3

Encoding, Entropy, and Annotation Schemes

3.1 Encoding Texts

At the most basic level, computers only understand binary digits and numbers. Corpora as well as any computerized text have to be converted into a digital format to be read by machines. From their American early history, computers inherited encoding formats designed for the English language. The most famous one is the American Standard Code for Information Interchange (ASCII). Although well established for English, the adaptation of ASCII to other languages led to clunky evolutions and many variants. It ended (temporarily?) with Unicode, a universal scheme compatible with ASCII and intended to cover all the scripts of the world.

We saw in Chap. 2 that some corpora include linguistic information to complement raw texts. This information is conveyed through annotations that describe quantities of structures. They range from text organization, such as titles, paragraphs, and sentences, to semantic information including grammatical data, part-of-speech labels, or syntactic structures, etc. In contrast to character encoding, no annotation scheme has yet reached a level where it can claim to be a standard. However, the Extensible Markup Language (XML), a language to define annotations, is well underway to unify them under a shared markup syntax. XML in itself is not an annotation language. It is a scheme that enables users to define annotations within a specific framework.

In this chapter, we will introduce the most useful character encoding schemes and review the basics of XML. We will examine related topics of standardized presentation of time and date, and how to sort words in different languages. Finally, we will introduce two significant theoretical concepts behind codes – entropy and perplexity – how they can help design efficient codes, and how we can use them in a machine-learning algorithm.

3.2 Character Sets

3.2.1 Representing Characters

Words, at least in European languages, consists of characters. Prior to any further digital processing, it is necessary to build an encoding scheme that maps the character or symbol repertoire of a language to numeric values – integers. The Baudot code is one of the oldest electric codes. It uses five bits and hence has the capacity to represent $2^5 = 32$ characters: the Latin alphabet and some control commands like the carriage return, the bell. The ASCII code uses seven bits. It can represent $2^7 = 128$ symbols with positive integer values ranging from 0 to 127. The characters use the contiguous positions from 32 to 126. The values in the interval [0, 31] and 127 correspond to controls used, for instance, in data transmission (Table 3.1).

Table 3.1. The ASCII character set.

Code	Char	Code	Char	Code	Char	Code	Char
32		33	!	34	"	35	#
36	$	37	%	38	&	39	’
40	(	41	)	42	*	43	+
44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3
52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;
60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C
68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K
76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S
84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[
92	\	93	]	94	^	95	_
96	‘	97	a	98	b	99	c
100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k
108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s
116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{
124	|	125	}	126	~	127	

ASCII was created originally for English. It cannot handle other European languages that have accented letters, such as *é*, *à*, or other diacritics like *ø* and *ä*, not to mention languages that do not use the Latin alphabet. Table 3.2 shows characters used in French and German that are ignored by ASCII. Most computers used to

represent characters on octets – words of eight bits – and ASCII was extended with the eighth unoccupied bit to the values [128, 255] (2^8 = 256). Unfortunately, these extensions were not standardized and depended on the operating system. The same character, for instance, *ê*, could have a different encoding in the Windows, Macintosh, and Unix operating systems.

Table 3.2. Characters specific to French and German.

	French															**German**			
Lowercase	à	â	æ	ç	é	è	ê	ë	î	ï	ô	œ	ù	û	ü	ÿ	ä	ö	ü	ß
Uppercase	À	Â	Æ	Ç	É	È	Ê	Ë	Î	Ï	Ô	Œ	Ù	Û	Ü	Ÿ	Ä	Ö	Ü	

The ISO Latin 1 character set (ISO-8859-1) is a standard that tried to reconcile Western European character encodings (Table 3.3). Unfortunately, Latin 1 was ill-designed and forgot characters such as the French *Œ*, *œ*, the German quote „ or the Dutch ij, IJ. Operating systems such as Windows and Mac OS used a variation of it that they had to complement with the missing characters. They used positions in the interval ranging from 128 to 159 (Table 3.4). Later, ISO Latin 9 (ISO-8859-15) updated Latin 1. It restored forgotten French and Finnish characters and added the euro currency sign, €.

3.2.2 Unicode

While ASCII has been very popular, its 128 positions could not support the characters of many languages in the world. Therefore a group of companies formed a consortium to create a new, universal coding scheme: Unicode. Unicode is quickly replacing older encoding schemes, and Windows, Mac OS, and Java platforms now adopt it while sometimes ensuring backward compatibility.

The initial goal of Unicode was to define a superset of all other character sets, ASCII, Latin 1, and others, to represent all the languages of the world. The Unicode consortium has produced character tables of most alphabets and scripts of European, Asian, African, and Near Eastern languages, and assigned numeric values to the characters. Unicode started with a 16-bit code that could represent up to 65,000 characters. It has subsequently been extended to 32 bits.

The Universal Character Set (UCS) is the standardized name of the Unicode character representation. The 2-octet code (UCS-2) is called the Basic Multilingual Plane (BMP). All common characters fit on 16 bits, with the exception of some Chinese ideograms. The 4-octet code (UCS-4) can represent more than a million characters. They cover all the UCS-2 characters and rare characters: historic scripts, some mathematical symbols, private characters, etc.

Unicode groups characters or symbols by script – Latin, Greek, Cyrillic, Hebrew, Arabic, Indic, Japanese, Chinese – and identifies each character by a single hexadecimal number, called the code point, and a name as

Table 3.3. The ISO Latin 1 character set (ISO-8859-1).

Code	Char	SGML	Code	Char	SGML	Code	Char	SGML
160			161	¡	¡	162	¢	¢
163	£	£	164	¤	¤	165	¥	¥
166	¦	&brvbar	167	§	§	168	¨	¨
169	©	©	170	ª	ª	171	«	«
172	¬	¬	173	-		174	®	&reg
175	¯	¯	176	°	°	177	±	±
178	²	²	179	³	³	180	´	´
181	μ	µ	182	¶	¶	183	·	·
184	¸	¸	185	¹	¹	186	º	º
187	»	»	188	¼	¼	189	½	½
190	¾	¾	191	¿	¿	192	À	À
193	Á	Á	194	Â	Â	195	Ã	Ã
196	Ä	Ä	197	Å	Å	198	Æ	&Aelig;
199	Ç	Ç	200	È	È	201	É	É
202	Ê	Ê	203	Ë	Ë	204	Ì	Ì
205	Í	Í	206	Î	Î	207	Ï	Ï
208	Ð	Ð	209	Ñ	Ñ	210	Ò	Ò
211	Ó	Ó	212	Ô	Ô	213	Õ	Õ
214	Ö	Ö	215	×	×	216	Ø	Ø
217	Ù	Ù	218	Ú	Ú	219	Û	Û
220	Ü	Ü	221	Ý	Ý	222	Þ	Þ
223	ß	ß	224	à	à	225	á	á
226	â	â	227	ã	ã	228	ä	ä
229	å	å	230	æ	æ	231	ç	ç
232	è	è	233	é	é	234	ê	ê
235	ë	ë	236	ì	ì	237	í	í
238	î	î	239	ï	ï	240	ð	ð
241	ñ	ñ	242	ò	ò	243	ó	ó
244	ô	ô	245	õ	õ	246	ö	ö
247	÷	÷	248	ø	ø	249	ù	ù
250	ú	ú	251	û	û	252	ü	ü
253	ý	ý	254	þ	þ	255	ÿ	ÿ

```
U+0041 LATIN CAPITAL LETTER A
U+0042 LATIN CAPITAL LETTER B
U+0043 LATIN CAPITAL LETTER C

...

U+0391 GREEK CAPITAL LETTER ALPHA
U+0392 GREEK CAPITAL LETTER BETA
U+0393 GREEK CAPITAL LETTER GAMMA
```

The `U+` symbol means that the number after it corresponds to a Unicode position.

Table 3.4. The Windows and Mac OS extensions to the ISO Latin 1 set represent some of the forgotten Western European characters, here Windows Latin 1 or Windows-1252.

Code	Char	Code	Char	Code	Char	Code	Char
128	€	129		130	‚	131	*f*
132	„	133	…	134	†	135	‡
136	ˆ	137	‰	138	Š	139	‹
140	Œ	141		142	Ž	143	
144		145	‘	146	’	147	“
148	”	149	•	150	–	151	—
152	˜	153	™	154	š	155	›
156	œ	157		158	ž	159	Ÿ

Unicode also allows the composition of accented characters from a base character and one or more diacritics. That is the case for the French $\hat{E}$ or the Scandinavian $\mathring{A}$, which can be defined as combinations. They are created by typing a sequence of two keys: E + ˆ and A + °, corresponding to

```
U+0045 LATIN CAPITAL LETTER E
U+0302 COMBINING CIRCUMFLEX ACCENT
U+0041 LATIN CAPITAL LETTER A
U+030A COMBINING RING ABOVE
```

Both characters also have a single code point:

```
U+00CA LATIN CAPITAL LETTER E WITH CIRCUMFLEX
U+00C5 LATIN CAPITAL LETTER A WITH RING ABOVE
```

The resulting graphical symbol is called a grapheme. A grapheme is a “natural” character or a symbol. It may correspond to a single code point as *E* or *A*, or result from a composition as $\hat{E}$ or $\mathring{A}$.

Unicode allocates contiguous blocks of code to scripts from U+0000. They start with alphabetic scripts: Latin, Greek, Cyrillic, Hebrew, Arabic, etc., then the symbols area, and Asian ideograms or alphabets. Ideograms used by the Chinese, Japanese, and Korean (CJK) languages are unified to avoid duplication. Table 3.5 shows the script allocation. The space devoted to Asian scripts occupies most of the table.

3.2.3 The Unicode Encoding Schemes

Unicode offers three major different encoding schemes: UTF-8, UTF-16, and UTF-32. The UTF schemes – Unicode transformation format – encode the same data by units of 8, 16, or 32-bits and can be converted from one to another without loss.

UTF-16 was the original encoding scheme when Unicode started with 16 bits. It uses fixed units of 16 bits – 2 bytes – to encode directly most characters. The code units correspond to the sequence of their code points using precomposed characters, such as $\hat{E}$ in *FÊTE*

Table 3.5. Unicode subrange allocation of the Universal Character Set (simplified).

Code	Name	Code	Name
U+0000	Basic Latin	U+1400	Unified Canadian Aboriginal Syllabic
U+0080	Latin-1 Supplement	U+1680	Ogham, Runic
U+0100	Latin Extended-A	U+1780	Khmer
U+0180	Latin Extended-B	U+1800	Mongolian
U+0250	IPA Extensions	U+1E00	Latin Extended Additional
U+02B0	Spacing Modifier Letters	U+1F00	Extended Greek
U+0300	Combining Diacritical Marks	U+2000	Symbols
U+0370	Greek	U+2800	Braille Patterns
U+0400	Cyrillic	U+2E80	CJK Radicals Supplement
U+0530	Armenian	U+2F80	KangXi Radicals
U+0590	Hebrew	U+3000	CJK Symbols and Punctuation
U+0600	Arabic	U+3040	Hiragana, Katakana
U+0700	Syriac	U+3100	Bopomofo
U+0780	Thaana	U+3130	Hangul Compatibility Jamo
U+0900	Devanagari, Bengali	U+3190	Kanbun
U+0A00	Gurmukhi, Gujarati	U+31A0	Bopomofo Extended
U+0B00	Oriya, Tamil	U+3200	Enclosed CJK Letters and Months
U+0C00	Telugu, Kannada	U+3300	CJK Compatibility
U+0D00	Malayalam, Sinhala	U+3400	CJK Unified Ideographs Extension A
U+0E00	Thai, Lao	U+4E00	CJK Unified Ideographs
U+0F00	Tibetan	U+A000	Yi Syllables
U+1000	Myanmar	U+A490	Yi Radicals
U+10A0	Georgian	U+AC00	Hangul Syllables
U+1100	Hangul Jamo	U+D800	Surrogates
U+1200	Ethiopic	U+E000	Private Use
U+13A0	Cherokee	U+F900	Others

```
0046 00CA 0054 0045
```

or composing it as with E+ˆ in FEˆTE

```
0046 0045 0302 0054 0045
```

UTF-8 is a variable-length encoding. It maps the ASCII code characters U+0000 to U+007F to their byte values 00 to 7F. It then takes on the legacy of ASCII. All the other characters in the range U+007F to U+FFFF are encoded as a sequence of two or more bytes. Table 3.6 shows the mapping principles of the 32-bit character code points to 8-bit units.

Let us encode *FÊTE* in UTF-8. The letters *F*, *T*, and *E* are in the range U-00000000 – U-0000007F. Their numeric code values are exactly the same in ASCII and UTF-8. The code point of *Ê* is `U+00CA` and is in the range U-00000080 – U-000007FF. Its binary representation is 0000 0000 1100 1010. UTF-8 uses the 11 rightmost bits of `00CA`. The first five underlined bits together with the prefix 110

Table 3.6. Mapping of 32-bit character code points to 8-bit units according to UTF-8. The xxx corresponds to the rightmost bit values used in the character code points.

Range	Encoding
U-00000000 – U-0000007F	0xxxxxxx
U-00000080 – U-000007FF	110xxxxx 10xxxxxx
U-00000800 – U-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
U-00010000 – U-001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
U-00200000 – U-03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
U-04000000 – U-7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

form the octet 1100 0011 that corresponds to `C3` in hexadecimal. The seven next boldface bits with the prefix 10 form the octet 10**00 1010** or `8A` in hexadecimal. The letter $\hat{E}$ is then encoded as 1100 0011 1000 1010 or `C3 8A` in UTF-8. Hence, the word FÊTE and the code points U+`0046 U+00CA U+0054 U+0045` are encoded as

```
46 C3 8A 54 45
```

UTF-32 represents exactly the codes points by their code values. One question remains: How does UTF-16 represent the code points above U+FFFF? The answer is: it uses two surrogate positions consisting of a high surrogate in the range U+DC00 .. U+DFFF and a low surrogate in the range U+D800 .. U+DBFF. This is made possible because the Unicode consortium does not expect to assign characters beyond the code point U+10FFFF. Using the two surrogates, characters between U+10000 and U+10FFFF can be converted from UTF-32 to UTF-16, and vice versa.

Finally, the storage requirements of the Unicode encoding schemes are, of course, different and depend on the language. A text in English will have approximately the same size in ASCII and in UTF-8. The size of the text will be doubled in UTF-16 and four times its original size in UTF-32, because all characters take four bytes.

A text in a Western European language will be larger in UTF-8 than in ASCII because of the accented characters: a nonaccented character takes one octet, and an accented one takes two. The exact size will thus depend on the proportion of accented characters. The text size will be twice its ASCII size in UTF-16. Characters in the surrogate space take 4 bytes, but they are very rare and should not increase the storage requirements. UTF-8 is then more compact for most European languages. This is not the case with other languages. A Chinese or Indic character takes, on average, three bytes in UTF-8 and only two in UTF-16.

3.3 Locales and Word Order

3.3.1 Presenting Time, Numerical Information, and Ordered Words

In addition to using different sets of characters, languages often have specific presentations for times, dates, numbers, or telephone numbers, even when they are restricted to digits. Most European languages outside English would write $\pi = 3,14159$ instead of $\pi = 3.14159$. Inside a same language, different communities may have different presentation conventions. The US English date February 24, 2003, would be written 24 February 2003 or February 24th, 2003, in England. It would be abridged 2/24/03 in the United States, 24/02/2003 in Britain, and 2003/02/24 in Sweden. Some communities may be restricted to an administration or a company, for instance, the military in the US, which writes times and dates differently than the rest of the society.

The International Organization for Standardization (ISO) has standardized the identification of languages and communities under the name of **locales**. Each locale uses a set of rules that defines the format of dates, times, numbers, currency, and how to sort – **collate** – strings of characters. A locale is defined by three parameters: the language, the region, and the variant that corresponds to more specific conventions used by a restricted community. Table 3.7 shows some locales for English, French, and German.

Table 3.7. Examples of locales.

Locale	Language	Region	Variant
English (United States)	en	US	
English (United Kingdom)	en	GB	
French (France)	fr	FR	
French (Canada)	fr	CA	
German (Germany)	de	DE	
German (Austria)	de	AT	

One of the most significant features of a locale is the collation component that defines how to compare and order strings of characters. In effect, elementary sorting algorithms consider the ASCII or Unicode values with a predefined comparison operator such as the inequality predicate `@</2` in Prolog. They determine the lexical order using the numerical ranking of the characters.

These basic sorting procedures do not arrange the words in the classical dictionary order. In ASCII as well as in Unicode, lowercase letters have a greater code value than uppercase ones. A basic algorithm would then sort *above* after *Zambia*, which would be quite misleading for most users.

Current dictionaries in English, French, and German use a different convention. The lowercase letters precede their uppercase equivalents when the strings are equal except for the case. Table 3.8 shows the collation results for some strings.

Table 3.8. Sorting with the ASCII code comparison and the dictionary order.

ASCII order	Dictionary order
ABC	abc
Abc	Abc
Def	ABC
aBf	aBf
abc	def
def	Def

A basic sorting algorithm may suffice for some applications. However, most of the time it would be unacceptable when the ordered words are presented to a user. The result would be even more confusing with accented characters, since their location is completely random in the extended ASCII tables.

In addition, the lexicographic ordering of words varies from language to language. French and English dictionaries sort accented letters as nonaccented ones, except when two strings are equal except for the accents. Swedish dictionaries treat the letters *Å*, *Ä*, and *Ö* as distinct symbols of the alphabet and sort them after *Z*. German dictionaries have two sorting standards. They process accented letters either as single characters or as couples of nonaccented letters. In the latter case, *Ä*, *Ö*, *Ü*, and *ß* are considered respectively as *AE*, *OE*, *UE*, and *ss*.

3.3.2 The Unicode Collation Algorithm

The Unicode consortium has defined a collation algorithm that takes into account the different practices and cultures in lexical ordering. It can be parameterized to cover most languages and conventions. It uses three levels of difference to compare strings. We outline their features for European languages and Latin scripts:

- The primary level considers differences between base characters, for instance, between *A* and *B*.
- If there are no differences at the first level, the secondary level considers the accents on the characters.
- And finally, the third level considers the case differences between the characters.

These level features are general, but not universal. Accents are a secondary difference in many languages, but we saw that Swedish sorts accented letters as individual ones and hence sets a primary difference between *A* and *Å*, or *o* and *Ö*. Depending on the language, the levels may have other features.

To deal with the first level, the Unicode collation algorithm defines classes of letters that gather upper- and lowercase variants, accented and unaccented forms. Hence, we have the ordered sets: {a, A, á, Á, à, À, etc.} < {b, B} < {c, C, ć, Ć, ĉ, Ĉ, ç, Ç, etc.} < {e, E, é, É, è, È, ê, Ê, ë, Ë, etc.} < ....

The second level considers the accented letters if two strings are equal at the first level. Accented letters are ranked after their nonaccented counterparts. The first

accent is the acute one (´), then come the grave accent (`), the circumflex (ˆ), and the umlaut (¨). So, instances of letter *E* with accents, in lower- and uppercase have the order: {e, E} << {é, É} << {è, È} << {ê, Ê} << {ë, Ë}, where << denotes a difference at the second level. The comparison at the second level is done from the left to the right of a word in English and most languages. It is carried out from the right to the left in French, i.e., from the end of a word to its beginning.

Similarly, the third level considers the case of letters when there are no differences at the first and second levels. Lowercase letters are before uppercase ones, that is, {a} <<< {A}, where <<< denotes a difference at the third level.

Table 3.9 shows the lexical order of *pêcher* 'peach tree' and *Péché* 'sin', together with various conjugated forms of the verbs *pécher* 'to sin' and *pêcher* 'to fish' in French and English. The order takes the three levels into account and the reversed direction of comparison in French for the second level. German adopts the English sorting rules for these accents.

Table 3.9. Lexical order of words with accents. Note the reversed order of the second level comparison in French.

English	French
Péché	*pèche*
PÉCHÉ	*pêche*
pèche	*Pêche*
pêche	*Péché*
Pêche	*PÉCHÉ*
pêché	*pêché*
Pêché	*Pêché*
pécher	*pécher*
pêcher	*pêcher*

Some characters are expanded or contracted before the comparison. In French, the letters *Œ* and *Æ* are considered as pairs of two distinct letters: *OE* and *AE*. In traditional German used in telephone directories, *Ä*, *Ö*, *Ü*, and *ß* are expanded into *AE*, *OE*, *UE*, and *ss* and then sorted as an accent difference with the corresponding letter pairs. In traditional Spanish, *Ch* is contracted into a single letter that sorts between *Cz* and *D*.

The implementation of the collation algorithm first maps the characters onto collation elements that have three numerical fields to express the three different levels of comparison. Each character has constant numerical fields that are defined in a collation element table. The mapping may require a preliminary expansion, as for *æ* and *œ* into *ae* and *oe* or a contraction. The algorithm then forms for each string the sequence of the collation elements of its characters. It creates a sort key by rearranging the elements of the string and concatenating the fields according to the levels: the first fields of the string, then second fields, and third ones together. Finally, the algorithm compares two sort keys using a binary comparison that applies to the first

level, to the second level in case of equality, and finally to the third level if levels 1 and 2 show no differences.

3.4 Markup Languages

3.4.1 A Brief Background

Corpus annotation uses sets of labels, also called markup languages. Corpus markup languages are comparable to those of standard word processors such as Microsoft Word or LaTeX. They consist of tags inserted in the text that request, for instance, to start a new paragraph, or to set a phrase in italics or in bold characters. Among the most widespread markup languages, there are the Rich Text Format (RTF) from Microsoft (2004) and the (La)TeX format designed by Donald Knuth (Knuth 1986) (Table 3.10).

Table 3.10. Some formatting tags in RTF and LaTeX.

	Text in italics	**New paragraph**	**Accented letter *é***
RTF	`{\i text in italics}`	`\par`	`\'e9`
LaTeX	`{\it text in italics}`	`\cr`	`\'{e}`

While RTF and LaTeX are used by communities of million of persons, they are not acknowledged as standards. The Standard Generalized Markup Language (SGML) takes this place. SGML could have failed and remained a forgotten international initiative. But the Internet and the World Wide Web, which use Hypertext Markup Language (HTML), a specific implementation of SGML, have ensured its posterity. In the next sections, we introduce the Extensible Markup Language (XML), which builds on the simplicity of HTML that has secured its success, and extends it to handle any kind of data.

3.4.2 An Outline of XML

XML is a coding framework: a language to define ways of structuring documents. XML can incorporate logical and presentation markups. Logical markups describe the document structure and organization such as, for instance, the title, the sections, and inside the sections, the paragraphs. Presentation markups describe the text appearance and enable users to set a sentence in italic or bold type, or to insert a page break. Contrary to other markup languages, like HTML, XML does not have a predefined set of tags. The programmer defines them together with their meaning.

XML separates the definition of structure instructions from the content – the data. Structure instructions are described in a document type definition (DTD) that models a class of XML documents. DTDs correspond to specific tagsets that enable users to mark up texts. A DTD lists the legal tags and their relationships with other tags,

for instance, to define what is a chapter and to verify that it contains a title. Among coding schemes defined by DTDs, there are:

- the Extensible Hypertext Markup Language (XHTML), a clean, XML implementation of HTML that models the Internet Web pages
- the Text Encoding Initiative (TEI), which is used by some academic projects to encode texts

A DTD is composed of three kinds of components defined in the XML jargon as elements, attributes, and entities. Comments of DTDs and XML documents are enclosed between `<!--` and `-->` tags.

Elements. Elements are the logical units of an XML document. They are delimited by surrounding tags. A start tag enclosed between angle brackets precedes the element content, and an end tag terminates it. End tags are the same as start tags with a / prefix. XML tags must be balanced, which means that an end tag must follow each start tag. Here is a simple example of an XML document:

```
<!-- My first XML document -->
<book>
  <title>Language Processing Cookbook</title>
  <author>Pierre Cagné</author>
  <!-- Image to show on the cover -->
  <img></img>
  <text>Here comes the text!</text>
</book>
```

where `<book>` and `</book>` are legal tags indicating respectively the start and the end of the book, and `<title>` and `</title>` the beginning and the end of the title. **Empty elements**, such as the image `<img></img>`, can be abridged as `<img/>`. Unlike HTML, XML tags are case sensitive: `<TITLE>` and `<title>` define different elements.

Attributes. An element can have attributes, i.e., a set of properties attached to the element. Let us complement our `book` example so that the `<title>` element has an alignment whose possible values are flush left, right, or center, and a character style taken from underlined, bold, or italics. Let us also indicate where `<img>` finds the image file. The DTD specifies the possible attributes of these elements and the value list among which the actual attribute value will be selected. The actual attributes of an element are supplied as name–value pairs in the element start tag.

Let us name the alignment and style attributes `align` and `style` and set them in boldface characters and centered, and let us store the image file of the `img` element in the `src` attribute. The markup in the XML document will look like:

```
<title align="center" style="bold">
  Language Processing Cookbook
</title>
```

```
<author>Pierre Cagné</author>
<img src="pierre.jpg"/>
```

Entities. Finally, entities correspond to data stored somewhere in a computer. They can be accented characters, symbols, strings as well as text or image files. The programmer can declare or define variables referring to entities and use them subsequently. There are two different types of entities: parameter entities are used in DTDs, and general entities or simply entities are used in XML document contents. The two types of entities correspond to two different contexts. They are declared and referred to differently.

An entity is referred to within an XML document by enclosing its name between the start delimiter "`&`" and the end delimiter "`;`", such as `&EntityName;`. The XML parser will substitute the reference with the content of `EntityName` when it is encountered.

There are five predefined entities recognized by XML. They correspond to characters used by the XML standard, which cannot be used as is in a document (Table 3.11). References to parameter entities use "`%`" and "`;`" as delimiters, such as `%ParameterEntityName;`. Parameter entity references can only occur in DTDs.

Table 3.11. The predefined entities of XML.

Symbol	Entity encoding
<	`<` (less than)
>	`>` (greater than)
&	`&`
"	`&quot;`
'	`&apos;`

3.4.3 Writing a DTD

The DTD specifies the formal structure of a document type. It enables an XML parser to determine whether a document is valid. The DTD file contains the description of all the legal elements, attributes, and entities.

Elements. The description of the elements is enclosed between the start and end delimiters `<!ELEMENT` and `>`. It contains the element name and the content model in terms of other elements or reserved keywords (Table 3.12). The content model specifies how the elements appear, their order, and their number of occurrences (Table 3.13). For example:

```
<!ELEMENT book (title, (author | editor)?, img,
  chapter+)>
<!ELEMENT title (#PCDATA)>
```

states that a book consists of a title, a possible author or editor, an image img, and one or more chapters. The title consists of PCDATA, that is, only text with no other embedded elements.

Table 3.12. Character types.

Character type	Description
PCDATA	Parsed character data. This data will be parsed and must only be text, punctuation, and special characters; no embedded elements
ANY	PCDATA or any DTD element
EMPTY	No content – just a placeholder

Table 3.13. List separators and occurrence indicators.

List notation	Description
,	Elements must all appear and be ordered as listed
|	Only one element must appear (exclusive or)
+	Compulsory element (one or more)
?	Optional element (zero or one)
*	Optional element (zero or more)

Attributes. Attributes are the possible properties of the elements. Attribute lists are usually defined after the element they refer to. Their description is enclosed between the delimiters <!ATTLIST and >. An attribute list contains:

- the element the attribute is referring to
- the attribute name
- the kind of value the attribute may take: a predefined type (Table 3.14) or an enumerated list of values between brackets and separated by vertical bars
- the default value between quotes or a predefined keyword (Table 3.15)

For example:

```
<!ATTLIST title
  style (underlined | bold | italics) "bold"
  align (left | center | right) "left">
<!ATTLIST author
  style (underlined | bold | italics) #REQUIRED>
```

says that title has two attributes, style and align. The style attribute can have three possible values and, if not specified in the XML document, the default value will be bold. author has one style attribute that must be specified in the document.

Table 3.14. Some XML attribute types.

Attribute types	Description
CDATA	The string type: any character except <, >, &, ', and "
ID	An identifier of the element unique in the document; ID must begin with a letter, an underscore, or a colon
IDREF	A reference to an identifier
NMTOKEN	String of letters, digits, periods, underscores, hyphens, and colons. It is more restrictive than CDATA, for instance, spaces are not allowed

Table 3.15. Some default value keywords.

Predefined default values	Description
#REQUIRED	A value must be supplied
#FIXED	The attribute value is constant and must be equal to the default value
#IMPLIED	If no value is supplied, the processing system will define the value

Entities. Entities can be used to insert non-ASCII symbols or characters. Character references consist of a Unicode number delimited by "&#x" and ";", such as É for *É* and © for ©.

Entities also enable users to define variables. Their declaration is enclosed between the delimiters <!ENTITY and >. It contains the entity name and the entity content (possibly a sequence):

```
<!ENTITY myEntity "Introduction">
```

Parameter entities have a "%" sign before the entity name, as in

```
<!ENTITY % myEntity "<!ELEMENT textbody (para)+>">
```

A DTD Example. Let us now suppose that we want to publish cookbooks. We define a document type, and we declare the rules that will form its DTD: a book will consist of a title, a possible author or editor, an image, one or more chapters, and one or more paragraphs in these chapters. Let us then suppose that the main title and the chapter titles can be in bold, in italics, or underlined. Let us finally suppose that the chapter titles can be numbered in Roman or Arabic notation. The DTD elements and attributes are

```
<!ELEMENT book (title, (author | editor)?, img,
  chapter+)>
<!ELEMENT title (#PCDATA)>
<!ATTLIST title style (u | b | i) "b">
<!ELEMENT author (#PCDATA)>
<!ATTLIST author style (u | b | i) "i">
```

```
<!ELEMENT editor (#PCDATA)>
<!ATTLIST editor style (u | b | i) "i">
<!ELEMENT img EMPTY>
<!ATTLIST img src CDATA #REQUIRED>
<!ELEMENT chapter (subtitle, para+)>
<!ATTLIST chapter number ID #REQUIRED>
<!ATTLIST chapter numberStyle (Arabic | Roman)
  "Roman">
<!ELEMENT subtitle (#PCDATA)>
<!ELEMENT para (#PCDATA)>
```

The name of the document type corresponds to the **root element**, here `book`, which must be unique.

XML Schema. You probably noticed that the DTD syntax does not fit very well with that of XML. This bothered some people, who tried to make it more compliant. This gave birth to XML Schema, a document definition standard using the XML style. As of today, DTD is still "king," however, XML Schema is gaining popularity. Specifications are available from the Web consortium at www.w3.org/XML/Schema.

3.4.4 Writing an XML Document

We shall now write a document conforming to the `book` document type. An XML document begins with a header: a declaration describing the XML version and an optional encoding. The default encoding is UTF-8.

```
<?xml version="1.0" encoding="UTF-8"?>
```

The document can contain any Unicode character. The encoding refers to how the characters are stored in the file. This has no significance if you only use unaccented characters in the basic Latin set from position 0 to 127. If you type accented characters, the editor will have to save them as UTF-8 codes. In the document above, *Cagné* must be stored as 43 61 67 6E C3 A9, where *é* is corresponds to C3 A9.

If your text editor does not manage UTF-8, you will have to enter the accented characters as entities with their Unicode number, for instance, `&#xc9;` for *É*, or `&#xE9;` for *é*. Other encodings, such as Latin 1 (ISO-8859-1), Windows-1252, or MacRoman would let you simply type the characters *É* or *é* from your keyboard instead and save it with your machine's default encoding.

Then, the document declares the DTD it uses. The DTD can be inside the XML document and enclosed between the delimiters `<!DOCTYPE [` and `]>`, for instance:

```
<!DOCTYPE book [
<!ELEMENT book (title, (author | editor)?, img,
  chapter+)>
<!ELEMENT title (#PCDATA)>
...
]>
```

Or the DTD can be external to the document, for instance, in a file called `book_definition.dtd`. In this case, `DOCTYPE` indicates its location on the computer using the keyword `SYSTEM`:

```
<!DOCTYPE book SYSTEM
  "/home/pierre/xml/book_definition.dtd">
```

Now, we can write the document content. Let us use the XML tags to sketch a very short book. It could look like this:

```
<book>
  <title style="i">Language Processing Cookbook
  </title>
  <author style="b">Pierre Cagné</author>
  <img src="pierre.jpg"/>
  <chapter number="c1">
    <subtitle>Introduction</subtitle>
    <para>Let&apos;s start doing simple things:
      Collect texts.
    </para>
    <para>First, choose an author you like.
    </para>
  </chapter>
</book>
```

Once, we have written an XML document, we must check that is **well formed**, which means that it has no syntax errors: the brackets are balanced, the encoding is correct, etc. We must also **validate** it, i.e., check that it conforms to the DTD. This can be done with a variety of parsers available from the Internet. An easy way to do it is to use Microsoft Explorer (or any modern Web browser), which has an embedded XML parser.

3.4.5 Namespaces

In our examples, we used element names that can be part of other DTDs. The string `title`, for instance, is used by XHTML. The XML namespaces is a device to avoid collisions. It is a naming scheme that enables us to define groups of elements and attributes in the same document and prevent name conflicts.

We declare a namespace using the predefined `xmlns` attribute as `<my-element xmlns:prefix="URI">`. It starts a namespace inside `my-element` and its descendants, where `prefix` defines a group of names. Names members of this namespace are preceded by the prefix, as in `prefix:title`. `URI` has the syntax of a Web address. However, it is just a unique name; it is never accessed.

Declaring two namespaces in `book`, we can reuse `title` for different purposes:

```
<book
  xmlns:pierre="http://www.cs.lth.se/~pierre"
```

```
    xmlns:raymond="http://www.grandecuisine.com">

    <pierre:title style="i">Language Processing
      Cookbook
    </pierre:title>

    <raymond:title style="i">A French Cookbook
    </raymond:title>

</book>
```

3.5 Codes and Information Theory

Information theory underlies the design of codes. Claude Shannon probably started the field with a seminal article (1948), in which he defined a measure of information: the **entropy**. In this section, we outline essential information theory concepts: entropy, optimal coding, cross entropy, and **perplexity**. Entropy and perplexity are used as metrics in many areas of language processing.

3.5.1 Entropy

Information theory models a text as a sequence of symbols. Let $x_1, x_2, ..., x_N$ be a discrete set of N symbols representing the characters. The **information content** of a symbol is defined as $I(x_i) = -\log_2 p(x_i) = \log_2 \frac{1}{p(x_i)}$, and it is measured in bits. When the symbols have equal probabilities, they are said to be equiprobable and $p(x_1) = p(x_2) = ... = p(x_N) = \frac{1}{N}$. The information content of x_i is then $I(x_i) = \log_2 N$.

The information content corresponds to the number of bits that is necessary to encode the set of symbols. The information content of the alphabet, assuming that it consists of 26 unaccented equiprobable characters and the space, is $\log_2(26 + 1) = 4.75$, which means that five bits are necessary to encode it. If we add 16 accented characters, the uppercase letters, 11 punctuation signs, [, . ; : ? ! " – ( ) ’], and the space, we need $(26 + 16) \times 2 + 12 = 96$ symbols. Their information content is $\log_2 96 = 6.58$, and they can be encoded on seven bits.

The information content assumes that the symbols have an equal probability. This is rarely the case in reality. Therefore this measure can be improved using the concept of entropy, the average information content, which is defined as:

$$H(X) = -\sum_{x \in X} p(x) \log_2 p(x),$$

where X is a random variable over a discrete set of variables, $p(x) = P(X = x), x \in X$, with the convention $0 \log_2 0 = 0$. When the symbols are equiprobable, $H(X) = \log_2 N$. This corresponds also to the upper bound on the entropy value, and for any random variable, we have the inequality $H(X) \leq \log_2 N$.

To evaluate the entropy of printed French, we computed the frequency of the printable French characters in Gustave Flaubert's novel *Salammbô*. Table 3.16 shows the frequency of 26 unaccented letters, the 16 accented or specific letters, and the blanks (spaces).

Table 3.16. Letter frequencies in the French novel *Salammbô* by Gustave Flaubert. The text has been normalized in uppercase letters. The table does not show the frequencies of the punctuation signs or digits.

Letter	Freq	Letter	Freq	Letter	Freq	Letter	Freq
A	42471	*B*	5762	*C*	14226	*D*	18912
E	71178	*F*	4996	*G*	5151	*H*	5315
I	33669	*J*	1220	*K*	92	*L*	30976
M	13101	*N*	32919	*O*	22629	*P*	13178
Q	3965	*R*	33577	*S*	46766	*T*	35110
U	29276	*V*	6924	*W*	1	*X*	2213
Y	1232	*Z*	413	*À*	1893	*Â*	607
Æ	9	*Ç*	452	*È*	2002	*É*	7728
Ê	898	*Ë*	6	*Î*	277	*Ï*	66
Ô	398	*Œ*	121	*Ù*	179	*Û*	213
Ü	0	*Ÿ*	0	Blanks	101,555	**Total:**	591,676

The entropy of the text restricted to the characters in Table 3.16 is defined as:

$$\begin{aligned} H(X) &= -\sum_{x \in X} p(x) \log_2 p(x). \\ &= -p(A) \log_2 p(A) - p(B) \log_2 p(B) - ... \\ &\quad -p(Z) \log_2 p(Z) - p(\grave{A}) \log_2 p(\grave{A}) - ... \\ &\quad -p(\ddot{Y}) \log_2 p(\ddot{Y}) - p(blanks) \log_2 p(blanks). \end{aligned}$$

If we distinguish between upper- and lowercase letters and if we include the punctuation signs, the digits, and all the other printable characters – ASCII ≥ 32 – the entropy of Gustave Flaubert's *Salammbô* in French is $H(X) = 4.39$.

3.5.2 Huffman Encoding

The information content of the French character set is less than the seven bits required by equiprobable symbols. Although it gives no cue on an encoding algorithm, it indicates that a more efficient code is theoretically possible. This is what we examine now with Huffman encoding, which is a general and simple method to build such a code.

Huffman encoding uses variable-length code units. Let us simplify the problem and use only the eight symbols A, B, C, D, E, F, G, and H with the count frequencies in Table 3.17.

Table 3.17. Frequency counts of the symbols.

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
Freq	42,471	5762	14,226	18,912	71,178	4996	5151	5315
Prob	0.25	0.03	0.08	0.11	0.42	0.03	0.03	0.03

Table 3.18. A possible encoding of the symbols on 3 bits.

A	B	C	D	E	F	G	H
000	001	010	011	100	101	110	111

The information content of equiprobable symbols is $\log_2 8 = 3$ bits. Table 3.18 shows a possible code with constant-length units.

The idea of Huffman encoding is to encode frequent symbols using short code values and rare ones using longer units. This was also the idea of the Morse code, which assigns a single signal to letter E: ., and four signals to letter X: -..-.

This first step builds a Huffman tree using the frequency counts. The symbols and their frequencies are the leaves of the tree. We grow the tree recursively from the leaves to the root. We merge the two symbols with the lowest frequencies into a new node that we annotate with the sum of their frequencies. In Fig. 3.1, this new node corresponds to the letters F and G with a combined frequency of 4996 + 5151 = 10,147 (Fig. 3.2). The second iteration merges B and H (Fig. 3.3); the third one, (F, G) and (B, H) (Fig. 3.4), and so on (Figs. 3.5–3.8).

Fig. 3.1. The symbols and their frequencies.

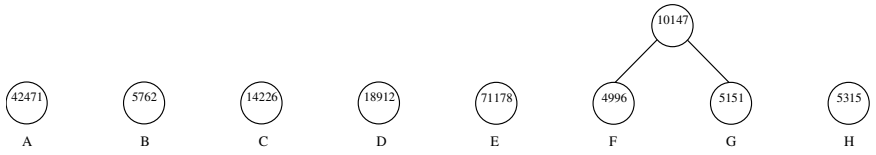

Fig. 3.2. Merging the symbols with the lowest frequencies.

The second step of the algorithm generates the Huffman code by assigning a 0 to the left branches and a 1 to the right branches (Table 3.19).

The average number of bits is the weighted length of a symbol. If we compute it for the data in Table 3.17, it corresponds to:

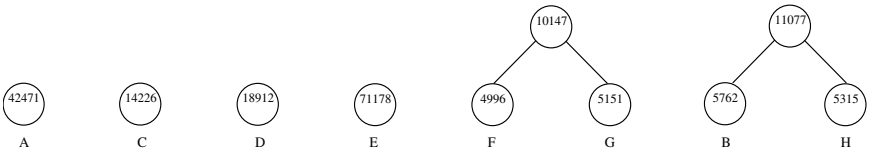

Fig. 3.3. The second iteration.

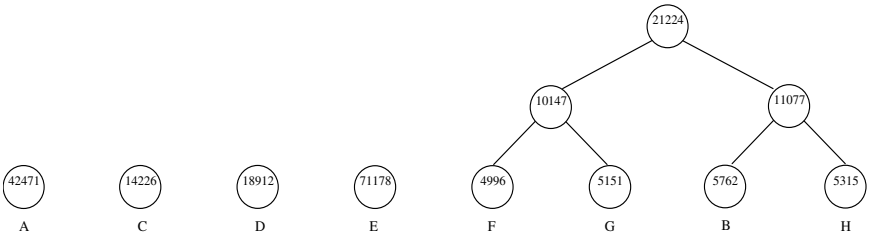

Fig. 3.4. The third iteration.

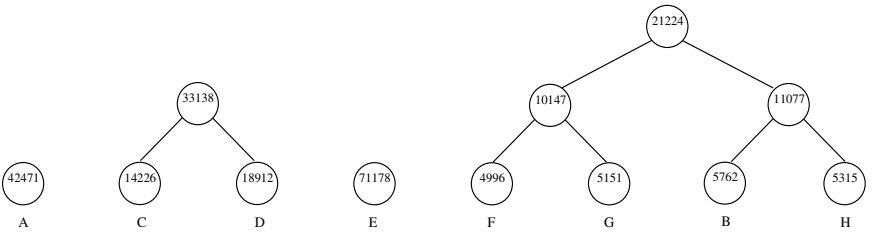

Fig. 3.5. The fourth iteration.

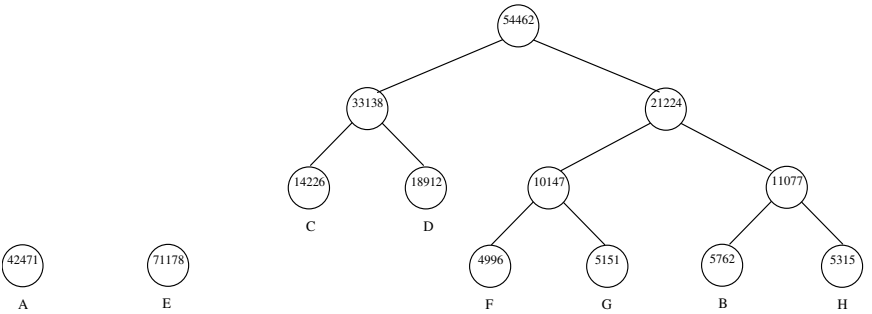

Fig. 3.6. The fifth iteration.

Table 3.19. The Huffman code.

A	B	C	D	E	F	G	H
10	11110	1100	1101	0	11100	11101	11111

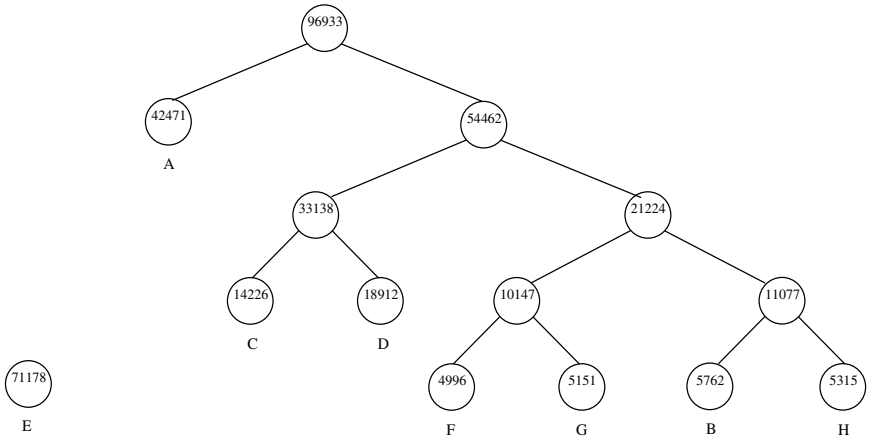

Fig. 3.7. The sixth iteration.

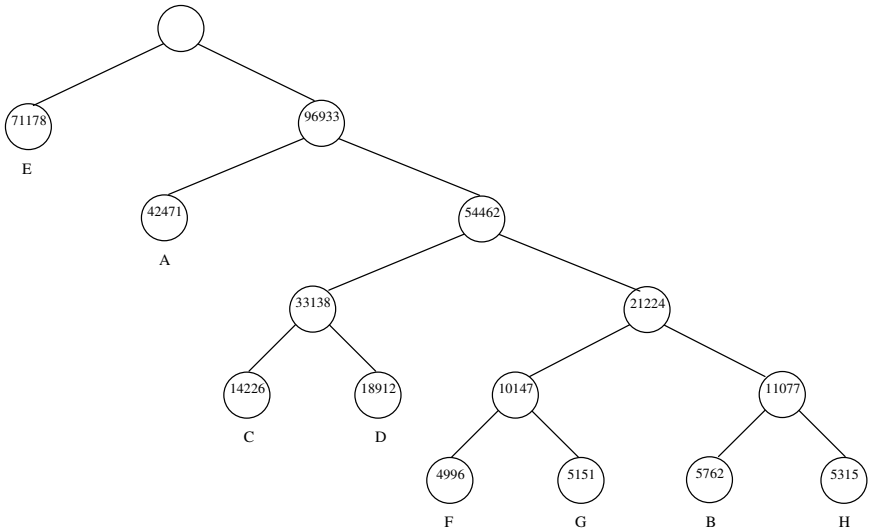

Fig. 3.8. The final Huffman tree.

$$0.25 \times 2 \text{ bit} + 0.03 \times 5 \text{ bit} + 0.08 \times 4 \text{ bit} + 0.11 \times 4 \text{ bit} + 0.42 \times 1 \text{ bit}$$
$$+0.03 \times 5 \text{ bit} + 0.03 \times 5 \text{ bit} + 0.03 \times 5 \text{ bit} = 2.35$$

Although the Huffman code reduces the average number of bits, it does not reach the entropy limit, which is, in our example, 2.27.

3.5.3 Cross Entropy

Let us now compare the letter frequencies between two parts of *Salammbô*, then between *Salammbô* and another text in French or in English. The symbol probabilities

will certainly be different. Intuitively, the distributions of two parts of the same novel are likely to be close, further apart between *Salammbô* and another French text from the 21st century, and even further apart with a text in English. This is the idea of cross entropy, which compares two probability distributions.

In the cross entropy formula, one distribution is referred to as the model. It corresponds to data on which the probabilities have been trained. Let us name it m with the distribution $m(x_1), m(x_2), ..., m(x_N)$. The other distribution, p, corresponds to the test data: $p(x_1), p(x_2), ..., p(x_N)$. The cross entropy of m on p is defined as:

$$H(p, m) = -\sum_{x \in X} p(x) \log_2 m(x).$$

Cross entropy quantifies the average surprise of the distribution when exposed to the model. We have the inequality $H(p) \leq H(p, m)$ for any other distribution m with equality if and only if $m(x_i) = p(x_i)$ for all i. The difference $H(p, m) - H(p)$ is a measure of the relevance of the model: the closer the cross entropy, the better the model.

To see how the probability distribution of Flaubert's novel could fare on other texts, we trained a model on the first fourteen chapters of *Salammbô*, and we applied it to the last chapter of *Salammbô* (Chap. 15), to Victor Hugo's *Notre Dame de Paris*, both in French, and to *Nineteen Eighty-Four* by George Orwell in English. The data in Table 3.20 conform to our intuition. They show that the first chapters of *Salammbô* are a better model of the last chapter of *Salammbô* than of *Notre Dame de Paris* and even better than of *Nineteen Eighty-Four*.

Table 3.20. The entropy is measured on the file itself and the cross entropy is measured with Chapters 1–14 of Gustave Flaubert's *Salammbô* taken as the model.

	Entropy	**Cross entropy**	**Difference**
Salammbô, chapters 1-14, training set	4.39481	4.39481	0.0
Salammbô, chapter 15, test set	4.34937	4.36074	0.01137
Notre Dame de Paris, test set	4.43696	4.45507	0.01811
Nineteen Eighty-Four, test set	4.35922	4.82012	0.46090

3.5.4 Perplexity and Cross Perplexity

Perplexity is an alternate measure of information that is mainly used by the speech processing community. Perplexity is simply defined as $2^{H(X)}$. The cross perplexity is defined similarly as $2^{H(p,m)}$.

Although perplexity does not bring anything new to entropy, it presents the information differently. Perplexity reflects the averaged number of choices of a random variable. It is equivalent to the size of an imaginary set of equiprobable symbols, which is probably easier to understand.

Table 3.21 shows the perplexity and cross perplexity of the same texts measured with Chaps. 1–14 of Gustave Flaubert's *Salammbô* taken as the model.

Table 3.21. The perplexity and cross perplexity of texts measured with Chapters 1–14 of Gustave Flaubert's *Salammbô* taken as the model.

	Perplexity	**Cross perplexity**
Salammbô, chapters 1-14, training set	21.04	21.04
Salammbô, chapter 15, test set	20.38	20.54
Notre Dame de Paris, test set	21.66	21.93
Nineteen Eighty-Four, test set	20.52	28.25

3.6 Entropy and Decision Trees

Decision trees are useful devices to classify objects into a set of classes. They have many applications in language processing. In this section, we will describe what they are and see how entropy can help us learn – or induce – automatically decision trees from a set of data. The algorithm, which resembles a reverse Huffman encoding, is one of the simplest machine-learning techniques.

3.6.1 Decision Trees

Decision trees consider objects defined by a set of attributes – also called features – where the nodes of the trees are conditions on the features. An object is presented at the root of the tree, and its features are tested by the tree nodes from the root down to a leaf. The leaves return an output, which can be the description of an object's membership or probabilities to be the member of a class.

Quinlan (1986) gives an example of a set where objects are members of two classes N and P (Table 3.22) and a decision tree that correctly classifies these objects (Fig. 3.9).

3.6.2 Inducing Decision Trees Automatically

It is possible to design many trees that classify the objects in Table 3.22 successfully. The tree in Fig. 3.9 is interesting because it is efficient: a decision can be made with a minimal number of tests.

An efficient decision tree can be induced from a set of examples, members of mutually exclusive classes using an entropy measure. We will describe the induction using two classes of p positive and n negative examples, although this can be generalized to any number of classes. Each example is defined by a finite number of attributes. Each node in the decision tree corresponds to an attribute that has as many branches as the attribute has possible values.

Table 3.22. A set of object members of two classes: N and P. After Quinlan (1986).

Object	Attributes				Class
	Outlook	**Temperature**	**Humidity**	**Windy**	
1	Sunny	Hot	High	False	N
2	Sunny	Hot	High	True	N
3	Overcast	Hot	High	False	P
4	Rain	Mild	High	False	P
5	Rain	Cool	Normal	False	P
6	Rain	Cool	Normal	True	N
7	Overcast	Cool	Normal	True	P
8	Sunny	Mild	High	False	N
9	Sunny	Cool	Normal	False	P
10	Rain	Mild	Normal	False	P
11	Sunny	Mild	Normal	True	P
12	Overcast	Mild	High	True	P
13	Overcast	Hot	Normal	False	P
14	Rain	Mild	High	True	N

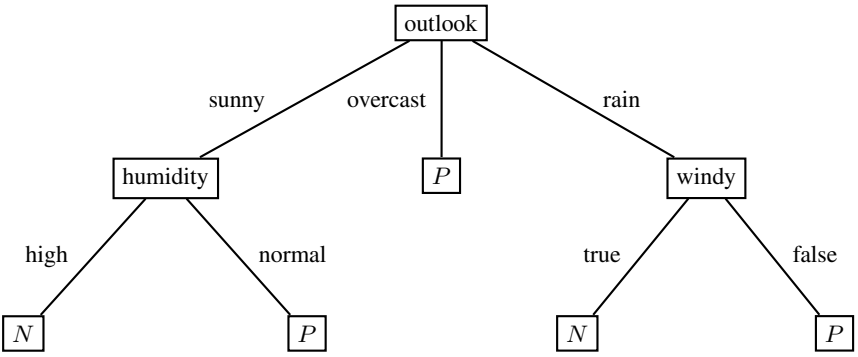

Fig. 3.9. A decision tree classifying the objects in Table 3.22. After Quinlan (1986).

At the root of the tree, the condition must be the most discriminating, that is, have branches gathering most positive examples while others gather negative examples. The ID3 (Quinlan 1986) algorithm uses the entropy to select the best attribute to be the root of the tree and recursively the next attributes of the resulting nodes. ID3 defines the information gain as the difference of entropy before and after the decision. It measures the separating power of an attribute: the more the gain, the better the attribute.

As defined previously, the entropy of a two-class set of p positive and n negative examples is:

$$I(p,n) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}.$$

If attribute A is the root of the tree and has v possible values $\{A_1, A_2, ..., A_v\}$, there will be v resulting nodes. Each node corresponds to one value of A and contains p_i positive and n_i negative examples. Its entropy is $I(p_i, n_i)$.

The weighted average of all the nodes below A is:

$$\sum_{i=1}^{v} \frac{p_i + n_i}{p + n} I(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i}).$$

The information gain is defined as $I_{before} - I_{after}$. For the tree in Fig. 3.9, let us compute the information gain of attribute *outlook*.

$$I_{before}(p, n) = -\frac{9}{14}\log_2\frac{9}{14} - \frac{5}{14}\log_2\frac{5}{14} = 0.940.$$

Outlook has three values: *sunny*, *overcast*, and *rain*. Their corresponding entropies are:

$$I(p_1, n_1) = -\tfrac{2}{5}\log_2\tfrac{2}{5} - \tfrac{3}{5}\log_2\tfrac{3}{5} = 0.971.$$
$$I(p_2, n_2) = 0.$$
$$I(p_3, n_3) = -\tfrac{3}{5}\log_2\tfrac{3}{5} - \tfrac{2}{5}\log_2\tfrac{2}{5} = 0.971.$$

Thus

$$I_{after}(p, n) = \frac{5}{14}I(p_1, n_1) + \frac{4}{14}I(p_{21}, n_2) + \frac{5}{14}I(p_3, n_3) = 0.694.$$

The gain is 0.940 – 0.694 = 0.246, which is the highest among the possible attributes.

The algorithm to build the decision tree is simple. The attribute that has the highest information gain is selected to be the root of the tree, and this process is repeated recursively for each node of the tree.

3.7 Further Reading

Many operating systems such as Windows, Mac OS, and Unix, or programming languages such as Java have adopted Unicode and take the language parameter of a computer into account. Basic lexical methods such as date and currency formatting, word ordering, and indexing are now supported at the operating system level. Operating systems or programming languages offer toolboxes and routines that you can use in applications.

The Unicode Consortium publishes books and technical reports that describe the various aspects of the standard. *The Unicode Standard, Version 4.0* (2003) is the most comprehensive document while Davis and Whistler (2002) describe in detail the Unicode collation algorithm. Both documents are available in electronic format from the Unicode Web site: http://www.unicode.org. IBM implemented a large library of Unicode components in Java and C++, which are available as open-source software (http://www.ibm.com/software/globalization/icu).

SGML started from a US DARPA initiative. Goldfarb (1990) is a difficult-to-read reference to this language by its designer. Derived markup standards such as

HTML and XML are continuously evolving. Their specifications are available from the World Wide Web consortium (http://www.w3.org). Finally, a good reference on XML is *Learning XML* (Ray 2003).

Information theory has been covered by many books, many of them requiring a good mathematical background. The text by Manning and Schütze (Chap. 2, 1999) provides a short and readable introduction oriented toward natural language processing.

We will use ID3 as a machine-learning algorithm in other chapters of this book. ID3 is one of the oldest and easiest-to-understand algorithms. There are many other machine-learning techniques that can use the same type of input data: a set of examples defined by features and members of a finite set of classes. As is the case for ID3, they automatically train classifiers from the annotated examples. Classifiers can then be reused for unannotated data. Support vector machines (Boser et al. 1992), which rely on a complex mathematic formulation, are very efficient devices. They enjoy a growing popularity in the language processing community. Their presentation is beyond the scope of this book. Fortunately, there are many free implementations of them. Schlkopf and Smola (2002) is a good reference on them.

Exercises

3.1. Implement UTF-8 that transforms a sequence of code points in a sequence of octets in Prolog.

3.2. Implement a word collation algorithm for English, French, German, or Swedish.

3.3. Modify the DTD in Sect. 3.4.4 so that the cookbook consists of meals instead of chapters, and each meal has an ingredient and a recipe section.

3.4. Modify the DTD in Sect. 3.4.4 to declare the general and parameter entities:

```
<!ENTITY myEntity "Introduction">
<!ENTITY %myEntity "<!ELEMENT textbody (para)+>">
```

Use these entities in the DTD and the document.

3.5. Write a Prolog program that removes the tags from a text encoded in HTML.

3.6. Write a Prolog program that process a text encoded in HTML: it retains headers (`Hn` tags) and discards the rest.

3.7. Implement the ID3 algorithm in Prolog or Perl.