Martin Isaksson
Elias Al-Tai 2016-12-01

Exercise 2

Content

This report consists of an introduction to *N-grams*, the approach to solve the exercise, the procedure to solve the exercise, the results from the built program and of course a discussion of the results with a conclusion of the whole exercise.

N-grams

In the manner of NLP (Natural Language Processing) it is very common to work with different language models (LM). One language model that is very common (and useful) is called *N-grams*. This is a simple but durable model. It is handy to use when the input is noisy and ambiguous, like for speech recognition. It is also used in machine translation, spelling correction, handwriting recognition, POS-tagging, word similarity, etc.

N-grams is used to find out the probability of that the *N* previous words, before the word of interest, is in a sequence and appearing in the text. So to calculate the probability of a sequence of words, the chain rule of probability is used, $p(w_1,w_2,…,w_n)=p(w_1)*p(w_2|w_1)*…*p(wn|w_1,…,w_{n-1})$. Thanks to the *Markov assumption*, it is not necessary to look too far into the past. So *N-grams* only consider *N* previous units, e.g. three words in a sequence (i.e. probability of a word w_n has two certain word
w_{n-1},w_{n-2} in sequence before the word occurs), also known as *trigrams*, would have the probability $p(w_n|w_1,…,w_{n-1})=p(w_n|w_{n-2},w_{n-1})$. To calculate the probabilities in practice, is done in the following way:

- *1-gram*, or *unigram*, equals *Count(w)/"total number of words in the corpus"*
- *2-gram*, or *bigram*, equals $Count(w_{i-1},w_i)/Count(w_{i-1})$
- *3-gram*, or *trigram*, equals $Count(w_{i-2},w_{i-1},w_i)/Count(w_{i-2},w_{i-1})$

Let's see an example:

The cat eats fish. The car is blue. The snow is white.

In this small text, the *2-gram* or *bigram* of the two words *The cat*, i.e. p(*cat*|*The*), would be 1/3. Even though it is called *N-gram*, *N* often never gets larger than 3 in practice (it can be 4 or 5 in some cases, but almost never more than that).

Especially in the *trigram* case it is quite common that some sequences never occur in a text. That means that those probabilities will be zero. When computing the *order N-1 model*, also known as the *entropy* or *H*, the total value will be zero if some probabilities are zero, since a multiplication is used between the probabilities to calculate the entropy. To avoid this practical problem, one can *smooth* the model, which will be seen how to do further on in this report.

Martin Isaksson
Elias Al-Tai 2016-12-01

Approach

The way to tackle the challenge of computing the entropy and the *N-gram models*, is to use the general known NLP-techniques, such as word-tokenizer, to retrieve tags and tagged words from a corpus and to calculate the counts corresponding to the *unigram*, *bigram* and *trigram*.

From these counts, one can easily calculate the entropy for each case (*unigram*, *bigram* and *trigram*). The equation for the entropies are, in respective order:

$$H = -\sum_{x} p(x) log p(x)$$

$$H = -\sum_{x} p(x) \sum_{y} p(y|x) log p(y|x)$$

$$H = -\sum_{x} p(x) \sum_{y} p(y|x) \sum_{z} p(z|xy) log p(z|xy)$$

Then to calculate the entropy for the *trigram* of the POS-tags is trivial and should be done as with the words.

The *perplexity*, which is a measure to compare different statistical models, is calculated from the equation:

$$Perplexity = 2^H$$

Then as a final step, a *smoothing* step needs to be done on the *trigrams*. This should be done on different setups and compare their different *perplexities*.

The *smoothing* is very important. If some word-sequence doesn't appear in the text, this will get a zero probability as mentioned above. Smoothing can be done in several ways, e.g. Laplace, Held-Out, Good Turing, back-off techniques (i.e. when *trigram* gives zero, one should go back and use *bigram*). In this exercise the smoothing is performed by using a back-off technique, but without reducing context. Instead of performing the *trigram* on only words, p(x,y,z), it will be combined words and POS-tags of some of the words, p(x',y,z) or p(x',y',z) (where x,y,z is words and x',y' is the tags of the respective words).

Procedure

The procedure followed the approach very well. First the text was word-tokenized. These words were then used to compute the *unicount*, *bicount* and *tricount*. These are the counts of the words, to be used when calculating the different *N-grams*.

Unigram & Entropy

The *unigram* (p(x)) is then computed by dividing each word by the total number of words.

Then the entropy is calculated by using the *unigram* in the *H*-formula mentioned above.

Bigram & Entropy

The *bigram* (p(y|x)) is calculated by taking each *bicount* and divide it by the corresponding *unicount*, as described above.

Then the entropy is calculated by using the *bigram* in the *H*-formula mentioned above.

Trigram & Entropy

The *trigram* (p(z|xy)) is calculated by taking each *tricount* and divide it by the corresponding *bicount*, as described above.

Then the entropy is calculated by using the *trigram* in the *H*-formula mentioned above.

The *perplexity* of the *trigram* is calculated as mentioned above, by the use of the equation:

$$Perplexity = 2^H$$

, where *H* here corresponds to the entropy of the *trigram*.

This whole procedure of computing the *trigram* of the words, the entropy and *perplexity* is done in the exactly same way with the POS-tags. The difference here is that the input, i.e. the set of POS-tags, is done in three different ways. First the full set, then half of the set and final a quarter of the set is used as input.

Smoothing

As described in the approach, the *trigram* is calculated not with only words, or only tags, but a combination. The first case is with a <tag,word,word>-sequence and then with a <tag,tag,word>-sequence and finally a <word,word,word>-sequence. The *perplexity* is calculated (as above) for all the three cases and compared. This is then repeated for half of the corpus and a quarter of the corpus.

Results

Unigram

H=7.79

Perplexity=221.44

Bigram

H=2.17

Perplexity=4.51

Trigram

H=0.00

Perplexity=1

Trigram for Tags and Different Corpus-Sizes

	Full Corpus	½ Corpus	¼ Corpus
Perplexity	1.98	1.75	1.60

Table 1 Trigrams for tags, with different sizes of the corpus.

Trigram with Smoothing and Different Corpus-Sizes

Perplexities	Full Corpus	½ Corpus	¼ Corpus
<x,y,z>	1	1	1
<x',y,z>	5.49	4.44	3.65
<x',y',z>	27.20	21.59	16.41

Table 2 Trigram with the smoothing and without smoothing.

Discussion & Conclusion

The most interesting part of this exercise/lab is the case when calculating the *trigram*. The infrequent sequences have so low probability that Python rounds it towards zero. And since this is probability, it is calculated by multiplication and the whole thing gets zero. This is something that needs to be prevented, by e.g. smoothing. There are several techniques of smoothing. Just to mention some, there are Laplace, Held-Out, Good Turing and back-off techniques. In this exercise/lab, a back-off technique has been used. The back-off technique that has been used has made sure that we don't lose context, i.e. we don't go from e.g. *trigram* to *bigram* when the *trigram* gets zero.

For *unigram* and *bigram* it works fine without smoothing. But the *trigram* becomes zero as expected, when not taking smoothing into consideration.

When a *N-gram* is zero, its *perplexity* gets 1. It is important to understand that this isn't a desirable value (even though the *perplexity* should be low for good indications).

This was when considering words. When considering POS-tags it doesn't get zero, i.e. the *perplexity* doesn't get 1. This is because it doesn't exist as many different tags as it exists different words. This pattern can be confirmed when decreasing the size of the corpus.

When doing the same thing but for the sake of smoothing, as can be seen in *Table 2,* the conclusion of the results is obvious. With no smoothing, the *perplexity* is 1, which is bad. Then when consider the case <x',y,z> the result gets much better. This means that the smoothing with this back-off technique works, and that we haven't lost the context. But when using <x',y',z> the *perplexity* gets worse, which isn't that odd, since a word can have many different tags. And if two tags in a sequence is followed by a certain word (which can have different tags), this can be seen as over-smoothing.

The *perplexity* is used to compare different statistical models. It is a measure of how well a probabilistic distribution (or probability model) predicts a sample. The lower value of the *perplexity*, the better the probabilistic distribution (or probabilistic model) predicts the sample. In *Table 2* it can be seen from the results that the best achieved result with this corpus, is when using <x',y,z>, and more specially with the quarter size of the corpus.