11

What can corpus software do?

Mike Scott

1. General constraints

It is important to put the computer in its proper place. Just as in the early days of the motor-car some people ran risks we would not think of running now, so in the beginnings of computing there were misapprehensions such as the estimation that very few computers would ever be needed (Cohen 1998), in an age when computers were air-conditioned, weighed tons and were chiefly used by the superpowers for computing ballistic equations. Nowadays there are not likely to be people who view the computer as a semi-magic being, as there were twenty or thirty years ago when the personal computer was a novelty and there was much excitement about artificial intelligence and the 5th Generation. Most of us experience computers every day and that experience is more likely to be frustrating or routine than rewarding. In order to get the best from Corpus Linguistics, though, it is important to know something more about what computers are good at, what they are bad at, and why.

Things computers do really well

Computers don't get tired. As long as they are fed and watered with sufficient electricity and internet connectivity, they go on with routine tasks night and day, even updating their own calendars as daylight-saving comes in, all automatically and with rather little attention. As a matter of fact, most of the time the computer chips are not actually doing anything at all, however. Like Marvin, the android in the science-fantasy story *The Hitch-Hiker's Guide to the Galaxy* (Adams 1978), they spend by far the majority of their clock cycles simply watching and waiting, or in the jargon, *polling*. That is, polling the keyboard to see whether a key has been pressed, polling the inputs to detect incoming e-mails, etc. In Marvin's fictional case the result is boredom and frustration, but in reality computers are quite unable to get bored or frustrated – those are qualities of their users.

Computers can multi-task. Strictly speaking, most of them don't actually do all the operations I shall describe simultaneously, but to the user it appears that they do, because the main computer processing unit (CPU) 'attention' is shared out so that it seems that the computer can multi-task. The computer, at the same time as it is polling the

keyboard, is also displaying results on the screen, calculating more results and, perhaps in another tab or another window, running numerous other operations. If an e-mail arrives, it may play a sound to show that it has recognised the arrival, probably checked the message for spam and possibly deleted it if so. (In that case why did it sound the bell, I wonder? The arrival of spam is not something I want to have my shoulder tapped for!)

The word *computer* comes from the fact that it can carry out routine numerical computations such as adding up numbers, rapidly, tirelessly, always getting the exact same answer. It is also true but less well known that it may actually get the wrong answer. According to Wolfram MathWorld:

> A notorious example is the fate of the Ariane rocket launched on June 4, 1996 … In the 37th second of flight, the inertial reference system attempted to convert a 64-bit floating-point number to a 16-bit number, but instead triggered an overflow error which was interpreted by the guidance system as flight data, causing the rocket to veer off course and be destroyed.
>
> (http://mathworld.wolfram.com/RoundoffError.html, accessed 2008)

The technical parts in that quote refer to how the computer stores its data, in 16-bit, 32-bit or 64-bit chunks. For our purposes in the present chapter, we do not need to concern ourselves with single bits, but it is important to understand that a number or a letter in an alphabet will need to be represented somehow and stored in the machine's memory chips or on a disk or pen-drive, etc., and to understand why, if the storage is not managed properly, an error may occur.

Imagine you want to display a pre-arranged cryptic message to a confederate, that you are being held under house arrest and unable to communicate normally. The only way of sending a message is to cover or obscure one of the panes of a window with a book or a cushion. Suppose further that your window has two panes. Covering the right-hand pane means 'carry on, the hidden supplies have still not been found' whereas the left pane means 'flee abroad, all our supplies have been discovered'. The message has only two possible options and there are only two panes to use for these messages. What if there were a need to communicate more messages, though? Our crude window-pane system could possibly be made to carry a little more information because it would be possible to use the absence of anything covering either pane to mean a third message such as 'we still don't know whether the supplies have been detected', but obviously enough, the method of covering up window-panes will be quite limited as a means of communication and it is unlikely that we could ever express a much larger range of different messages using this system. We might bring in time itself, by establishing a convention that each pane was covered or not covered for a long or a short time, or use a convention of part-covering panes, but still the system is very limited. The point is that for any messaging system, a set of choices must exist (number of panes, covered versus uncovered, etc.) and a system with inherently few choices built into it cannot express many alternative different messages. Meaning implies choice.

In the 64-bit floating point reference given above, what happened is that a floating point number (i.e. involving decimals such as 3.1412456321) was originally stored successfully in an area of computer memory but then copied out into another area of its memory only one quarter the size, into which it didn't really fit, and when that happened it was interpreted as if it had a quite different meaning, as if someone accidentally sat by the left window pane and caused a major panic.

Analogously, the personal computer of the 1980s and 1990s could easily fit a symbol from the English alphabet like *J* or *K* into a small space in its memory, simply because the range of alternatives envisaged was small: all the alphabet from *A* to *Z* and again from *a* to *z*, plus a few punctuation symbols and numbers. The people designing the systems themselves mostly spoke and wrote English, a language that uses a tiny alphabet. The total number of symbols in routine use in those computers was fewer than 300. What about people who wrote in Spanish or German? These needed a few more symbols, such as *ñ* and *Ñ* and *ü* and *Ü* (or else they just didn't bother because without the accents the text was still usually perfectly readable). Fitting those into the *character-set* was not difficult, but fitting Japanese, Hindi, Chinese or Korean symbols in too was really a difficulty. To take account of the many thousands of extra character shapes used in such languages required a bigger storage area than that which was OK for a mere 2–300. Technically, these alternatives fitted into a system which reserved one *byte* for each character, while the many thousands of alternatives of Chinese characters require a system using two or more bytes for each. Just as the complex Ariane rocket number was misinterpreted when it was read into a simpler system, the Chinese character 龍, if stored in a single byte system, would get misperceived as either *ù* or *Ä*. Why those particular characters, you may ask – they don't look a bit like 龍. One way of representing these characters is by the numerical code that has been allocated in the storage system, and for most purposes we use decimal numbers for that. Character 龍 is 63940. In other words, in the Unicode (of which more anon) convention, that particular character is number 63,940 in a set which starts off with characters like *A* and *B*. In computer science a different way of representing the same number, hex, represents the same number as 'F9C4'. If we take that number in byte-sized chunks, we get F9 and C4. Yes, you've guessed it, F9 is the character *ù* and C4 is *Ä*. Reading something intended to be F9C4 as F9 or C4 alone is definitely a mistake. It probably would not crash a rocket, though it might set off a serious dispute if it involved speakers of oriental languages!

Other tasks computers are good at are storing numbers, strings (text), or records (data structures) on disk, reading them and changing them. They don't usually lose them, muddle them up or get them corrupted, at least they do so much less than humans would if asked to retrieve and replace information from file stores very many times.

Computers can sort data into alphabetical or numerical order quickly and reliably; they can straightforwardly display, in fairly accurate colours, text or numbers on a screen or via a printer on paper, if the symbols belong to a standard character set. They can scan an image as a vast array of tiny dots, rather like a digital camera.

Computers remember when something was done. Without your doing anything, your documents get stored with a record of when they were last written to: day, month, year, hour, minute, second – even to the 100th of second. Computers may remember more than you expect: your .docs also almost certainly contain hidden information about who owned the software being used, what has been deleted, printers available and so on. (To see what is in a .doc, just open it using a program such as Microsoft's *Notepad*; it can be quite a shock to see what is in there.)

Computers are also good at storing links between pieces of information: the information can thus be organised in a database, where ideally every piece of data is typed in once only, and every single time the data is needed it is extracted from the database. For example, in a university, the publications, teaching and administrative duties and achievements of staff could each be stored in one place only and the data retried via linkages to generate a portfolio for staff promotion, a report for the national research assessment system, applications for research

awards, staff web-pages, etc. Computers are so good at these linkages, in fact, that the terms *hypertext*, *web* and *networking* have come about with the world of personal computers.

Finally, computers are quite robust in normal use. The parts that most often give trouble are keyboards, disk drives and printers – the bits with moving parts – but by and large the hardware is not likely to break down.

Things computers cannot do at all

Computers do not notice what they are doing at all. Ask them to perform a task 200 times and they will not assume the next request will again be for that same task, or helpfully do it without being asked. This is a tremendous strength. Many human errors come from our noticing patterns of repetition: 'she asked me to pass the salt again' leading to 'she fancies me', or 'they always ring the fire-alarm in the second week of term' to 'carry on, class, we can ignore it'.

Could software for computers be designed so that it would notice what it is doing? Well, it would be easy to do that at a simple level; for example, I have programmed into *WordSmith Tools* (Scott 2009) routines which offer advice when appropriate (such as when it would be best to sort one's data) but which also offer a 'don't tell me again' box – if the user ticks this box the reminder will not be offered next time. The software in a very simple sense learns what that user wants. Perhaps a better example would be if the software kept a log of every operation which each user performed, adding to a database each time, so that as time went by it had increasingly better records of what that user did, much as the supermarket software keeps a log (if you pay by card) of your detailed purchases. It would then be easy for the corpus software to offer a message such as 'shall I select the same set of text files again and get ready for concordancing straight away?' once a pattern had become sufficiently established. It would be as if the supermarket recognised you entering the door and immediately prepared a half-full shopping trolley with your favourite regular items already in there. In other words, it is possible for a programmer to imagine a system for *pseudo-noticing*, giving the appearance of noticing. This is not at all the same as an animal such as a mouse noticing, for example, that there's a smell of cheese and moving towards it, chiefly because it is the programmer who has anticipated certain possible options and events in advance, not the computer, whereas the mouse does its own noticing of varied changes in its environment.

Computers cannot prefer one answer to another, as a mouse could, or complain. Nor could they know what any data *means*. To them, 'to be or not to be' is just a string of bytes, eighteen bytes long (not thirteen as each space also takes up room), or eighteen pairs of bytes in length if written using Unicode, the system which makes Chinese much easier to represent. It is just a string of bytes, not a meaningful question about a quandary as well as a reference to a Shakespeare play. Nor can a computer know what the user *meant*. If s/he leans over the computer and a sleeve accidentally presses a key, there's no way the computer can guess that this was involuntary.

Things computers can do, with difficulty

Comparing strings

Computers can compare 'hell' with 'hello', easily spotting that *hell* is shorter and that *hello* contains *hell* within it, but would not be able to see that 'The hell of war' is similar to

'The hellishness of war' unless previously programmed to treat *hell* and *hellishness* as synonyms. Programming a computer to treat two forms as synonymous is hard. To determine that, say, *start* and *begin* are synonymous is liable to generate **the car won't begin*, or to seriously misread Barry *et al.*'s 2007 article 'START (Screening Tool to Alert Doctors to the Right Treatment) – An Evidence-based Screening Tool to Detect Prescribing Omissions in Elderly Patients'. *May* is sometimes an auxiliary, sometimes a female name, sometimes a month, sometimes a flower – and programming software to decide which is not at all easy. You might like to try the armchair exercise of deciding how you might go about it.

Recognising a pattern

Pattern-recognition is a very tricky exercise. Computers are very good at shuffling data around, re-ordering it, extracting sought strings as in a concordancer. In so doing, they very often present the user with some sort of configuration which to a human looks patterned. But the computer itself will not easily find a pattern. For example, a sequence of *o*s and hyphens makes a pattern:

o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-

and to a human a sequence of four symbols with a hyphen between is a roughly similar but less attractive-looking (because less symmetrical) pattern:

hyug-hyug-hyug-hyug-hyug-hyug-hyug-hyug-hyug-hyug-hyug-hyug-

but for computer software to see the similarity it has to recognise a *rule*, roughly of the type:

repeat a certain number of times (more than two and less than infinity)
a sequence of at least one single and optionally up to roughly seven or eight differing characters
followed by a hyphen or similar (where similar means in the following set such as [dot, slash, comma etc])

A device which trades on this poor computer pattern recognition is the *captcha* (Completely Automated Turing Test To Tell Computers and Humans Apart), a visual device designed to be easy for a human but hard for software to read, so as to be sure any response received came from a human and not from software (see Figure 11.1).

Unfortunately, in practice captchas can be hard for humans too! Recognising a scanned text in terms of its component letters is a similar problem. Optical Character Recognition (OCR) exists but is not yet 100 per cent reliable; it is more reliable in

Figure 11.1 Captcha (www.captcha.net).

English than in, say, French or Greek, as most work has been done on it in relation to English. Accordingly, there is not only the time for the scanner to look at the text (roughly the same time as a photocopier takes) but also OCR time and then a process of correcting mistakes in the text, probably with the help of a spell-checker. If you want text for corpus purposes, it is generally much better to spend time finding it already in electronic form than scanning it!

Learning

Computers, unlike babies, get switched off. They might store a lot of information in memory when switched on, but do not remember it unless they have been programmed to, as in spam detection, which attempts to identify and recognise certain words and phrases and use a growing stored list of them to check mail against. For Artificial Intelligence, they need some means of getting access to databases of information, procedures for sifting through it, and a means of storing inferences.

Graphics and speech

Animation, as in cartoons, video and speech reproduction or playback, is not too difficult, but does take up a lot of a PC's memory. Speech recognition, on the other hand, is still very much in its infancy and limited to a few hundred words.

Giving useful feedback on what is happening inside it

In 2002 Sinclair discussed the human–machine interface in terms of discourse analysis, showing how very rigid and limited it was. This is still very true: programming a computer to do a useful job and at the same time inform the user as to how it is progressing, what has been done and what remains to be done is still extremely difficult. Error messages are well known to be cryptic and get very widely ignored. Logging systems do exist but remain arcane for the majority of users. The hourglass and progress bar are in general use but the information they provide is very limited. It is as if the PC got tired and could not do two things at once, which as we have seen is not at all the case.

Things computers are sometimes expected or believed to be able to do

Sinclair (2001) talks disparagingly of 'tomorrow's Information Paradise', and 'the hype of New Age communications' and it isn't rare to see the sort of headline shown in Figure 11.2.

Science News

Chill Out, Your Computer Knows What's Best For You

ScienceDaily (June 21, 2008) — Computers are starting to become more human-centric, anticipating your needs and smoothly acting to meet them. The

Ads by Google

Video Conference Faciliti

Check out our Corporate Sol

Figure 11.2 Information Paradise?

In the *ScienceDaily* case above, the story concerned attempts to process data about where customers move about in stores and from that build up software that can aid in identifying suspicious behaviour, and in similar kinds of research such as face recognition software. Probably the cause of the optimistic headline was when the story moved on to a project to get software running a house to be able to predict its occupants' needs, presumably controlling heating and lighting as the occupants moved from room to room and suchlike. This is not a case where computers know best; it is merely a case of getting sensors to react when fairly simple rules apply (nobody in the room? turn the lights off!).

Let us examine a more corpus-related example and indeed with a much less hyped title: 'Predicting Human Brain Activity Associated with the Meanings of Nouns'. Mitchell *et al.* (2008) have programmed computers to predict human brain scans after subjects heard ordinary concrete nouns belonging to familiar categories such as *celery* (food) and *airplane* (vehicles), by processing large corpora (one trillion tokens) and determining their collocational configurations. That is, their computers generated pseudo brain scan images on the basis of word patterns after being trained with (a) lots of word data which might, for example, find that items like *airplane* and *helicopter* share similar lexical environments, and (b) genuine brain scans generated with magnetic resonance imaging equipment after subjects had been prompted with relevant words. This is an impressive feat, though it must be noted that the degree of matching the computer obtained was not at all high, even if it was statistically much greater than chance, and that one person's brain scan itself after hearing a word like *celery* or *airplane* is in any case not very like another's. Essentially the impressive feature of all these projects is the imagination and persistence of the programmers. To process an enormous corpus attempting to build up contextual profiles of certain nouns is quite impressive if this database can subsequently be used to generate predictions. In this case the predictions are concerned with the images generated by brain scans, but what matters in principle is that a complex database can be made to generate patterns and these patterns can then be matched up, sometimes, with others in the real world. A database of movements in a store might thus identify suspicious loitering. There is no magic. The computers which do so well at chess do it by using algorithms clever programmers have designed to take advantage of the things discussed above – that in a flash a computer can be made to evaluate millions of board positions.

2. Sorting out your data

Re-formatting and re-organising to match up better with what computers can manage

Because computers have no intelligence, we need to prepare our corpora carefully in advance so that the texts contain nothing unexpected and are in the right format for use. We have already seen that there is an issue of whether the underlying text format assumes that each written character is to be stored using one byte or more than one. What if some characters take up one byte but others take up two, three, four or more, though? In widespread current use, there is a format called UTF-8 which stores many characters as single bytes but some others as more than 1.

Figure 11.3 shows a tiny text prepared using Microsoft *Word 2007*, and then saved as Plain Text (.txt) using the standard defaults offered, then re-opened in MS *Notepad*.

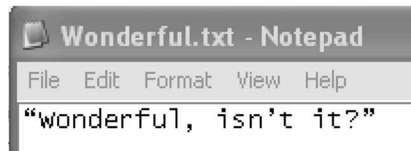

Figure 11.3 Plain text with curly quotes.

If you look carefully you will see that the quotation marks and the apostrophe are not symmetrical and that they are slightly curved. In the format which *Word* considers standard plain text, the underlying character codes are as shown in Figure 11.4 where the opening double-quote is coded 93 and the closing double-quote 94, 92 being used for the apostrophe.

If we now re-save the same text in a format called UTF-8 (Figure 11.5), and again open up the text file in *WordSmith's* File Viewer (Figure 11.6), what we find is that although the letters expressing wonderful are straightforward, one byte per character, there is now a 'signature' (EF-BB-BF) telling the computer that the whole file is in UTF-8 and then the curly double-quote at the beginning needs three byte-codes (E2–80–9C) and the double-quotes at the end similarly have E2–80–9D. The apostrophe has now grown to an E2-80–99 sequence.

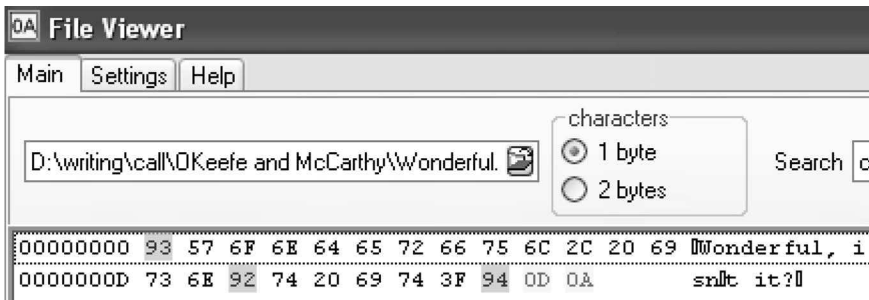

Figure 11.4 Plain text with different quote characters.

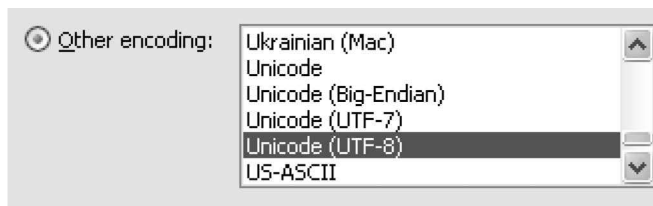

Figure 11.5 Saving as UTF-8 in *Word 2007*.

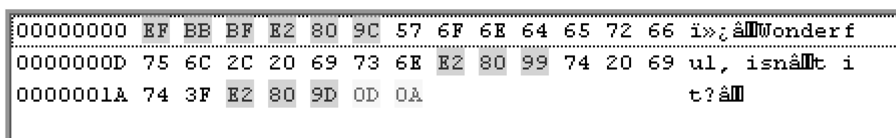

Figure 11.6 Plain UTF-8 text with some multi-byte characters.

This sort of low-level fact is normally kept hidden from users of the computer, of course. However, the fact that a rather weird system is in use can end up unexpectedly affecting corpus linguistic findings. I call the system weird, because everything would be so much more straightforward if it were possible to assume that a text file of 1,000 bytes contained exactly 1,000 characters, and that the 500th character was to be found by leaping forward 500 places from the start. In the case of the text above, the second character of the text, a W (represented by code 57), is in fact to be found after a three-byte signature and another three-byte character, at the seventh byte in the text stream.

The need for this strange kludge came about because on the one hand many users like to see curly quotes (even if sometimes the word processor curves them the wrong way!), and on the other hand disk and other memory space were for years at a premium, so operating systems and software didn't want to waste bytes unnecessarily. It is 'cheaper' to use one byte for most characters and then three for odd ones like curly quotes or m-dashes, rather than to use two bytes for every single character. And that way English text was stored economically (even if Japanese took up more bytes, and Cherokee or Hindi could not be represented at all …).

However, in the year 2008 and for some years prior to that, disk space and computer memory have been relatively cheap. At the time of writing it is possible in the UK to buy a 500 GB drive for a price equivalent to about seventeen large hamburgers. That is only the burgers, no chips or soft drinks to go with them. A 500 GB drive would hold a lot of text, which would take more than seventeen mealtimes to read: Charles Dickens' novel, *Bleak House*, the 800 pages of which when printed weigh a lot more than a big hamburger, is just under 2MB in size (at one byte per character) and that means more than 500 copies of *Bleak House* would fit into one GB. The 500 GB drive could therefore store 500 times 500 copies – you would not have shelf space for 250,000 copies of *Bleak House*, I suspect. A similar picture obtains for the computer's internal memory, its RAM. The cost of 2GB of RAM (best-selling laptops in late 2008 have either 2GB or 4GB) is approximately eight hamburgers.

Now that we are well into the new millennium, it would not be problematic to use a double-byte system; indeed, gradually, the world's computer services and operating systems have mostly moved that way. New documents are beginning to be routinely stored that way, too. There are several implications here. One is that it is quite likely that without the user thinking much about it, a coding difference can easily come between a left- and a right-quotation mark, or between an apostrophe and a single quotation mark. When corpus software comes to process them, because these forms as we have seen are encoded differently, there is a danger that what the user seeks is not found because of a trivial mismatch. This will typically occur when a corpus is put together containing texts formatted by different people following different conventions or using different word-processing software. There will be some texts using curly and others using straight quotes, some with differing n-dashes, m-dashes and hyphens, and others which have them all the same as each other.

Another implication is that corpus software may end up showing oddities. If software which does not handle UTF-8 is asked to process it, as is the case with the now obsolete *WordSmith 3.0*, anomalies may appear (Figure 11.7).

The 'word' *Â* is *WordSmith 3.0*'s capitalisation of *â*, which is character E2, and as we have seen above E2 is one of the components of a three-byte character. That is not the only anomaly – the 9C visible to the left of the 57 for the *W* of Wonderful has come out in the default font as Œ.

new wordlist (F)

N	Word	Freq.
1	Â	2
2	Ï	1
3	ISNÂ	1
4	IT	1
5	ŒWONDERFUL	1
6	T	1

Figure 11.7 UTF-8-inspired oddities in WS3 word list.

And a further implication is that corpus software has to be flexible to recognise and handle these various formats. *WordSmith 3.0*, which ceased development at the end of the last millennium, did not recognise or handle two-byte text at all, and as we have seen makes mistakes with some characters in UTF-8. As time goes by, it is very likely that these problems will fade away; old legacy texts will either be dropped completely or else converted to more modern standards, and the world will be one of Unicode.

But what exactly is Unicode? After all, MS *Word 2007*'s options for saving plain text visibly include four different Unicodes! Well, UTF-7 and UTF-8 are basically one-byte systems. UTF-7 is a one-byte-per-character system which won't handle any accented characters like *é* at all. What MS *Word 2007* calls 'Unicode' is in fact a system of two bytes per character, and 'Unicode (Big-Endian)' is the same but with the two bytes reversed. (This is like the tradition of first name, surname: the French and some other cultures in many contexts prefer the order DUPONT, Charles, while for many similar contexts the English prefer the first name to come first, as in Charles DUPONT. Big-Endian would represent the character 龍 [F9C4] with the F9 and C4 in reverse order as C4F9). Whether 'Unicode' or 'Unicode (Big-Endian)', *Word* will generally use two bytes for each character.

The Unicode Consortium, which controls these things and which is endorsed and supported by all the major software companies, announces at its site

Unicode provides a unique number for every character,
no matter what the platform,
no matter what the program,
no matter what the language.

(Unicode's slogan taken from
http://www.unicode.org/standard/WhatIsUnicode.html)

In other words, Unicode allows for the possibility of encoding any character for use in any language. A character is, roughly speaking, what we think of as a letter, although *A* is usually considered a different character from *a*. The actual shape as displayed will depend on the font which will represent it using the *glyph* designed for that font. For example *A* and **A** are different glyphs representing the same character.

Shapes are also needed, naturally, for all sorts of symbols that do not represent alphabetic or equivalent characters, such as dashes of varying lengths, copyright symbols, mathematical symbols for equations, and so forth. Many of these took up valuable space in the character sets of the 1960s to 1990s, elbowing out the Portuguese *ã*, for example.

Now in the discussion above, I distinguished between single-byte and double-byte systems, and implied that a double-byte system was good enough for MS *Word 2007*. In fact, even a double-byte system has limitations. As we saw above, a single-byte system can cope with a couple of hundred choices (256 to be precise), so how many can a double-byte system represent? The answer is 65,536 (from hex 0 to hex FFFF). Now with 65,000 character shapes a lot of languages can be handled, but it is also clear that 65,000 will not be enough for all the world's languages and scripts, living and dead – Chinese alone could require as many as 47,000. Accordingly, the Unicode consortium also allows for character encoding using up to four bytes per character. Using four bytes would allow for up to four billion different characters to be encoded.

In practice, we still live with the kludge described above. Even using a two-byte system, in some cases three or four pairs of bytes may be needed to represent specific characters (Unicode Technical Reports 17) and some combinations of characters are much more complex to handle than the description here can cover, as in cases analogous to Œ where two characters affect each other in some sort of special partnership.

However, I suspect that in a few more years we will have moved to a more rational simple system where all characters are stored using four-byte characters and there are no longer any limits to the character or other shapes that can be shown, and where it is certain that the thousandth character can immediately be found by leaping in exactly 4,000 bytes into a text file, in true random access.

You might wonder why I keep mentioning 'leaping into a text file' a given distance. In routine word-processing, e-mailing, etc., nobody ever does this. In corpus processing software, however, quite often one needs to open up a text file and start reading from a given spot, e.g. when showing the user the wider context of a concordance line.

The Unicode endeavour covers an enormous number of languages and their scripts. The Unicode CLDR Charts Languages and Scripts chart (2008) lists over 500 languages which are represented in varying scripts using Unicode. The consortium is still attempting to add more to the encoding. At the time of writing, the following scripts, whether ancient or modern, were still not perfectly represented: Meetei Mayek (Meitei Mayek, Manipuri, Kanglei), Miao (Pollard), Moso (Naxi), Mandaic, Mayan, Meroitic (Kush), Modi – and that is only the ones starting with the letter M. See the Unicode website for the current listing.

The problem is that text is all around us, but in slightly divergent formats. To the human reader these slight divergences pass unnoticed, but to software they can cause confusion and error. Analogously, when preparing a corpus of Shakespeare plays, the edition I was using abbreviated the character names before each speech so that the *to be or not to be* soliloquy originated in 'Ham.' To the human this is not a problem, which is why text editions used such abbreviations, but to the computer there would not be an easy linkage to *Hamlet* because computers are incapable of guessing. From all this, then, it follows that the user will need either to accept some slight 'noise' in the analysis, because of format inconsistencies, or else to try to put the basic corpus into a shape which will be handled appropriately by our corpus software. In practice, it is likely that some slight noise will always be present. The very existence of some of *WordSmith Tools*' utilities, such as its Text Converter and File Viewer, has arisen from the need to remove inconsistencies. The File Viewer was accordingly designed to look deep into the innards of a text file so as to see exactly how it is encoded; the Text Converter (www.lexically.net/downloads/version5/HTML/?convert_text_file_format.htm) was designed to be able to convert to Windows, for example, from formats used in Unix or Macintosh systems,

which differ slightly from those used in Windows chiefly in regard to paragraph and line end formats producing plain Unicode text, in order to standardise a mixed corpus, or to re-structure the files using a more intuitive folder system, particularly necessary for the BNC with its uninformative filenames.

3. Concordances

Once the corpus has been put together and cleaned up, one is in a position to generate concordances. Tribble (this volume) explains and illustrates concordances themselves in depth, so here only a few basic observations on how a concordance is generated will be made (see also Evison, this volume).

There are two basic methods: the 'on the fly' system and an index-based one. If working 'on the fly', the concordancer reads each text file in order and searches for the desired string(s) in the long stream of bytes which it has just read in. If it finds the desired string, it then checks to see whether any other contextual requirements are met, such as whether the string is bounded appropriately to left and right, for example by a space or a tab or a punctuation symbol. It may also search the environment for another string that the user has specified must be present in the environment. If these tests are passed, some part of the context will be stored in the computer's memory or in a disk file, along with information about the filename, the place within the file where the hit was found, and so on. A set of such pieces of information is stored for each successful 'hit'.

Alternatively, the entire corpus is first processed to build up an index which gives a set of pointers. This processing usually involves tokenising where each word of the source text is represented by a number, and a sizeable database is created which 'knows' (i.e. holds pointers to) all the instances of all the words. Then when the search starts, the database is asked for all instances of the word being sought, and additional requirements such as context words can be put to the database too. Once the request has been met, again the program builds up a set of structured pieces of information about the context, the filename, the exact location of each word in the context, and so forth.

With such a set of records, it is then possible to sort the output in a desired way, and present it to the user using appropriate colours. The KWIC (key word in context) context is usually, though not always, presented so that the search-word or phrase appears centred both horizontally and vertically in the output, and often with a colour or typeface marking to help it stand out visually.

4. Word lists

Word lists are usually created 'on the fly' as described above. A 'current word' variable is first allocated in the computer's memory and set to be empty. Then, as the stream of bytes is processed, every time a character is encountered that is alphanumeric, it's added to that 'current word'. If on the other hand the character turns out to be non-alphanumeric, such as a comma or a tab or a space, the program assumes that an end-of-word may have been reached. It may have to check this in certain cases such as the apostrophe in *mother's*, where a setting may allow for that character not to count as an end-of-word character. To do so, it might have to be able to handle not only straight apostrophes but also curly ones, as we have seen. Whenever the program determines that the end of the

'current word' has been reached, it stores that word much as, when concordancing, it clears the 'current word' ready for new data, and proceeds in the stream of bytes.

As the word gets stored, a search process will be needed, to find out whether that same word token has already been encountered. If it has, then a frequency counter will need to be incremented. It may be that the program will also check whether that word-type has already been found in the text file which is currently being handled, and if necessary increment a counter recording how many times it occurs in each text.

At some point the set of words stored needs to be sorted and displayed, possibly alphabetically and by frequency order, and maybe with some accompanying statistical information such as type-token or standardised (Scott 2008) type-token ratios, which report on how the different word-forms (types) relate to the numbers of running words (tokens) in the text or corpus.

Word lists by themselves are often best seen as a first approach to a corpus. It is by processing the words and looking at the most frequent of them that one can get a rough idea of the kinds of topics being explored, the wealth of vocabulary being used – as well as any formatting anomalies as shown above! It may be possible to select word(s) in a word list and have the corpus software generate a concordance of that/those words in the very same text files.

Finally, it may be possible to get software to show word lists of n-grams where pairs or triples of words are shown in their text clusters or bundles (Biber and Conrad 1999), by identifying any repeated consecutive sequences of words of the length desired (see Greaves and Warren, this volume).

Figure 11.8 shows a word list based on three-word clusters in Shakespeare plays. It is clear that *I pray you* is not only a frequent n-gram but it is also widespread since it occurs in thirty-four (out of thirty-seven) plays.

index: shakespeare

File Edit View Compute Settings Window Help

N	Word	Freq.	%	Texts	%
1	I PRAY YOU	250	0.03	34	91.89
2	I WILL NOT	214	0.03	36	97.30
3	I KNOW NOT	162	0.02	36	97.30
4	I DO NOT	160	0.02	33	89.19
5	I AM A	141	0.02	35	94.59
6	I AM NOT	139	0.02	34	91.89
7	MY GOOD LORD	132	0.02	29	78.38
8	AND I WILL	129	0.02	34	91.89
9	I WOULD NOT	126	0.02	34	91.89

frequency alphabetical statistics filenames notes

8,553 Type-in

Figure 11.8 Three-word-cluster word list.

5. Key word lists

A key word, as identified in *WordSmith Tools*, is a word (or word cluster) which is found to occur with unusual frequency in a given text or set of texts. As such it may be found to occur much more frequently than would otherwise be expected or much less frequently (a negative key word). In order to know what is expected, a reference of some sort must be used. In *WordSmith*'s implementation, key word lists use as their starting points word lists created as described above. One word list is first made of the text or set of texts which one is interested in studying, and another one is then made of some suitable reference corpus. This may be a superset of the text-type including the one(s) being studied, such as when one uses a word list based on the *Daily Mirror*'s news texts 2004–7 as a reference when studying one specific *Daily Mirror* text. Or it might be a general-purpose corpus like the British National Corpus (BNC) used as a reference corpus when one is studying the specific *Daily Mirror* text.

When the specific word list and the reference corpus word list are ready, the key word software simply goes through the specific word list and checks each word (or cluster) in it with the corresponding word or cluster in the reference corpus word list. In most cases these comparisons do not reveal much difference in percentage frequency. For example, THE may be roughly equally frequent as a percentage of a given *Daily Mirror* text as it is of a large set of *Daily Mirror* texts, say around 5 per cent in both cases. Similarly, it is likely that I PRAY YOU is about equally frequent in *Othello*, say, or *As You Like It* as it is in all the Shakespeare plays where, as we see above, its frequency is 0.03 per cent of the running words of the plays. Some of the strings, though, will be found to be unusually high or low. *Unusually* is determined using statistical procedures, generally Dunning's (1993) Log Likelihood procedure. As with the concordancing and word listing procedures described above, once an item has been found to meet the criterion, the program will store it; eventually it will sort the stored forms, e.g. in alphabetical order or keyness order, and display them.

The screenshot in Figure 11.9 shows the key word clusters of the play *Hamlet*, compared with the clusters of all the Shakespeare plays (including *Hamlet*) as identified in the word list above.

KeyWords

File Edit View Compute Settings Window Help

N	Key word	Freq.	%	. Freq.	RC. %	Keyness	P	emm:
1	AY MY LORD	9	0.03	20		25.74	0.0000003887	
2	MY LORD I	17	0.06	108	0.01	22.10	0.0000025882	
3	GOOD MY LORD	15	0.05	90	0.01	20.71	0.0000053490	
4	TO A NUNNERY	5	0.02	5		19.95	0.0000079332	
5	MY LORD I HAVE	6	0.02	14		16.72	0.0000433459	
6	I THE EARTH	5	0.02	8		16.71	0.0000435916	
7	I MY LORD	6	0.02	18		14.45	0.0001440618	
8	LORD I HAVE	6	0.02	18		14.45	0.0001440618	
9	WELL MY LORD	6	0.02	22		12.63	0.0003794433	

KWs plot links clusters filenames notes source text

9 Type-in

Figure 11.9 KW clusters in *Hamlet*.

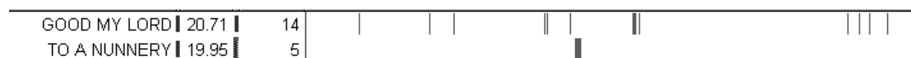

Figure 11.10 KW plot of clusters in *Hamlet*.

Most of them include 'my Lord', a speech formality which suggests that relationships or argumentation may be especially important here. The cluster GOOD MY LORD occurs with a frequency of 0.05 per cent of the running words in the play, which contrasts with the 0.01 per cent overall in the Shakespeare plays as a whole, and this is why the algorithm has found it to be key. AY MY LORD, on the other hand, though it is slightly *less* frequent, is found to be more striking in its frequency in *Hamlet*; its frequency in the whole set of plays is below 0.01 and hence not shown.

A plot of the locations of some of these clusters (Figure 11.10) is akin to a map of the play, with the left edge of the plot representing the beginning and the right edge the end of the pay. TO A NUNNERY is quite sharply located in one specific burst in Act III Scene 1, while GOOD MY LORD is more globally located.

The most appropriate interpretation of these key words or key word clusters is as pointers; they suggest some statistical anomaly which itself merits further research. There is probably a pattern underlying some of the clustered strings in *Hamlet* but as yet we are not close enough to it. It is definitely necessary to concordance these clusters, for example, to find out who keeps addressing whom as 'my lord'.

The researcher's aim is really to find such patterns, but as we have seen here, to do that effectively it is often necessary to look 'under the hood' to see how corpus software does some of its work, and to see how texts it may process are structured and formatted.

Further reading

Hunston, S. (2002) *Corpora in Applied Linguistics*. Cambridge: Cambridge University Press. (This provides a solid introduction to many aspects of Corpus Linguistics.)

Scott, M. (2008) 'Developing *WordSmith'*, in M. Scott, P. Pérez-Paredes and P. Sánchez-Hernández (eds) 'Monograph: Software-aided Analysis of Language', special issue of *International Journal of English Studies* 8(1): 153–72. (This paper discusses principles of software development.)

Scott, M. and Tribble, C. (2006) *Textual Patterns: Keyword and Corpus Analysis in Language Education*. Amsterdam: John Benjamins. (This develops many of the above ideas in book form.)

References

Adams, D. (1978) *The Hitch-Hiker's Guide to the Galaxy*. BBC 4 radio programme (book published 1979, London: Pan Books).

Barry, P. J., Gallagher, P., Ryan, C. and O'Mahony, D. (2007) 'START (Screening Tool to Alert Doctors to the Right Treatment) – An Evidence-based Screening Tool to Detect Prescribing Omissions in Elderly Patients', *Age and Ageing* 36(6): 632–8.

Biber, D. and Conrad, S. (1999) 'Lexical Bundles in Conversation and Academic Prose', in H. Hasselgard and S. Oksefjell (eds) *Out of Corpora: Studies in Honor of Stig Johansson*. Amsterdam: Rodopi, pp. 181–9.

Cohen, B. (1998) 'Howard Aiken on the Number of Computers Needed for the Nation', *IEEE Annals of the History of Computing* 20(3): 27–32.

Dunning, T. (1993) 'Accurate Methods for the Statistics of Surprise and Coincidence', *Computational Linguistics* 19(1): 61–74.

Mitchell, T. M., Shinkareva, S. V., Carlson, A., Chang, K.-M., Malave, V. L., Mason, R. A., Just, M. A. (2008) 'Predicting Human Brain Activity Associated with the Meanings of Nouns', *Science* 320(5880): 1191–5.

Scott, M. (2008) *WordSmith Tools* online help, available at www.lexically.net/downloads/version5/HTML/?type_token_ratio_proc.htm

——(2009) *WordSmith Tools*. Liverpool: Lexical Analysis Software Ltd.

Sinclair, J. (2001) 'The Deification of Information', in M. Scott and G. Thompson (eds) *Patterns of Text: In Honour of Michael Hoey*. Amsterdam: John Benjamins, pp. 287–314.

Unicode Technical Reports 17, available at www.unicode.org/reports/tr17/ (accessed October 2008).

Unicode CLDR Charts Languages and Scripts, 25 July 2008. available at www.unicode.org/cldr/data/charts/supplemental/languages_and_scripts.html (accessed October 2008).

Wolfram MathWorld, available at http://mathworld.wolfram.com/RoundoffError.html (accessed 10 Oct 2008).