holmes sets, I discovered that many paragraphs e.g. in the stories *Resident Patient* and *Cardboard Box* are almost identically. This means we are (partially) evaluating on the same data that we trained on, producing unreasonably good results.

```
(
  "(?P<sent>"                    # Sentence segmentation group:
  "(?<=[.!?])[≪≫‘\`´'\"]"        # Sentence ends in quote
  "|[.!?](?=\s|$))"              # Must be followed by a blank or new line
  "(?!\s*[a-z])"                 # May not be followed by a lower character

  "|(?P<tok>"                    # Tokenization group:
  "(?:Nov|Dr|Ltd|Corp|Mr|Mrs|Co|Mass)\." # Common abbreviations
  "|n't|'m|'re|'s|'ve|'d|'ll"             # (Negation-) Suffixes
  "|[-'`´‘]{2,}",                # Symbols occurring multiple times
  "|(?:\d+[.,]\d+)"              # Numbers with delimiter
  "|(?:\w+\.){2,}"               # Groups of abbreviations like `Ph.D.`
  "|[A-Z]\."                     # Single char abbreviations like `J.`
  "|[≪≫‘\`´'\",;:.!?%\$&]"        # Single symbols
  "|(?:\w"                       # General token expression:
  "(?!'t)"                       # > Strip negation `'t`
  "(?:-(?!-))?)+)"               # > Allow only a single dash in a token
)
```

Figure 1: The complete regular expression for tokenization and sentence segmentation. Using group names allows me to handle a matched token differently than a detected sentence boundary.

$$P_{\text{MLE}}(w_1...w_n) = \frac{C(w_1...w_n)}{N} \tag{1}$$

$$P_{\text{MLE}}(w_n|w_1...w_{n-1}) = \frac{C(w_1...w_n)}{C(w_1...w_{n-1})} \tag{2}$$

Figure 2: Maximum Likelihood Estimation equations. N is the number of training instances, $C(x)$ the number of occurrences of x.

Laplace Smoothing

n	LASTBOW	LOSTWORLD	OTHERS
1	425.38	495.36	449.64
2	**130.829**	**196.53**	**187.307**
3	143.26	211.55	202.37
4	152.44	220.35	211.4
5	154.19	221.65	212.34
6	154.52	221.79	212.39
7	154.66	221.82	212.39
8	154.67	221.82	212.39
9	154.67	221.82	212.39

Back-off Smoothing

n	LASTBOW	LOSTWORLD	OTHERS
1	426.11	495.82	449.98
2	105.64	154.41	151.44
3	**96.1474**	**143.298**	**142.903**
4	96.92	144.09	144.05
5	97.58	144.68	144.71
6	97.8	144.82	144.79
7	97.88	144.86	144.79
8	97.91	144.88	144.78
9	97.92	144.9	144.78

Figure 3: Perplexity values of a language model trained for various n-grams. *Laplace smoothing* uses a value of 0.01, the back-off type is *Kneser-Ney discounting*. The results are visualized in Figure 4.

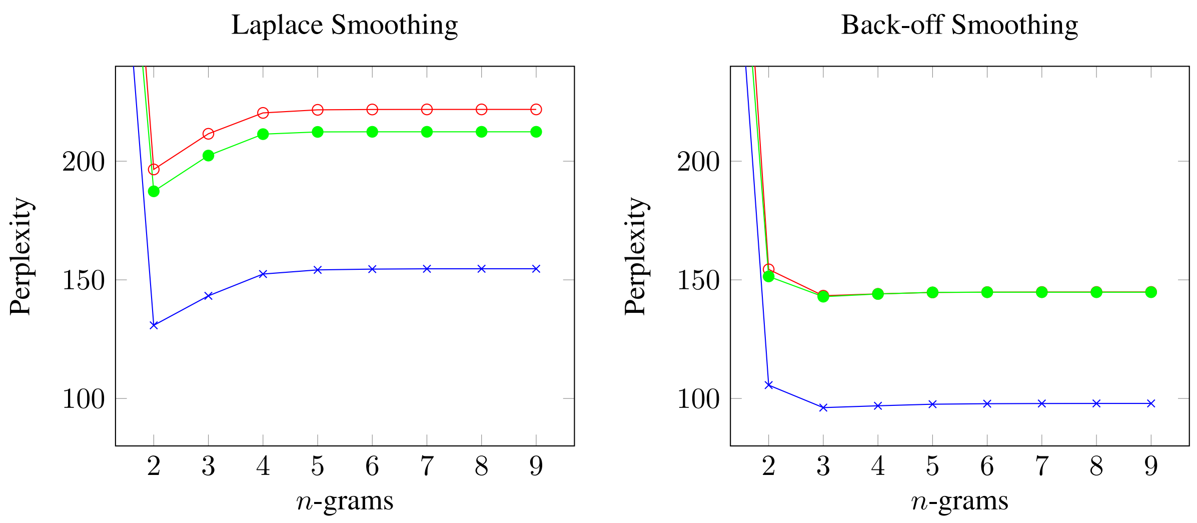

Figure 4: Evaluation of language models trained with different smoothing methods: *Laplace smoothing* with a value of 0.01, and *Kneser-Ney discounting*, a form of *back-off* smoothing. Test sets: *His Last Bow* in blue, *The Lost World* in red, *Other Authors* in green.

References

Daniel Jurafsky and James Martin. *Speech and language processing: An introduction to natural language processing*. Prentice Hall, 2009.

Christopher D Manning, Hinrich Schütze, et al. *Foundations of statistical natural language processing*, volume 999. MIT Press, 1999.

Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.