4 Counting Words

4.1 Counting Words and Word Sequences

We saw in Chap. 2 that words have specific contexts of use. Pairs of words like *strong* and *tea* or *powerful* and *computer* are not random associations but the result of a preference. A native speaker will use them naturally, while a learner will have to learn them from books – dictionaries – where they are explicitly listed. Similarly, the words *rider* and *writer* sound much alike in American English, but they are likely to occur with different surrounding words. Hence, hearing an ambiguous phonetic sequence, a listener will discard the improbable *rider of books* or *writer of horses* and prefer *writer of books* or *rider of horses* (Church and Mercer 1993).

In lexicography, extracting recurrent pairs of words – collocations – is critical to finding the possible contexts of a word and citing real examples of its use. In speech recognition, the statistical estimate of a word sequence – also called a **language model** – is a key part of the recognition process. The language model component of a speech recognition system enables the system to predict the next word given a sequence of previous words: *the writer of books, novels, poetry,* etc., rather than of *the writer of hooks, nobles, poultry*.

Knowing the frequency of words and sequences of words is crucial in many fields of language processing. In addition to speech recognition and lexicography, they include parsing, semantic interpretation, and translation. In this chapter, we introduce techniques to obtain word frequencies from a corpus and to build language models. We also describe a set of related concepts that are essential to understand them.

4.2 Words and Tokens

4.2.1 What Is a Word?

The definition of what a word is, although apparently obvious, is in fact surprisingly difficult. A naïve description could be a sequence of alphabetic characters delimited by two white spaces. This is an approximation. In addition to white spaces, words can end with a comma, a question mark, a period, etc. Words can also include dashes and apostrophes that, depending on the context, have a different meaning. They may vary according to the language. Compare the French word *aujourd'hui* 'today', which forms a single word, and *l'article* 'the article', where the sequence of an article and a noun must be separated before any further processing.

In corpus processing, text elements are generally called **tokens**. Tokens include words but also punctuation, numbers, abbreviations, or any other similar type of string. Tokens may mix characters and symbols as:

- numbers: 9,812.345 (English and French from the 18th–19th century), 9 812,345 (current French and German) 9.812,345 (French from the 19th–early 20th century)
- dates: 28/02/1996 (French and British English), 2002/11/20 (Swedish)
- abbreviations and acronyms: km/h, m.p.h., S.N.C.F
- nomenclatures: A1-B45, /home/pierre/book.tex
- destinations: Paris–New York, Las Palmas–Stockholm, Rio de Janeiro–Frankfurt am Main
- telephone numbers: (0046) 46 222 96 40
- tables
- formulas: $E = mc^2$

The definition of what is a sentence is also tricky. As in the case for words, a naïve definition would be a sequence of words ended by a period. Unfortunately, periods are also ambiguous. They occur in numbers and terminate abbreviations, as in *etc.* or *Mr.*, which makes sentence isolation equally complex. In the next sections, we examine techniques to break a text into words and sentences and to count the words.

4.2.2 Breaking a Text into Words: Tokenization

Tokenization breaks a character stream, that is, a text file or a keyboard input, into tokens – separated words – and sentences. In Prolog, it results into a list of atoms. For this paragraph, such a list looks like:

```
[['Tokenization', breaks, a, character, stream, (,),
that, is, (,), a, text, file, or, a, keyboard, input,
(,), into, tokens, -, separated, words, -, and,
sentences, '.'], ['In', 'Prolog', it, results, into,
a, list, of, atoms, '.'], ['For', this, paragraph,
(,), such, a, list, looks, like, :]]
```

Tokenization is a necessary step to morphological and syntactic parsing since these analyses consider words or sentences most of the time. A tokenizer can also remove formatting instructions, such as XML tags, if any.

This section introduces tokenization techniques. For sake of simplicity, we consider that words are contiguous segments of alphanumeric characters and that other symbols mark a separation. We can then define a tokenizer by a grammar:

- A token is a sequence of alphabetic characters or digits.
- Other characters mark the token termination and consist of carriage returns, blanks, tabulations, punctuation signs, or other ASCII symbols or commands.
- A sentence is a sequence of tokens ended by a period, a colon, a semicolon, an exclamation point, or a question mark.

We provide an implementation of tokenization using Prolog and Perl. Perl is generally faster and is well suited to process large quantities of text.

4.3 Tokenizing Texts

4.3.1 Tokenizing Texts in Prolog

A Basic Program. The tokenization program `tokenize/2` takes a list of character codes as input and returns a list of tokens. The predicate `char_typ/2` determines the type of a character code: alphanumerical, blank, or other. It uses the Latin 1 character set (charset) plus some Windows extensions. The first `tokenize/2` rule corresponds to the termination condition. The second `tokenize/2` rule tests the type of the head of the list. It skips the blanks. When it reaches an alphanumerical character in the third rule, it calls `make_word/5`, which builds a word out of next letters or digits in the list. When `tokenize/2` encounters another symbol in the fourth rule, it makes a single token out of it.

You can use the `read_file/2` predicate from Appendix A, "An Introduction to Prolog," to read the character codes from a file.

```
% tokenize(+CharCodes, -Tokens)
%  breaks a list of character codes into
%  a list of tokens.
tokenize([], []).
tokenize([CharCode | RestCodes], Tokens) :-
  char_typ(CharCode, blank),
  !,
  tokenize(RestCodes, Tokens).
tokenize([CharCode | CharCodes], [Word | Tokens]) :-
  char_typ(CharCode, alnum),
  !,
  make_word(CharCode, alnum, CharCodes, WordCodes,
    RestCodes),
  name(Word, WordCodes),
  tokenize(RestCodes, Tokens).
tokenize([CharCode | CharCodes], [Char | Tokens]) :-
  !,
  name(Char, [CharCode]),
  tokenize(CharCodes, Tokens).
```

```
% make_word(+CharCode, +Type, +CharCodes, -WordCodes,
%     -RestCodes)
make_word(CharCode1, alnum, [CharCode2 | CharCodes],
    [CharCode1 | WordCodes], RestCodes) :-
  char_typ(CharCode2, alnum),
  !,
  make_word(CharCode2, alnum, CharCodes, WordCodes,
    RestCodes).
make_word(CharCode, alnum, RestCodes, [CharCode] ,
    RestCodes).

% char_typ(+CharCode, -Type)
%  Returns the type of CharCode.
%  There are several cases:
%  Blanks
char_typ(CharCode, blank) :-
  CharCode =< 32,
  !.
%  Lower-case letters without accent
char_typ(CharCode, alnum) :-
  97 =< CharCode,
  CharCode =< 122,
  !.
%  Upper-case letters without accent
char_typ(CharCode, alnum) :-
  65 =< CharCode,
  CharCode =< 90,
  !.
%  Accented characters.
%  The values 215 and 247 correspond to
%  the multiplication and division symbols:  Œ œ
char_typ(CharCode, alnum) :-
  192 =< CharCode,
  CharCode =< 255,
  CharCode =\= 215,
  CharCode =\= 247,
  !.
%  Digits
char_typ(CharCode, alnum) :-
  48 =< CharCode,
  CharCode =< 57,
  !.
%  The oe, OE, and Y" letters
char_typ(CharCode, alnum) :-
  (CharCode =:= 140 ; CharCode =:= 156 ;
    CharCode =:= 159),
  !.
```

Improving the Tokenizer. The program we have written may work badly for some strings. For instance, it does not process the point of decimal numbers such as 3.14, and inserts a sentence end between 3 and 14. This can be fixed using two-pass processing. The first pass recognizes decimal numbers with an appropriate grammar and annotates them. The second one runs the tokenizer on the resulting text.

We can generalize this strategy to improve the tokenizer with specific grammars recognizing numbers, dates, and percentages that will be run as as many different processing stages. However, there will remain cases where the program fails, notably with abbreviations, which often have fixed and variable parts as in *Dr. Watson*. *Dr.* is a common fixed title, and *Watson* a possible name. We can bring a second improvement to the tokenizer using rules and a lexicon of common abbreviations: *Mr.*, *Sen.*, *Rep.*, *Oct.*, *Fig.*, *pp.*, etc. Rules may recognize likely abbreviations by testing whether the period is followed either by a comma, a semicolon, a question mark or a lowercase letter. The tokenizer can test words ending with a period using the rules or the lexicon to decide whether they are a sentence end or not (Grefenstette and Tapanainen 1994). Mikheev (2002) describes a more efficient method that learns tokenization rules from the set of ambiguous tokens distributed in a document.

4.3.2 Tokenizing Texts in Perl

Perl offers a simple and very fast way to tokenize files into words using the `tr` operator. We will consider that contiguous sequences of characters, including the dash and the quote, are words, and we will isolate them on a single line. We will isolate the punctuation symbols on a single line as well.

The Perl program formulates this a little differently:

- If the character is not a letter or punctuation sign, then replace it by a new line. Note that the dash character in `tr` as well as in character classes means an interval and that we have to quote it to process it in a text.
- If it is a punctuation sign, then insert it between two new lines.
- Finally, reduce contiguous sequences of new lines to a single occurrence.

```
$text = <>;
while ($line = <>) {
  $text .= $line;
}
$text =~ tr/a-zåàâäæçéèêëîïôöœùûüßA-
ZÅÀÂÄÆÇÉÈÊËÎÏÔÖŒÙÛÜ'()\-,.?!:;/\n/cs;
$text =~ s/([,.?!:;()'\-])/\n$1\n/g;
$text =~ s/\n+/\n/g;
print $text;
```

4.4 N-grams

4.4.1 Some Definitions

The first step of lexical statistics consists in extracting the list of **word types** or **types**, i.e., the distinct words, from a corpus, along with their frequencies. Within the context of lexical statistics, word types are opposed to word tokens, the sequence of running words of the corpus. The excerpt from George Orwell's *Nineteen Eighty-Four*:

War is peace
Freedom is slavery
Ignorance is strength

has nine tokens and seven types. The type-to-token ratio is often used as an elementary measure of a text's density.

Collocations and language models also use the frequency of pairs of adjacent words: **bigrams**, for example how many *of the* there are in this text; of word triplets: **trigrams**; and more generally of fixed sequences of n words: n**-grams**. In lexical statistics, single words are called **unigrams**.

Jelinek (1985) exemplified corpus statistics and trigrams with the phrase

We need to resolve all the important issues

selected from a 90-million-word corpus of IBM office correspondences. Table 4.1 shows each word of this phrase, its rank in the corpus, and other words ranking before it according to a linear combination of trigram, bigram, and unigram probabilities. In this corpus, *We* is the ninth most probable word to begin a sentence. More likely words are *The*, *This*, etc. Following *We*, *need* is the seventh most probable word. More likely bigrams are *We are*, *We will*, *We the*, *We would*.... Knowing that words *We need* have been written, *to* is the most likely word to come after them. Similarly, *the* is the most probable word to follow *all of*.

Table 4.1. Trigram generation. After Jelinek (1985).

Word	Rank	More likely alternatives
We	9	*The This One Two A Three Please In*
need	7	*are will the would also do*
to	1	
resolve	85	*have know do...*
all	9	*the this these problems...*
of	2	*the*
the	1	
important	657	*document question first...*
issues	14	*thing point to...*

4.4.2 Counting Unigrams in Prolog

Counting unigrams in a corpus consists simply in tokenizing it, sorting the words, and counting the number of times a type occurs in the corpus. We will not use the Prolog predefined `sort/2` predicate because it removes the duplicates. Instead, we can use a predicate implementing the quicksort algorithm or `msort/2` in some Prologs.

The predicate `count_duplicates/2` counts the duplicates. It takes the ordered list of words as an input and returns a list of pairs with the frequency of each word `[N, Word]` in the output list:

```
count_duplicates(OrderedList, CountedList) :-
  count_duplicates(OrderedList, 1, [],
    CountedListRev),
  reverse(CountedListRev, CountedList).

count_duplicates([X, X | Ordered], N, Counting,
    Counted) :-
  N1 is N + 1,
  !,
  count_duplicates([X | Ordered], N1, Counting,
    Counted).
count_duplicates([X | Ordered], N, Counting,
    Counted) :-
  !,
  count_duplicates(Ordered, 1, [[N, X] | Counting],
    Counted).
count_duplicates([], _, L, L).
```

We get the unigrams with their counts with

```
?- read_file(myFile, CharacterList),
   tokens(TokenList, CharacterList, []),
   quicksort(TokenList, OrderedTokens),
   count_duplicates(OrderedTokens, UnigramList).
```

4.4.3 Counting Unigrams with Perl

Counting unigrams is straightforward and very fast with Perl. We can obtain them with the following algorithm:

1. Tokenize the text file, putting one word per line with `tr`.
2. Count the words using a hash table.
3. Possibly, sort the words according to their alphabetical order and numerical ranking.

The tokenizing part is the same as in the previous section, and we use the `split` function to assign each word of the text to the elements of an array. As we saw in Chap. 2, `split` takes two arguments: a regular expression, which describes a delimiter, and a string, which is split everywhere the delimiter matches. The resulting fragments are assigned sequentially to an array. Let `$text` be a big string containing the whole text with one word per line. The instruction:

```
@words = split(/\n/, $text);
```

assigns the first line and hence the first word to `$words[0]`, the second word to `$words[1]`, and so on. A useful generalization of this instruction is

```
@words = split(/\s+/, $text);
```

which splits the text at each sequence of white space characters.

Then, we use a hash table or associative array. Instead of being indexed by consecutive numbers, as in classical arrays, hash tables are indexed by strings. The next three lines

```
$wordcount{"a"} = 21;
$wordcount{"And"} = 10;
$wordcount{"the"} = 18;
```

create a hash table `$wordcount` with three indices called the keys: `a`, `And`, `the`, whose values are 21, 10, and 18. Hash keys can be numbers as well as strings. We refer to the whole array using the notation `%wordcount`. The instruction `keys` return the keys of the array as in

```
keys %wordcount
```

A hash entry is created when a value is assigned to it. Its existence can be tested using the `exists` Boolean function.

The counting program scans the `@words` array and increments the frequency of the words as they occur. We finally introduce two new instructions and functions. The instruction `foreach item (list)` iterates over the items of an array, and `sort(array)` returns a sorted array.

```
$text = <>;
while ($line = <>) {
  $text .= $line;
}
$text =~ tr/a-zåàâäæçéèêëîïôöœùûüßA-
 ZÅÀÂÄÆÇÉÈÊËÎÏÔÖŒÙÛÜ’()\-,.?!:;/\n/cs;
$text =~ s/([,.?!:;()’\-])/\n$1\n/g;
$text =~ s/\n+/\n/g;
@words = split(/\n/, $text);
for ($i = 0; $i <= $#words; $i++) {
  if (!exists($frequency{$words[$i]})) {
    $frequency{$words[$i]} = 1;
  } else {
    $frequency{$words[$i]}++;
  }
}
foreach $word (sort keys %frequency){
  print "$frequency{$word} $word\n";
}
```

4.4.4 Counting Bigrams with Perl

We count bigrams and n-grams just as we did with unigrams. The only difference is that we create an array of bigrams by concatenating the adjacent words. The following Perl program enables us to obtain them:

```
$text = <>;
while ($line = <>) {
  $text .= $line;
}
$text =~ tr/a-zåàâäæçéèêëîïôöœùûüßA-
ZÅÀÂÄÆÇÉÈÊËÎÏÔÖŒÙÛÜ'()\-,.?!:;/\n/cs;
$text =~ s/([,.?!:;()'\-])/\n$1\n/g;
$text =~ s/\n+/\n/g;
@words = split(/\n/, $text);
for ($i = 0; $i < $#words; $i++) {
  $bigrams[$i] = $words[$i] . " " . $words[$i + 1];
}
for ($i = 0; $i < $#words; $i++) {
  if (!exists($frequency_bigrams{$bigrams[$i]})) {
    $frequency_bigrams{$bigrams[$i]} = 1;
  } else {
    $frequency_bigrams{$bigrams[$i]}++;
  }
}
foreach $bigram (sort keys %frequency_bigrams){
  print "$frequency_bigrams{$bigram} $bigram \n";
}
```

4.5 Probabilistic Models of a Word Sequence

4.5.1 The Maximum Likelihood Estimation

We observed in Table 4.1 that some word sequences are more likely than others. Using a statistical model, we can quantify these observations. The model will enable us to assign a probability to a word sequence as well as to predict the next word to follow the sequence.

Let $S = w_1, w_2, ..., w_i, ..., w_n$ be a word sequence. Given a training corpus, an intuitive estimate of the probability of the sequence, $P(S)$, is the relative frequency of the string $w_1, w_2, ..., w_i, ..., w_n$ in the corpus. This estimate is called the *maximum likelihood estimate* (MLE):

$$P_{MLE}(S) = \frac{C(w_1, ..., w_n)}{N},$$

where $C(w_1, ..., w_n)$ is the frequency or count of the string $w_1, w_2, ..., w_i, ..., w_n$ in the corpus, and N is the total number of strings of length n.

Most of the time, however, it is impossible to obtain this estimate. Even when corpora reach billions of words, they have a limited size, and it is unlikely that we can always find the exact sequence we are searching. We can try to simplify the computation and decompose $P(S)$ a step further as:

$$\begin{aligned} P(S) &= P(w_1, ..., w_n), \\ &= P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)P(w_n|w_1, ..., w_{n-1}), \\ &= \prod_{i=1}^{n} P(w_i|w_1, ..., w_{i-1}). \end{aligned}$$

The probability $P(It\ was\ a\ bright\ cold\ day\ in\ April)$ from *Nineteen Eighty-Four* corresponds then to the probability of having *It* to begin the sentence, then *was* knowing that we have *It* before, then *a* knowing that we have *It was* before, and so on until the end of the sentence. It yields the product of conditional probabilities:

$$\begin{aligned} P(S) = &P(It) \times P(was|It) \times P(a|It, was) \times P(bright|It, was, a) \times ... \\ &\times P(April|It, was, a, bright, ..., in). \end{aligned}$$

To estimate $P(S)$, we need to know unigram, bigram, trigram, so far, so good, but also 4-gram, 5-gram, and even 8-gram statistics. Of course, no corpus is big enough to produce them. A practical solution is then to limit the n-gram length to 2 or 3, and thus to approximate them to bigrams:

$$P(w_i|w_1, w_2, ..., w_{i-1}) \approx P(w_i|w_{i-1}),$$

or trigrams:

$$P(w_i|w_1, w_2, ..., w_{i-1}) \approx P(w_i|w_{i-2}, w_{i-1}).$$

Using a trigram language model, $P(S)$ is approximated as:

$$\begin{aligned} P(S) \approx &P(It) \times P(was|It) \times P(a|It, was) \times P(bright|was, a) \times ... \\ &\times P(April|day, in). \end{aligned}$$

Using a bigram grammar, the general case of a sentence probability is:

$$P(S) \approx P(w_1) \prod_{i=2}^{n} P(w_i|w_{i-1}),$$

with the estimate

$$P_{MLE}(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i)}{\sum_{w} C(w_{i-1}, w)} = \frac{C(w_{i-1}, w_i)}{C(w_{i-1})}.$$

Similarly, the trigram maximum likelihood estimate is:

$$P_{MLE}(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i)}{C(w_{i-2}, w_{i-1})}.$$

And the general case of n-gram estimation is:

$$\begin{aligned} P_{MLE}(w_{i+n}|w_{i+1}, ..., w_{i+n-1}) &= \frac{C(w_{i+1}, ..., w_{i+n})}{\sum_{w} C(w_{i+1}, ..., w_{i+n-1}, w)}, \\ &= \frac{C(w_{i+1}, ..., w_{i+n})}{C(w_{i+1}, ..., w_{i+n-1})}. \end{aligned}$$

4.5.2 Using ML Estimates with *Nineteen Eighty-Four*

Training and Testing the Language Model. Before computing the probability of a word sequence, we must train the language model. The corpus used to derive the n-gram frequencies is classically called the **training set**, and the corpus on which we apply the model, the **test set**. Both sets should be distinct. If we apply a language model to a word sequence, which is part of the training corpus, its probability will be biased to a higher value, and thus will be inaccurate. The training and test sets can be balanced or not, depending on whether we want them to be specific of a task or more general.

For some models, we need to optimize parameters in order to obtain the best results. Again, it would bias the results if at the same time, we carry out the optimization on the test set and run the evaluation on it. For this reason some models need a separate **development set** to fine-tune their parameters.

In some cases, especially with small corpora, a specific division between training and test sets may have a strong influence on the results. It is then preferable to apply the training and testing procedure several times with different sets and average the results. The method is to divide randomly the corpus into two sets. We learn the parameters from the training set, apply the model to the test set, and repeat the process with a new random division, for instance, ten times. This method is called **cross-validation**, or 10-fold cross-validation if we repeat it 10 times. Cross-validation smoothes the impact of a specific partition of the corpus.

Marking up the Corpus. Most corpora use some sort of markup language. The most common markers of N-gram models are the sentence delimiters `<s>` to mark the start of a sentence and `</s>` at its end. For example:

`<s>` *It was a bright cold day in April* `</s>`

Depending on the application, both symbols can be counted in the n-gram frequencies just as the other tokens or can be considered as context cues. Context cues are vocabulary items that appear in the condition part of the probability but are never predicted – they never occur in the right part. In many models, `<s>` is a context cue and `</s>` is part of the vocabulary. We will adopt this convention in the next examples.

The Vocabulary. We have defined language models that use a finite and predetermined set of words. This is never the case in reality, and the models will have to handle out-of-vocabulary (OOV) words. Training corpora are typically of millions, or even billions, of words. However, whatever the size of a corpus, it will never have a complete coverage of the vocabulary. Some words that are unseen in the training corpus are likely to occur in the test set. In addition, frequencies of rare words will not be reliable.

There are two main types of methods to deal with OOV words:

- The first method assumes a **closed vocabulary**. All the words both in the training and the test sets are known in advance. Depending on the language model settings, any word outside the vocabulary will be discarded or cause an error. This method is used in some applications, like voice control of devices.
- The **open vocabulary** makes provisions for new words to occur with a specific symbol, `<UKN>`, called the unknown token. All the OOV words are mapped to `<UNK>`, both in the training and test sets.

The vocabulary itself can come from an external dictionary. It can also be extracted directly from the training set. In this case, it is common to exclude the rare words, notably those seen only once – the *hapax legomena*. The vocabulary will then consist of the most frequent types of the corpus, for example, the 20,000 most frequent types. The other words, unseen or with a frequency lower than a cutoff value, 1, 2, or up to 5, will be mapped to `<UKN>`.

Computing a Sentence Probability. We trained a bigram language model on a very small corpus consisting of the three chapters of *Nineteen Eighty-Four*. We kept the appendix, "The Principles of Newspeak," as the test set and we selected this sentence from it:

> `<s>` *A good deal of the literature of the past was, indeed, already being transformed in this way* `</s>`

We first normalized the text: We created a file with one sentence per line. We inserted automatically the delimiters `<s>` and `</s>`. We removed the punctuation, parentheses, quotes, stars, dashes, tabulations, and double white spaces. We set all the words in lowercase letters. We counted the words, and we produced a file with the unigram and bigram counts.

The training corpus has 115,212 words; 8,635 types, including 3,928 hapax legomena; and 49,524 bigrams, where 37,365 bigrams have a frequency of 1. Table 4.2 shows the unigram and bigram frequencies for the words of the test sentence. It excludes `<s>` from the unigram probabilities.

Table 4.2. Frequencies of unigrams and bigrams.

w_i	$C(w_i)$	$P_{MLE}(w_i)$	w_{i-1}, w_i	$C(w_{i-1}, w_i)$	$P_{MLE}(w_i \vert w_{i-1})$
`<s>`	7072	–	–	–	–
A	2482	0.023	`<s>` *a*	133	0.019
good	53	0.00049	*a good*	14	0.006
deal	5	$4.62\ 10^{-5}$	*good deal*	0	0.0
of	3310	0.031	*deal of*	1	0.2
the	6248	0.058	*of the*	742	0.224
literature	7	$6.47\ 10^{-5}$	*the literature*	1	0.0002
of	3310	0.031	*literature of*	3	0.429
the	6248	0.058	*of the*	742	0.224
past	99	0.00092	*the past*	70	0.011
was	2211	0.020	*past was*	4	0.040
indeed	17	0.00016	*was indeed*	0	0.0
already	64	0.00059	*indeed already*	0	0.0
being	80	0.00074	*already being*	0	0.0
transformed	1	$9.25\ 10^{-6}$	*being transformed*	0	0.0
in	1759	0.016	*transformed in*	0	0.0
this	264	0.0024	*in this*	14	0.008
way	122	0.0011	*this way*	3	0.011
`</s>`	7072	0.065	*way* `</s>`	18	0.148

All the words of the sentence have been seen in the training corpus, and we can compute a probability estimate of it using the unigram relative frequencies:

$$P(S) \approx P(a) \times P(good) \times ... \times P(way) \times P(< /s >),$$
$$\approx 1.18 \times 10^{-48}.$$

The bigrams estimate is defined as

$$P(S) \approx P(a| < s >) \times P(good|a) \times ... \times P(way|this) \times P(< /s > |way).$$

and has a zero probability, which is due to **sparse data**: the fact that the corpus is not big enough to have all the bigrams covered with a realistic estimate. We shall see in the next section how to handle them.

4.6 Smoothing N-gram Probabilities

4.6.1 Sparse Data

The approach using the maximum likelihood estimation has an obvious disadvantage because of the unavoidably limited size of the training corpora. Given a vocabulary of 20,000 types, the potential number of bigrams is $20,000^2 = 400,000,000$, and with trigrams, it amounts to the astronomic figure of $20,000^3 = 8,000,000,000,000$. No corpus yet has the size to cover the corresponding word combinations.

Among the set of potential n-grams, some are almost impossible, except as random sequences generated by machines; others are simply unseen in the corpus. This phenomenon is referred to as **sparse data**, and the maximum likelihood estimator gives no hint on how to estimate their probability.

In this section, we introduce **smoothing** techniques to estimate probabilities of unseen n-grams. As the sum of probabilities of all the n-grams of a given length is 1, smoothing techniques also have to re-arrange the probabilities of the observed n-grams. Smoothing allocates a part of the probability mass to the unseen n-grams that, as a counterpart, it shifts – or **discounts** – from the other n-grams.

4.6.2 Laplace's Rule

Laplace's rule (Laplace 1820, p. 17) is probably the oldest published method to cope with sparse data. It just consists in adding one to all the counts. For this reason, some authors also call it the add-one method. The frequency of unseen n-grams is equal to 1 and the general estimate of a bigram probability is:

$$P_{Laplace}(w_{i+1}|w_i) = \frac{C(w_i, w_{i+1}) + 1}{\sum\limits_{w} (C(w_i, w) + 1)} = \frac{C(w_i, w_{i+1}) + 1}{C(w_i) + Card(V)},$$

where $Card(V)$ is the number of word types. The denominator correction is necessary to have the probability sum equal to 1.

Table 4.3. Frequencies of bigrams using Laplace's rule.

w_i, w_{i+1}	$C(w_i, w_{i+1})$	$C(w_i) + Card(V)$	$P_{Lap}(w_{i+1}\|w_i)$
`<s>` *a*	133	7072 + 8634	0.0085
a good	14	2482 + 8634	0.0013
good deal	0	53 + 8634	0.00012
deal of	1	5 + 8634	0.00023
of the	742	3310 + 8634	0.062
the literature	1	6248 + 8634	0.00013
literature of	3	7 + 8634	0.00046
of the	742	3310 + 8634	0.062
the past	70	6248 + 8634	0.0048
past was	4	99 + 8634	0.00057
was indeed	0	2211 + 8634	0.000092
indeed already	0	17 + 8634	0.00012
already being	0	64 + 8634	0.00011
being transformed	0	80 + 8634	0.00011
transformed in	0	1 + 8634	0.00012
in this	14	1759 + 8634	0.0014
this way	3	264 + 8634	0.00045
way `</s>`	18	122 + 8634	0.0022

With Laplace's rule, we can use bigrams to compute the sentence probability (Table 4.3):

$$\begin{aligned} P(S) &\approx P(a| < s >) \times P(good|a) \times ... \times P(< /s > |way), \\ &\approx 4.62 \times 10^{-57}. \end{aligned}$$

Laplace's method is easy to understand and implement. It has an obvious drawback however: it shifts an enormous mass of probabilities to the unseen n-grams and gives them a considerable importance. The frequency of the unlikely bigram *the of* will be 1, a quarter of the much more common *this way*.

The **discount** value is the ratio between the smoothed frequencies and their actual counts in the corpus. The bigram *this way* has been discounted by $\frac{0.011}{0.00045} = 24.4$ to make place for the unseen bigrams. This is unrealistic and shows the major drawback of this method.

If adding 1 is too much, why not try less, for instance, 0.5. This is the idea of the Lidstone's rule. This value is denoted λ. The new formula is then:

$$P_{Lidstone}(w_{i+1}|w_i) = \frac{C(w_i, w_{i+1}) + \lambda}{C(w_i) + \lambda Card(V)},$$

which, however, is not a big improvement.

4.6.3 Good–Turing Estimation

The Good–Turing estimation (Good 1953) is one of the most efficient smoothing methods. As with Laplace's rule, it reestimates the counts of the n-grams observed in the corpus by discounting them, and it shifts the probability mass it has shaved to the unseen bigrams. The discount factor is variable, however, and depends on the number of times a n-gram has occurred in the corpus. There will be a specific discount value to n-grams seen once, another one to bigrams seen twice, a third one to those seen three times, and so on.

Let us denote N_c the number of n-grams that occurred exactly c times in the corpus. N_0 is the number of unseen n-grams, N_1 the number of n-grams seen once, N_2 the number of n-grams seen twice, and so on. If we consider bigrams, the value N_0 is $Card(V)^2$ minus all the bigrams we have seen.

The Good–Turing method reestimates the frequency of n-grams occurring c times using the formula:

$$c* = (c+1)\frac{E(N_{c+1})}{E(N_c)},$$

where $E(x)$ denotes the expectation of the random variable x. This formula is usually approximated as

$$c* = (c+1)\frac{N_{c+1}}{N_c}.$$

Hence, the Good–Turing estimation of the unseen n-grams is $c* = \frac{N_1}{N_0}$, and the n-grams that have been seen once in the training corpus are reestimated to $c* = \frac{2N_2}{N_1}$.

The three chapters in *Nineteen Eighty-Four* contain 37,365 unique bigrams and 5820 bigrams seen twice. Its vocabulary of 8634 words generates $8634^2 =$ 74,545,956 bigrams, of which 74,513,701 are unseen. The Good–Turing method reestimates the frequency of each unseen bigram to $37,365/74,513,701 = 0.0005$ and unique bigrams to $2 \times (5820/37,365) = 0.31$. Table 4.4 shows the complete the reestimated frequencies for the n-grams up to 9.

In practice, only high values of N_c are reliable, which correspond to low values of c. In addition, above a certain threshold, most frequencies of frequency will be equal to zero. Therefore, the Good–Turing estimation is applied for $c < k$, where k is a constant set to 5, 6, ..., or 10. Other counts are not reestimated.

The probability of a n-gram is given by the formula

$$P_{GT}(w_1, ..., w_n) = \frac{c*(w_1, ..., w_n)}{N},$$

where $c*$ is the reestimated count of $w_1...w_n$, and N the original count of n-grams in the corpus. The conditional frequency is

$$P_{GT}(w_n|w_1, ..., w_{n-1}) = \frac{c*(w_1, ..., w_n)}{C(w_1, ..., w_{n-1})}.$$

Table 4.5 shows the conditional frequencies where only frequencies less than 10 have been reestimated.

Table 4.4. The reestimated frequencies of the bigrams.

Frequency of occurrence	N_c	$c*$
0	74,513,701	0.0005
1	37,365	0.31
2	5,820	1.09
3	2,111	2.02
4	1,067	3.37
5	719	3.91
6	468	4.94
7	330	6.06
8	250	6.44
9	179	8.93

4.7 Using N-grams of Variable Length

In the previous section, we used smoothing techniques to reestimate the probability of n-grams of constant length, whether they occurred in the training corpus or not. A property of these techniques is that they assign a same probability to all the unseen n-grams.

Table 4.5. The conditional frequencies using the Good–Turing method.

w_i, w_{i+1}	$C(w_i, w_{i+1})$	$c*(w_i, w_{i+1})$	$P_{GT}(w_{i+1}\|w_i)$
`<s>` *a*	133	133	0.019
a good	14	14	0.006
good deal	0	0.0005	$9.46\ 10^{-6}$
deal of	1	0.31	0.062
of the	742	742	0.224
the literature	1	0.31	$4.99\ 10^{-5}$
literature of	3	2.02	0.29
of the	742	742	0.224
the past	70	70	0.011
past was	4	3.37	0.034
was indeed	0	0.0005	$2.27\ 10^{-7}$
indeed already	0	0.0005	$2.95\ 10^{-5}$
already being	0	0.0005	$7.84\ 10^{-6}$
being transformed	0	0.0005	$6.27\ 10^{-6}$
transformed in	0	0.0005	0.00050
in this	14	14	0.008
this way	3	2.02	0.0077
way `</s>`	18	18	0.148

Another strategy is to rely on the frequency of observed sequences but of lesser length: $n-1$, $n-2$, and so on. As opposed to smoothing, the estimate of each unseen n-gram will be specific to the words it contains. In this section, we introduce two techniques: the linear interpolation and the Katz back-off model.

4.7.1 Linear Interpolation

Linear interpolation, also called deleted interpolation (Jelinek and Mercer 1980), combines linearly the maximum likelihood estimates from length 1 to n. For trigrams, it corresponds to:

$$\begin{aligned} P_{DelInterpolation}(w_n|w_{n-2}, w_{n-1}) = {} & \lambda_1 P_{MLE}(w_n|w_{n-2}w_{n-1}) + \\ & \lambda_2 P_{MLE}(w_n|w_{n-1}) + \lambda_3 P_{MLE}(w_n), \end{aligned}$$

where $0 \leq \lambda_i \leq 1$ and $\sum_{i=1}^{3} \lambda_i = 1$.

The values can be constant and set by hand, for instance, $\lambda_1 = 0.6$, $\lambda_2 = 0.3$, and $\lambda_3 = 0.1$. They can also be trained and optimized from a corpus (Jelinek 1997).

We can now understand why bigram *we the* is ranked so high in Table 4.1 after *we are* and *we will*. Although, it can occur in English, as in the American constitution, *We, the people*..., it is not a very frequent combination. In fact, the estimation has been obtained with an interpolation where the term $\lambda_3 P_{MLE}(the)$ boosted the bigram to the top because of the high frequency of *the*.

4.7.2 Back-off

The idea of the back-off model (Katz 1987) is to use the frequency of longest available n-grams, and if no n-gram is available to back off to the $(n-1)$-gram, and then to $(n-2)$-gram, and so on. If n is 3, we first try trigrams, then bigrams, and finally unigrams. This can be expressed as:

$$P_{Backoff}(w_i|w_{i-2}, w_{i-1}) = \begin{cases} \tilde{P}(w_i|w_{i-2}, w_{i-1}), & \text{if } C(w_{i-2}, w_{i-1}, w_i) \neq 0, \\ \alpha_1 P(w_i|w_{i-1}), & \text{if } C(w_{i-2}, w_{i-1}, w_i) = 0 \\ & \text{and } C(w_{i-1}, w_i) \neq 0, \\ \alpha_2 P(w_i), & \text{otherwise.} \end{cases}$$

So far, this model does not tell us how to estimate the n-gram probabilities to the right of the formula. A first idea would be to use the maximum likelihood estimate,

$$P_{MLE}(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i)}{C(w_{i-2}, w_{i-1})},$$

but in this case, the sum of all the probabilities would be more than 1. Therefore we need the α_1 and α_2 values to ensure that the sum of probabilities is equal to 1. In addition, to make room for them, we need to discount the trigram estimates.

The back-off model is often used in conjunction with the Good–Turing discounting and the estimation is solved recursively. Let us first assume that for all the possible trigrams, we can back off to an observed bigram. We use the Good–Turing estimate,

$$\tilde{P}(w_i|w_{i-2}, w_{i-1}) = \frac{C^*(w_{i-2}, w_{i-1}, w_i)}{C(w_{i-2}, w_{i-1})},$$

to discount the observed trigrams and we assign the remaining probability mass to the unseen trigrams. That is,

$$1 - \sum_{w_i, C(w_{i-2}, w_{i-1}, w_i) > 0} \tilde{P}(w_i|w_{i-2}, w_{i-1}).$$

We compute α_1 so that the probability mass of the observed bigrams fits this value.

But we are not finished yet. In case of an unseen bigram, the model uses a unigram estimate. To compute it, the Katz model recursively applies the method it used with the trigrams. It further discounts the observed bigrams with the Good–Turing estimation to make room for the unigrams. It then adjusts the α_2 value so that the sum of probabilities of the discounted observed bigrams and the weighted unigram probabilities is equal to 1.

4.8 Quality of a Language Model

4.8.1 Intuitive Presentation

We can compute the probability of sequences of any length or of whole texts. As each word in the sequence corresponds to a conditional probability less than 1, the product will naturally decrease with the length of the sequence. To make sense, we normally average it by the number of words in the sequence and extract its nth root. This measure, which is a sort of a per-word probability of a sequence L, is easier to compute using a logarithm:

$$H(L) = -\frac{1}{n} \log_2 P(w_1, ..., w_n).$$

We have seen that trigrams are better predictors than bigrams, which are better than unigrams. This means that the probability of a very long sequence computed with a bigram model will normally be higher than with a unigram one. The log measure will then be lower.

Intuitively, this means that the $H(L)$ measure will be a quality marker for a language model where lower numbers will correspond to better models. This intuition has mathematical foundations, as we will see in the two next sections.

4.8.2 Entropy Rate

We used entropy with characters in Chap. 3. We can use it with any symbols such as words, bigrams, trigrams, or any n-grams. When we normalize it by the length of the word sequence, we define the **entropy rate**:

$$H_{rate} = -\frac{1}{n} \sum_{w_1,...,w_n \in L} p(w_1, ..., w_n) \log_2 p(w_1, ..., w_n),$$

where L is the set of all possible sequences of length n.

It has been proven that when $n \to \infty$ or n is very large and under certain conditions, we have

$$\begin{aligned} H_{rate}(L) &= \lim_{n\to\infty} -\tfrac{1}{n} \sum_{w_1,...,w_n \in L} p(w_1, ..., w_n) \log_2 p(w_1, ..., w_n), \\ &= \lim_{n\to\infty} -\tfrac{1}{n} \log_2 p(w_1, ..., w_n), \end{aligned}$$

which means that we can compute $H_{rate}(L)$ from a very long sequence, ideally infinite, instead of summing of all the sequences of a definite length.

4.8.3 Cross Entropy

We can also use cross entropy, which is measured between a text, called the language and governed by an unknown probability p, and a language model m. Using the same definitions as in Chap. 3, the cross entropy of m on p is given by:

$$H(p, m) = -\frac{1}{n} \sum_{w_1,...,w_n \in L} p(w_1, ..., w_n) \log_2 m(w_1, ..., w_n).$$

As for the entropy rate, it has been proven that, under certain conditions

$$\begin{aligned} H(p,m) &= \lim_{n\to\infty} -\tfrac{1}{n} \sum_{w_1,...,w_n \in L} p(w_1,...,w_n) \log_2 m(w_1,...,w_n), \\ &= \lim_{n\to\infty} -\tfrac{1}{n} \log_2 m(w_1,...,w_n). \end{aligned}$$

In applications, we generally compute cross entropy on the complete word sequence of a test set, governed by p, using a bigram or trigram model, m, derived from a training set.

In Chap. 3, we saw the inequality $H(p) \leq H(p,m)$. This means that the cross entropy will always be an upper bound of $H(p)$. As the objective of a language model is to be as close as possible to p, the best model will be the one yielding the lowest possible value. This forms the mathematical background of the intuitive presentation in Sect. 4.8.1.

4.8.4 Perplexity

The perplexity of a language model is defined as:

$$PP(p,m) = 2^{H(p,m)}.$$

Perplexity is interpreted as the average "branching factor" of a word: the statistically weighted number of words that follow a given word. Perplexity is equivalent to entropy. The only advantage of perplexity is that it results in numbers more comprehensible for human beings. It is then more popular to measure the quality of language models. As is the case for entropy, the objective is to minimize it: the better the language model, the lower the perplexity.

4.9 Collocations

Collocations are recurrent combinations of words. They are ubiquitous and arbitrary in English, French, German, and other languages (Smadja 1993). Simplest collocations are fixed n-grams such as *The White House*, and *Le Président de la République*. Other collocations involve some morphological or syntactic variation such as the one linking *make* and *decision* in American English: *to make a decision*, *decisions to be made*, *made an important decision*. Smadja (1993) calls the latter collocations predicative relations.

Collocations underlie word preferences that most of the time cannot easily be explained by a syntactic or semantic reasoning: they are merely resorting to usage. Collocations are in the mind of a native speaker. S/he can recognize them as valid. On the contrary, nonnative speakers may make mistakes when they are not aware of them or try to produce word-for-word translations. For this reason, many second language learners' dictionaries describe most frequent associations. In English, the *Oxford Advanced Learner's Dictionary*, *The Longman Dictionary of Contemporary English*, *The Cambridge International Dictionary*, and *The Collins COBUILD* carefully list verbs and prepositions or particles commonly associated such as phrasal verbs *set up*, *set off*, and *set out*.

Lexicographers used to identify collocations by introspection and by observing corpora, at the risk of forgetting some of them. Statistical tests can automatically extract associated words or "sticky" pairs from raw corpora. We introduce three of these tests in this section together with programs in Perl to compute them.

4.9.1 Word Preference Measurements

Mutual information (Church and Hanks 1990), t-score (Church and Mercer 1993), and the likelihood ratio (Dunning 1993) are statistical tests that are widely used to measure the strength of word associations:

- Mutual information is defined as

$$I(w_i, w_j) = \log_2 \frac{P(w_i, w_j)}{P(w_i)P(w_j)} \approx \log_2 \frac{NC(w_i, w_j)}{C(w_i)C(w_j)}.$$

- t-scores are defined as

$$\begin{aligned} t(w_i, w_j) &= \frac{mean(P(w_i,w_j)) - mean(P(w_i))mean(P(w_j))}{\sqrt{\sigma^2(P(w_i,w_j)) + \sigma^2(P(w_i))\sigma^2(P(w_j))}}, \\ &\approx \frac{C(w_i,w_j) - \frac{1}{N}C(w_i)C(w_j)}{\sqrt{C(w_i,w_j)}}. \end{aligned}$$

where $C(w_i)$ and $C(w_j)$ are respectively the frequencies of word w_i and word w_j in the corpus, $C(w_i, w_j)$ is the frequency of bigram w_i, w_j, and N is the total number of words in the corpus. The bigram count can be extended to the frequency of word w_i when it is followed or preceded by w_j in a window of k words. The latter definition is a generalization of the former with $k = 1$ and $j = i + 1$.

High t-scores show recurrent combinations of grammatical or very frequent words such as *of the*, *and the*, etc. Table 4.6 shows collocates of *set* extracted from the Bank of English using the t-score test. High mutual information shows pairs of words occurring together but generally with a lower frequency, such as technical terms. Table 4.7 gives collocates of the word *surgery*.

Table 4.6. Collocates of set extracted from Bank of English using the t-score test.

Word	Frequency	Bigram *set* + word	t-score
up	134,882	5512	67.980
a	1,228,514	7296	35.839
to	1,375,856	7688	33.592
off	52,036	888	23.780
out	12,3831	1252	23.320

Dunning (1993) criticized the t-score test and proposed an alternative based on likelihood ratios:

Table 4.7. Collocates of *surgery* extracted from the Bank of English using the mutual information test. Note the misspelled word *pioneeing*.

Word	Frequency	Bigram word + *surgery*	Mutual info
arthroscopic	3	3	11.822
pioneeing	3	3	11.822
reconstructive	14	11	11.474
refractive	6	4	11.237
rhinoplasty	5	3	11.085

$$LR(w_1, w_2) = 2 \log \frac{L(p_1, k_1, n_1) L(p_2, k_2, n_2)}{L(p, k_1, n_1) L(p, k_2, n_2)},$$

where
$p = \frac{C(w_2)}{N}$, $p_1 = \frac{C(w_1,w_2)}{C(w_1)}$, $p_2 = \frac{C(w_2)-C(w_1,w_2)}{N-C(w_1)}$, and $L(p, n, k) = p^k(1 - p)^{n-k}$.

4.9.2 Extracting Collocations with Perl

Both programs use unigram and bigram statistics. To compute them, we must first tokenize the text, and count words and bigrams using the tools we have described before:

```
$text = <>;
while ($line = <>) {
  $text .= $line;
}
$text =~ tr/a-zåàâäæçéèêëîïôöœùûüßA-
ZÅÀÂÄÆÇÉÈÊËÎÏÔÖŒÙÛÜ’()\-,.?!:;/\n/cs;
$text =~ s/([,.?!:;()’\-])/\n$1\n/g;
$text =~ s/\n+/\n/g;
@words = split(/\n/, $text);
for ($i = 0; $i < $#words; $i++) {
  $bigrams[$i] = $words[$i] . " " . $words[$i + 1];
}
for ($i = 0; $i <= $#words; $i++) {
  $frequency{$words[$i]}++;
}
for ($i = 0; $i < $#words; $i++) {
  $frequency_bigrams{$bigrams[$i]}++;
}
```

Finally, we must know the number of words in the corpus. This corresponds to the size of the `word` array: `$#word`.

Mutual Information. The Perl program iterates over the `word` array and applies the mutual information formula. The program is not optimal and computes the same value several times:

```
for ($i = 0; $i < $#words; $i++) {
  $mutual_info{$bigrams[$i]} = log(($#words + 1) *
    $frequency_bigrams{$bigrams[$i]}/
      ($frequency{$words[$i]} *
        $frequency{$words[$i + 1]}))/log(2);
}

foreach $bigram (keys %mutual_info){
  @bigram_array = split(/ /, $bigram);
  print $mutual_info{$bigram}, " ", $bigram, "\t",
  $frequency_bigrams{$bigram}, "\t",
  $frequency{$bigram_array[0]}, "\t",
  $frequency{$bigram_array[1]}, "\n";
}
```

***t*-Scores.** The program is similar to the previous one except the formula:

```
for ($i = 0; $i < $#words; $i++) {
  $t_scores{$bigrams[$i]} =
    ($frequency_bigrams{$bigrams[$i]} -
      $frequency{$words[$i]} *
        $frequency{$words[$i + 1]}/($#words + 1))
         /sqrt($frequency_bigrams{$bigrams[$i]});
}

foreach $bigram (keys %t_scores ){
  @bigram_array = split(/ /, $bigram);
  print $t_scores{$bigram}, " ", $bigram, "\t",
  $frequency_bigrams{$bigram}, "\t",
  $frequency{$bigram_array[0]}, "\t",
  $frequency{$bigram_array[1]}, "\n";
}
```

4.10 Application: Retrieval and Ranking of Documents on the Web

The advent of the Web in the mid-1990s made it possible to retrieve automatically quantities of electronic documents at a modest cost. Companies providing such a service are among the most popular sites on the Internet. The most notable ones include Google, Yahoo, and MSN Search.

Web search systems or engines are based on "spiders" or "crawlers" that visit Internet addresses, follow links they encounter, and collect all the pages they traverse. Crawlers can amass billions of pages every month. They necessitate massive network bandwidth, storage capacity, and computing power.

All the pages the crawlers download are tokenized and undergo a full text indexing. The engine lists all the words of its collection of documents and links each word with the pages where this word occurs in. This is pretty much like a book index except that it considers all the words. When a user asks for a specific word, the search system answers with the pages that contain it.

Search engines represent documents internally using statistical or popularity models. A popular representation is the vector space model (Salton 1988). The idea is to represent the documents in a vector space whose directions are the words. Then documents are vectors in a space of words. Let us first suppose that the document coordinates are the occurrence count of each word. A document would be represented as: $\mathbf{d} = (C(w_1), C(w_2), C(w_3), ..., C(w_n))$. Table 4.8 shows the matrix representing a collection of documents where each cell (D_i, w_j) contains the frequency of w_j in document D_i.

Table 4.8. The word by document matrix. Each cell (D_i, w_j) contains the frequency of w_j in document D_i.

Words\Documents	D_1	D_2	D_3	**...**	D_m
W_1					
W_2					
...					
W_n					

Using the vector space model, we can measure the similarity of two documents by the angle they form in the vector space. It is easier to computer the cosine of the angle:

$$\cos(\mathbf{q}, \mathbf{d}) = \frac{\sum\limits_{i=1}^{n} q_i d_i}{\sqrt{\sum\limits_{i=1}^{n} q_i^2}\sqrt{\sum\limits_{i=1}^{n} d_i^2}}.$$

In fact, the rough word count is replaced by a more elaborate term: the term frequency times the inverted document frequency, better known as $tf \times idf$ (Salton 1988). To examine how it works, let us take the phrase *Internet in Somalia* as an example.

A document that contains many *Internet* words is probably more relevant than a document that has only one. The frequency $tf_{i,j}$ of a term j in a document i reflects this. It is a kind of a "mass" relevance. However, since *Internet* is a very common word, it is not specific. The number of documents that contain it must downplay its importance. This is the role of $idf_j = \log(\frac{N}{n_j})$, where N is the total number of documents in the collection – the total number of pages the crawler has collected – divided by the number of pages n_j, where a term j occurs at least once. *Somalia* probably appears in fewer documents than *Internet* and idf_j will give it a chance. The weight of a term j in document i is finally defined as $tf_{i,j} \times \log(\frac{N}{n_j})$.

The user may query a search engine with a couple of words or a phrase. Most systems will then answer with the pages that contain all the words and any of the words of the question. Some questions return hundreds or even thousands of valid documents. Ranking a document consists in projecting the space to that of the question words using the cosine. With this model, higher cosines will indicate better relevance. In addition to $tf \times idf$, search systems may employ heuristics such as giving more weight to the words in the title of a page (Mauldin and Leavitt 1994).

Google's PageRank algorithm (Brin and Page 1998) uses a different technique that takes into account the page popularity. PageRank considers the "backlinks", the links pointing to a page. The idea is that a page with many backlinks is likely to be a page of interest. Each backlink has a specific weight, which corresponds to the rank of the page it comes from. The page rank is simply defined as the sum of the ranks of all its backlinks. The importance of a page is spread through its forward links and contributes to the popularity of the pages it points to. The weight of each of these forward links is the page rank divided by the count of the outgoing links. The ranks are propagated in a document collection until they converge.

4.11 Further Reading

Statistical techniques have been applied first to speech recognition, lexicography, and later to other domains of linguistics. Their use has been a matter of debate because they opposed Chomsky's competence model. For a supporting review and a historical turning point, see the special issues of *Computational Linguistics* (1993, 1 and 2).

Interested readers will there find details on χ^2 tests and likelihood ratios to improve collocation detection in Dunning (1993). Other methods to obtain semantic clusters have been described by Brown et al. (1992). Manning and Schütze (1999) describe statistical methods in detail.

There are several language modeling toolkits available from the Internet. The SRI Language Modeling collection (Stolcke 2002) is a C++ package to create and experiment language models (http://www.speech.sri.com). The CMU-Cambridge Statistical Language Modeling Toolkit (Clarkson and Rosenfeld 1997) is another set of tools (http://svr-www.eng.cam.ac.uk/~prc14/toolkit.html).

Exercises

4.1. Retrieve a text you like on the network. Give the five most frequent words.

4.2. Write a Prolog program that connects to a Web site, and explore hypertext Web links using a breadth-first strategy.

4.3. Implement a Prolog program to obtain bigrams and their statistics.

4.4. Implement a Prolog program to obtain trigrams and their statistics.

4.5. Retrieve a text you like on the network. Give the five most frequent bigrams and trigrams.

4.6. Retrieve a text you like on the network. Divide it into a training set and a test set. Implement the Laplace rule either in Perl or in Prolog. Learn the probabilities on the training set and compute the perplexity of the test set.

4.7. Retrieve a text you like on the network. Divide it into a training set and a test set. Implement the Good–Turing estimation either in Perl or in Prolog. Learn the probabilities on the training set and compute the perplexity of the test set.

4.8. Implement the mutual information test in Prolog.

4.9. Implement the t-score test in Prolog.

4.10. Implement the likelihood ratio in Perl.

4.11. Implement the mutual information test with a window of five words to the left and to the right of the word.