Encoding Text

Christopher Vickery
Queens College, CUNY

Shannon's information theory tells us how many bits we need to represent the different elements in a set, $\log_2(n)$, where n is the number of elements in the set. For example, to represent the 26 letters of the English alphabet would require 4.7 bits. That is, to find out which letter of the alphabet I'm thinking of, you would have to ask me an *average* of 4.7 yes-no questions. In practice, you could get the answer in exactly either four or five questions, but if we played this guessing game a lot of times, the average of all the 4's and 5's would get close to 4.7 provided my letters and your guesses are random.

Although information theory tells how many bits of information are inherent in the alphabet, it doesn't address the question of how to use bits to send letters over a communication channel. To do that, we need to decide how to *encode* the letters. That is, we need a way to assign binary numbers to letters in a way so that both the sender at one end of the communication channel and the receiver at the other end of the channel know how to interpret the bits being sent. The sender must first *encode* the letters by assigning binary values to each one, and the receiver must *decode* the binary values it receives by translating them back into the letters.

Each mapping between a letter and its binary number is called a *code point*, and all the code points being used are called a *code table.* A code table has one row for each code point. So to transmit letters we need to have a code table with 26 code points. Taken together, the 26 code points would simply be called a "code."

One big difference between information theory and communication codes is that there is no mechanism for transmitting fractional bits over a communication channel: it's all ones or zeros, with nothing in between. So that theoretical 4.7 bits per letter has to be rounded up to the next larger integer: five. Since 2^5 is 32, our alphabet code will have 32 code points, even though we need only 26. The extra 6 code points could either go unused, or could be used for special characters, like *space* to separate groups of letters into words.

Youcouldsendwordswithoutspacesbetweenthembutitwouldbehardtoread!

The extra code points could also be used for basic punctuation: period, comma, question mark, exclamation point, apostrophe … oops, no room for semicolons, dollar signs, parentheses, etc.

The code points can be arbitrary pairings of letters and codes (such as 'A' = 10110, 'B' = 11001, …) or orderly (such as 'A' = 00001, 'B' = 00010, …). For effective communication it doesn't matter how the code points are ordered in the code table so long as the sender and receiver agree to use the same pairings.

For now we are going to work with codes that use the same number of bits for each code point. As an aside, that's not necessarily the case. For example, some letters of the alphabet are used more frequently than others ('E' is the most common letter, and 'Z' is the least common letter), so the sender and receiver could agree on a code that uses fewer bits for E's than for Z's in order to be more efficient. But first things first: we will use a fixed number of bits per code point.

The *Baudot Code* was an actual code with 5-bit code values. A French man named Jean-Maurice-Émile Baudot invented it in 1870 for telegraph communication. An important feature of the code was to use two of the "extra" code points to signal which of two codes tables was being used. One code table was for letters (including both E and É, perhaps because of the É in Baudot's name!), and the other table was for digits and punctuation symbols. Once the sender transmitted the special code value that meant, "Use the characters table," all code values after that were letters. But if the sender transmitted the special code value that meant, "Use the figures table," all code values after that were digits or punctuation marks until the next "Use the characters table" code was sent. This trick almost doubled the number of characters that could be represented from 32 (2^5) to 62 for this five-bit code. (62, not 64, because two code points were used for the "switch tables" codes.) You can see vestiges of this trick on modern keyboards: the Esc ("escape") key was originally intended as a way to "escape" from one code table into another one, although it is no longer used that way. But the analogy often remains: when you use the Esc key to exit full-screen mode when watching a video, for example, you are "escaping" from one video mode to another, somewhat like escaping from one code table to another in Baudot.

What follows is a chronological account of the evolution of character codes over years. Each new code listed here was designed to deal with a shortcoming of the codes that came before it. It's taken several iterations to get to where we are today. Each of the codes listed here is still in active use today, with some deprecated alternatives mentioned in passing. ("Deprecated" is a popular word in computing, used to mean something that once seemed like a good idea, but that now should be avoided where possible because something better is now available that should be used instead.)

ASCII

The American Code for Information Interchange is a 7-bit code that overcame the inability of the Baudot code to represent both uppercase and lowercase letters. With $2^7 = 128$ code points, the code can handle the alphabet, digits, and almost three dozen punctuation symbols, with an additional 32 code points for "control" codes. This code remains the basis for keyboards used in the US (note that the name of the code refers to it as an "American" code). The code was adopted as a standard by the American Standards Association (now the American National Standards Institute) in 1963, the same year the Teletype Corporation of Skokie, Illinois introduced their

Model 33 teleprinters that used ASCII for exchanging printed information: what was typed on one teleprinter was printed on another one, connected to it over a telephone line. Before the Model 33, teleprinters used a Baudot code to exchange information.

> Sending bits over telephone wires isn't as simple as it might seem. Because the telephone system is designed to transmit analog voltages representing sound (speech, in particular), it does a poor job of maintaining the ideal square-shaped waveform of a sequence of ones and zeros. The solution, until digital phone lines came into use, was to use two different tones to represent ones and zeros. Because the bits are transmitted so rapidly we can't actually hear the two tones: it just sounds like static.

Several features of ASCII are closely related to the design of the Teletype machines of the day. In fact 32 of the available 128 control points were reserved for operations that would control the electromechanical printing mechanism or the communication protocols of the day: Carriage Return (CR) returned the printing mechanism to the left margin of the paper; Linefeed (LF) advanced the paper to the next line; Tab advanced the printing mechanism to the next tab stop position on a line; Bel rang the mechanical bell that was normally used on typewriters to signal the approaching end of a line, etc.

> When ASCII was adopted for storing information in computer memories, the fact that both CR and LF were needed to mark the end of a line of text presented an opportunity to save a few (then) precious bytes of storage. The developers of the Unix operating system adopted the convention of storing only a LF character at the end of lines of text. When the text was displayed or printed, the necessary CR would be generated automatically. At the same time, Apple Computer adopted the convention of storing only a CR at the end of each line, and software automatically generated the LF to go with it when needed. Meantime, Microsoft adopted the final possible configuration for line endings by storing both the CR and LF. Remnants of this lack of uniformity still show up occasionally when text files are copied from one type of system to another.

> Another issue that showed up with the use of ASCII for storing text in computers is the fact that it is more efficient to design computer memories with the smallest "chunk" of memory having a number of bits equal to a power of 2. The now-universal 8-bit size of a byte as the smallest chunk of memory means that each byte can hold one ASCII code value, with an extra bit left over. ASCII characters are always stored in the rightmost seven bits of a byte, with the leftmost bit set to 0.

ISO-Latin-1

As the word "American" in the name of ASCII suggests, it was designed without consideration of alphabets that include other letters than the standard 26 used in this country. It can't even represent the É in Monsieur Baudot's name.

Given the common standard of 8-bit bytes, there were two main efforts to define new codes with 2^8 = 256 code points. Microsoft developed one set of codes, which changed ASCII in two ways: one way was to introduce a set of printable symbols that replaced some of the 32 control codes that were seldom used, and the other way was to introduce a new set of 128 characters that had their leftmost bit equal to 1. Since even 128 additional letters couldn't handle all the languages of the world, Microsoft also introduced the notion of "code pages," which were alternate sets of 128 letters to use when the leftmost bit of a byte is 1. There is a code page for the Greek Alphabet, another one for Scandinavian characters, another for Eastern European characters, etc.

About the same time, the International Standards Organization (ISO) developed its own 8-bit extension to ASCII. Unlike Microsoft's code pages, the ISO codes maintained all 32 ASCII control codes. Beyond that, the concept is the same: different ISO codes provided 128 code points for various alphabets. One code in particular is important for our story: ISO-Latin-1 (also known as ISO-8859-1) covers the special characters of most of the "Western" languages that are based on the Latin alphabet.

ISO-Latin-1 and Microsoft's code page 1252 are very similar, but not the same. Web sites that show "funny" characters in place of quotation marks and certain other symbols typically do so because they do not handle the differences between the two codes correctly.

Unicode

The need for multiple code sets to cover different languages plus the incompatibility between the ISO codes and Microsoft's led to development of a 32-bit "universal" code: Unicode. With 2^{32} = *4 billion* code points, Unicode can represent all characters in all known alphabets, past or present, with room left over for the symbols used in dance notation, mathematics, emoticons, and more.

Unicode is based on ISO-Latin-1, which in turn is based on ASCII. Here is a diagram to help explain the structure:

UUUUUUUU UUUUUUUU UUUUUUUU LAAAAAAA

Each letter represents a bit. If all the U bits are zeros, the remaining eight bits (the L bit and the A bits) are an ISO-Latin-1 code point. Within the ISO-Latin-1 code points, if the L bit is a zero, the character is an ASCII code point.

The only problem with Unicode is that it requires four bytes (32 bits) to store each character. When storing common information that consists primarily of text that could be stored using single bytes (ISO-Latin-1 code points), three extra bytes of all zeros have to be stored. To overcome this inefficiency, there are two Unicode variants: UTF-16 uses two bytes to store the most-common 2^{16} = 65, 636 characters, and UTF-8 uses one byte to store ISO-Latin-1 code points. Both UTF-8 and UTF-16 use a version of Baudot's idea of escape codes to include special code points that allow any of the other 4 billion Unicode characters to be inserted into a stream of UTF-16 or UTF-8 characters.

Text versus Renderings

An important concept to understand is that character codes say nothing about the shapes and other visual properties of the characters they represent. The visual properties of text fall are the subject of *typography.* As the *–graphy* part of that word suggests, characters are represented by drawings of the letter shapes, and the layout of those letter shapes on the paper or screen. *Rendering* is the term used to refer to drawing text on a visual medum.

Here are some visual properties of text, drawn primarily from the CSS rules used for web page design:

- Margins, padding, leading (line spacing), character spacing
- Font family
- Font size
- Font weight
- Font variants
- Text decoration
  - Underline
  - Strikethrough
  - Overline