Document and Content Analysis

Lecture 04 – Document Storage and Display

Faisal Shafait

12.05.2011

Motivation

Which document file formats are you familiar with?

Motivation

Which document file formats are you familiar with?
“.txt, .doc, .odt, .docx, .pdf”

Motivation

Which document file formats are you familiar with?
“.txt, .doc, .odt, .docx, .pdf”

Have you seen any documents in a compressed format?

Motivation

Which document file formats are you familiar with?
“.txt, .doc, .odt, .docx, .pdf”

Have you seen any documents in a compressed format?
“.zip”

Motivation

Which document file formats are you familiar with?
“.txt, .doc, .odt, .docx, .pdf”

Have you seen any documents in a compressed format?
“.zip”

Do you know how compressing a document works?

Motivation

Which document file formats are you familiar with?
“.txt, .doc, .odt, .docx, .pdf”

Have you seen any documents in a compressed format?
“.zip”

Do you know how compressing a document works?
You should know after this lecture

Data vs. Information

The same information can be encoded as data in many different forms, requiring different numbers of bits, e.g.

0080, 0090, 0070, 0070, 0050, 0010, 0090, 0070, 0080, 0090

- ASCII sequence

  ”0080,0090,0070,0070,0050,0010,0090,0070,0080,0090”

  ⇒ 49 bytes, 392 bits

Data vs. Information

The same information can be encoded as data in many different forms, requiring different numbers of bits, e.g.

0080, 0090, 0070, 0070, 0050, 0010, 0090, 0070, 0080, 0090

- ASCII sequence

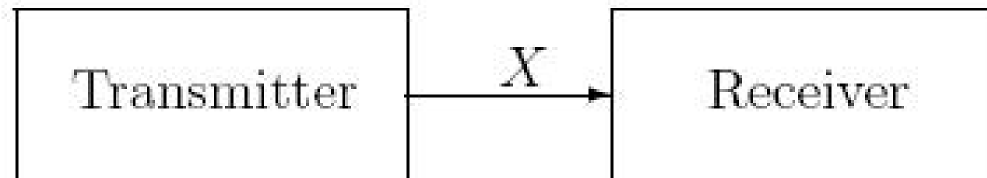
  ”0080,0090,0070,0070,0050,0010,0090,0070,0080,0090”

  ⇒ 49 bytes, 392 bits
- 32-bit integers: 80,90,70,70,50,10,90,70,80,90

  ⇒ 10 · 4 bytes, 40 bytes, 320 bits
- ASCII: ”8,9,7,7,5,1,9,7,8,9” ⇒ 19 bytes, 152 bits
- ASCII: ”8977519789” ⇒ 10 bytes, 80 bits
- ASCII-hex: 2171a14ad ⇒ 9 bytes, 72 bits
- binary: one 34 bit integer value ⇒ 34 bits

Compression as Exchange of Information

•compression as an encoding/decoding problem:



•each side has a codebook to code or decode the message X
•codes can be bit-strings of arbitrary length
•the codes should be uniquely decodable by looking at one bit at a time (prefix-codes)

Information and Coding Theory

- consider the message X of length n as a random process with i.i.d. (independent, identically distributed) events
- the probability for an event *i* is *p(i)*.
- the **information entropy** of a message channel is

$$H := -\sum_i p(i) \log_2 p(i)$$

- entropy gives a lower bound on how many bits are needed to encode a message in the channel on the average.

Entropy as a measure of Information: Example

The message is 0, 1, 0, 2.

- symbols are 0, 1 and 2
- we assume $p(0) = 0.5$, $p(1) = p(2) = 0.25$.
- naive codebook: 0 = **00**, 1 = **01**, 2 = **10**
- The code is **00010010**, i.e. 8 bits, or 2 bits per symbol
- entropy:

$$H = -0.5 \cdot \log2\,(0.5) - 0.25 \cdot \log2\,(0.25) - 0.25 \cdot \log2\,(0.25)$$

$$= 0.5 \cdot 1 + 0.25 \cdot 2 + 0.25 \cdot 2 = 1.5$$

- A better codebook would be: 0 = **0**, 1 = **10**, 2 = **11**
- The total code is **010011**, requiring 6 bits, i.e. we reach the theoretical bound of 1.5 bits per symbol.

Entropy as a measure of Information: Examples

The message 0, 1, 2, 3, 4, 5, 6, . . . , 255

- p(i) = $\frac{1}{256}$, for i = 1, . . . , 256.
- Naive coding takes 8 bits per symbol.
- $H = -\sum_{i=0}^{255} \frac{1}{256} \log(\frac{1}{256}) = 8$
- So we cannot do better than naive 8-bit.

The message 0, 0, 0, 0, ...

- There is only one symbol, 0. p(0) = 1
- H=0
- We don’t have to send anything. We agree on a codebook
- where ”nothing” is mapped to ”0”.
- problem ?

Variable Length Coding (VLC)

- The codes we send do not have to be of identical length:
- original: 128 128 128 128 128 129 129 129 129 130 131 132
- usual 8-bit representation: $12 \cdot 8 = 96$ bits
- Entropy:
  - $p(0, \ldots, 127) = 0$
  - $p(128) = \frac{5}{12} = 0.416$
  - $p(129) = \frac{4}{12} = 0.333$
  - $p(130) = \frac{1}{12} \approx 0.083$
  - $p(131) = \frac{1}{12} \approx 0.083$
  - $p(132) = \frac{1}{12} \approx 0.083$
  - $p(133, \ldots, 255) = 0$

$$
\begin{aligned}
H &= -\frac{5}{12} \cdot \log_2\left(\frac{5}{12}\right) - \frac{1}{3} \cdot \log_2\left(\frac{1}{3}\right) - 3 \cdot \frac{1}{12} \cdot \log_2\left(\frac{1}{12}\right) \\
&\approx 1.95
\end{aligned}
$$

Utilizing if fewer than 256 values occur

- original: 128 128 128 128 128 129 129 129 129 130 131 132
- We have 5 distinct symbols, so we can use a 3 bit code:
  - 128 ⇒ 000
  - 129 ⇒ 001
  - 130 ⇒ 010
  - 131 ⇒ 011
  - 132 ⇒ 100
- 3 bits per symbol means $12 \cdot 3 = 36$ bits in total

Variable Length Coding (VLC) II

- In the previous example, 132 is uniquely identified after 1 bit:
  - 128 ⇒ 000
  - 129 ⇒ 001
  - 130 ⇒ 010
  - 131 ⇒ 011
  - 132 ⇒ 1
- The message becomes
- 3 + 3 + 3 + 3 + 3 + 3 + 3 + 3 + 3 + 3 + 3 + 1 = 34 bits
- Expected rate $\frac{5}{12} \cdot 3 + \frac{4}{12} \cdot 3 + \frac{1}{12} 3 + \frac{1}{12} 3 + \frac{1}{12} \cdot 1 = 2.834$ bits per symbol

Variable Length Coding (VLC) III

- In general, it is better to code frequent symbols with short codes.
- Since 128 is more frequent than 132, it is better to use the single 1 for 128 and the longer codes for the less frequent values
- 128 ⇒ 1
- 129 ⇒ 000
- 130 ⇒ 001
- 131 ⇒ 010
- 132 ⇒ 011
- Expected rate: $\frac{5}{12} \cdot 1 + \frac{4}{12} \cdot 3 + \frac{1}{12} 3 + \frac{1}{12} 3 + \frac{1}{12} \cdot 3 = 2.168$
- Signal length:

  1 + 1 + 1 + 1 + 1 + 3 + 3 + 3 + 3 + 3 + 3 + 3 = 26 bits

Huffman Codes

- Huffman codes can be constructed for any given distribution
- $128 \Rightarrow 1$
- $129 \Rightarrow 01$
- $130 \Rightarrow 001$
- $131 \Rightarrow 0000$
- $132 \Rightarrow 0001$
- Expected rate: $\frac{5}{12} \cdot 1 + \frac{4}{12} \cdot 2 + \frac{1}{12} 3 + \frac{1}{12} 4 + \frac{1}{12} \cdot 4 = 2.0$
- Encoded signal: 111110101010100100000001 has 24 bits
- Lower bound from coding theorem is $12 \cdot 1.95 = 23.4$

  $\Rightarrow$ 24 is optimal.

Computing Huffman Codes

- compute probabilities of all symbols
- arrange symbols in descending order of probability
- construct a tree by combining two least probability nodes
- assign either 1 or 0 to each branch
- traverse the constructed tree from root node to any leaf to find the code for the leaf

Huffman Codes

- Huffman codes are optimal in the sense that no other code that maps symbols onto fixed bit-strings can create a shorter sequence.
- It doesn't mean that they perform closely to the entropy bound, e.g. for a binary signal $p(0) = 0.9999$, $p(1) = 0.0001$: $H \approx 0.0015$, but we need at least 1 bit per symbol.
- If we drop the requirement of mapping symbols onto fixed bit-strings, we can do better → arithmetic encoding

Signal Compression by Entropy Coding

- Storing information (text, images, sound) can be performed more efficiently by using a good code, based on the entropy.
- We have achieved a compression ratio of $\frac{96}{24} = 4$ compared to raw 8-bit integers, just by using different symbols.
- Many compression utilities, e.g. WinZIP, rely on entropy codings, often by constructing Huffman codes.
- Huffman compression achieves rates of approx. 2 to 3 on English text, and 1 to 4 on images.
- Huffman coding doesn’t have a fixed codebook. It depends on the data.
- In practice, the codebook and the length of the signal have to

  be transmitted together with the data!

Run-Length Encoding (RLE)

In many cases, neighboring data points are not statistically independent, but are strongly correlated (consider sending a fax). If the same values occur frequently next to each other, we can use Run-Length-Encoding :

- original: 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 1 1 0 0 0 0 0 ⇒ 23 bit
- $p(0) = 16/23 \quad p(1) = 7/23 \quad H \sim 0.88$
- Run+Value: 6 × 0, 5 × 1, 5 × 0, 2 × 1, 5 × 0
- values alternate in fixed manner, only store run: 6 5 5 2 5
- naive 3-bit code: 15 bits

The result can be entropy-compressed:

- How ?
- Huffman code: ?
- expected rate: ?
- compressed message: ? , how many bits ?

Intermediate Summary

- Information is not the same as data. The same information can be expressed (**coded**) using more or less data.
- If we code a signal, the **entropy** of the coding alphabet is a **lower bound** on how many bits we will need per symbol (Shannon's Source Coding Theorem).
- The lower the entropy, the more efficient we can code.
- The more ”peaked” the distribution, the lower its entropy.
- For each source, we can build a **Huffman-Code**. This is **optimal** (when...?).
- The Huffman code usually does not reach the entropy bound, but it is guaranteed to stay within 1 bit of it.
- Since entropy and coding efficiency depend on the **coding alphabet** distribution, we can achieve better compression by first transforming a signal, e.g. by RLE.

Displaying Documents
(Fonts and Font Rendering)

What is a Font?

What is a Font?

•a complete character set of a **single** size and style of a particular **typeface**

•A letter or symbol in a font is called a **glyph**

•typeface designates a consistent visual appearance or style which can be a "family" or related set of fonts

How is font size measured?

How is font size measured?

- Font size is measured in “points”
- 1pt = 1/72 inch when printed on paper
- What about size on screen?

What are typical font families?

What are typical font families?

- Serif vs. Sans Serif
- Proportional spacing vs. monospace

```
for(qab=0; qab<111; qab++)
{
    for(ijl=0; ijl<100; ijl++)
    {
        qab = ijl + ijl * 3
        ijl = ijl + 3
    }
}
```

```
for(qab=0; qab<111; qab++)
{
    for(ijl=0; ijl<100; ijl++)
    {
        qab = ijl + ijl * 3
        ijl = ijl + 3
    }
}
```

How to represent a font?

How to represent a font?

- Bitmap fonts
- Vector fonts
- Stroke-based fonts

Bitmap fonts

- Stores each glyph as an array of pixels
- Simply collections of raster images of glyphs constitute a font
- For each variant, a complete set of glyph images is required
- Consider 10 sizes and any combination of bold and italic, how many complete sets of glyphs would be needed?

Advantages of Bitmap Fonts

- Extremely fast and simple to render
- Easier to create than other kinds
- Unscaled bitmap fonts always give exactly the same output

Disadvantages of Bitmap Fonts

- A new set of glyph images for each size
- Scale poorly to other sizes
- Memory intensive

Vector Fonts

- Outline of each glyph is represented as a set of lines and curves
- Adobe PostScript is a standard example
  - Postscript type 1 and type 3
  - TrueType
  - OpenType

Postscript Fonts

•Postscript created by Adobe Systems in 1982
•outline font specifications using PostScript file format to encode font information
•the glyphs are described with cubic Bézier curves
•a single set of glyphs can be resized through simple mathematical transformations

TrueType Fonts

- Outline font standard developed by Apple Computer in the late 1980s
- Offers font developers a high degree of control over precisely how their fonts are displayed
- Microsoft added TrueType into the Windows 3.1 operating system
  - Times New Roman
  - Arial
  - Courier New
- A regular TrueType font comes with .ttf extension

OpenType Fonts

- Developed by Microsoft and Adobe Systems
- Intended to supersede Postscript Fonts and TrueType Fonts
- The font character encoding is based on Unicode and can support any script
- OpenType fonts can have up to 65,536 glyphs
- Fonts can have advanced typographic features that allow proper typographic treatment of complex scripts
- Font files are intended to be cross-platform

Vector Fonts

Advantages of Vector Fonts:

- Easily transformed by applying a mathematical function to each vector point
- Can be scaled to any size without compromising visual quality

Disadvantages of Vector Fonts:

- Rendering is slow and compute intensive
- Rendering can change shape depending on desired size and position

Stroke-Based Fonts

- A glyph's outline is defined by the vertices of individual strokes and stroke's profile.

Advantages:

- Reduces number of vertices needed to define a glyph
- Editing a font by stroke is easier than editing by outline

How do we render these fonts on screen?

Font rendering is the process of converting text from a vector description (as found in scalable fonts such as TrueType fonts) to a raster or bitmap description

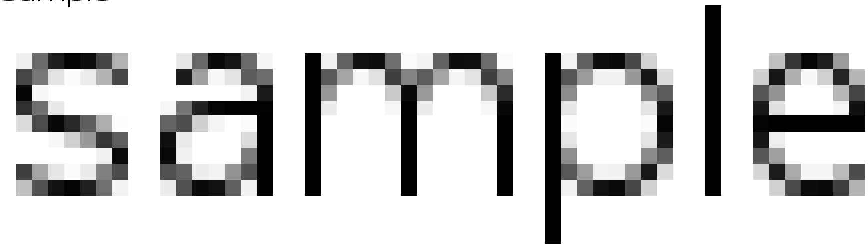
Simple Font Rendering

sample

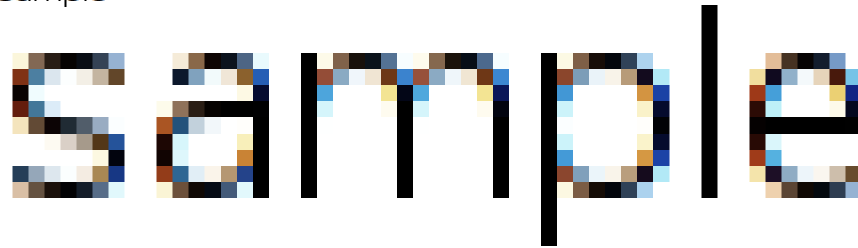
sample

Font Rendering with Anti-aliasing

Font Rendering with Anti-aliasing and Hinting

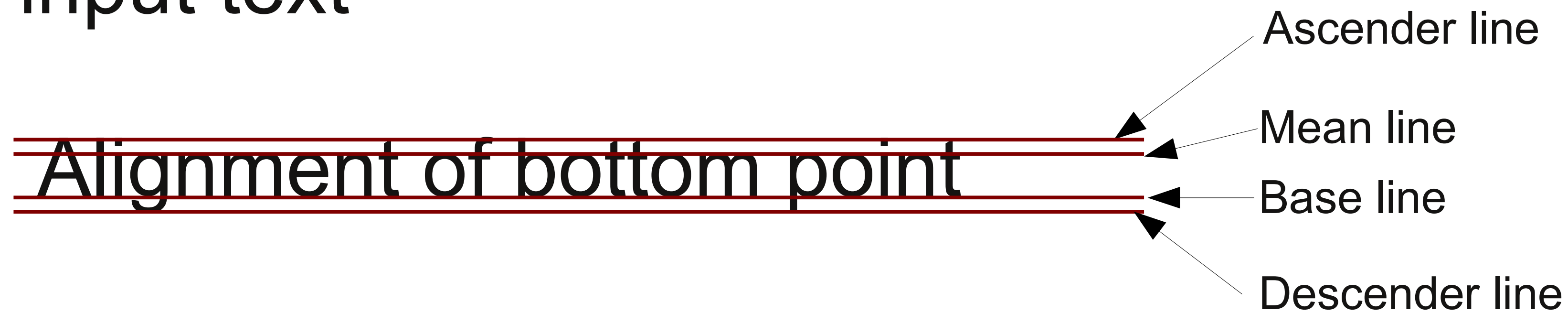


Sub-Pixel Font Rendering



Other Challenges in Font Rendering

- Arranging input text



- Glyph substitution based on context

ف ران س -> فرانس (France)

ق س ط ن ط ن ی ه -> قسطنطنیه (Constantine)

Font Rendering Engines

What they do:
- arrange input text from the input sequence to visual sequence
- substitute glyphs according to context (e.g. different forms of Arabic characters)
- ordering displayed text based on text flow direction (e.g. LTR vs RTL, Horizontal vs Vertical)

Some examples:
- Uniscribe (Windows)
- Pango (Linux)
- Apple Advanced Technology (Mac OS X)

Παν 語

- The name pango is from Greek pan (παν, "all") and Japanese go (語 , "language")
- Cross-platform
- Open source
- All major languages are supported
- Used as the rendering engine in Mozilla Firefox and Mozilla Thunderbird

Summary

Storing documents

- entropy
- variable-length coding
- run-length coding

Displaying documents

- font families
- font representation
- font rendering