UNIVERSITY OF OSLO
Department of Informatics

Dehyphenation

Some empirical methods

Master thesis

Ola S. Bauge

1 February 2012

Programmer's summary

To dehyphenate a text, the most straightforward way is to compile a frequency dictionary (which can be sourced from the very texts that are going to be dehyphenated). Armed with this frequency word-list, choosing the most frequent alternative—for example, *cooperate* instead of *co-operate*— will give the fewest errors.

Instead of using a hundred-megabyte hash to store the dictionary it's possible to do something clever with a Bloom filter, at the cost of some precision. The fallback method is to always delete the hyphen (which may cause an error rate anywhere between three and thirty percent; see section 6.6).

Typeset in 11pt Linux Libertine using LaTeX 2$_{\varepsilon}$.

Contains 25 pages **in colour**, namely p. 32, p. 58, and pp. 61–83.

This document contains examples of misshapen words and wrong hyphenations, which might introduce noise into dictionaries that are automatically compiled (as with the method described in this very text). Sorry about that.

Contents

CHAPTER 1

Introduction

> The optimal line length seems to be one which accommodates about ten to twelve words or 60 to 70 characters.
>
> —Herbert Spencer, *The Visible Word*

Given that it's introduced to close up 'holes' and 'rivers' in the paragraph, hyphenation of words across lines is literally a stop-gap measure. If left unchecked, these streaks of accidentally-exposed whitespace running through the text have the potential to derail the eye from the horizontal line, tricking it to scan along the vertical instead. To prevent this, hyphenation provides the flexibility needed to make the paragraph evenly spaced, preserving its visual integrity and 'greyness' while keeping both margins flush. Even when professionally-produced text is set with an uneven right margin it still tends to be hyphenated occasionally, to prevent excessive raggedness on the right; despite the fact that many readers now have had more than a decade's exposure to unjustified text from word processors and web browsers, printed matter still tends to appear in the familiar column-shape. One possible reason why the prevalent layout remains the one with even margins may be that neat justification produces a text-shape which is visually more consistent and thereby easier to navigate (more on this in section 2.5).

With the increasing digitization of text, however, the concept of an unchanging canonical line-length is beginning to seem almost quaint; today, the same news item can be viewed on either a widescreen desktop-monitor or a narrow handheld screen. The many different output surfaces require special care to ensure that the text stays legible at many different resolutions, to avoid forcing users to scroll back and forth because the lines are longer than their display, and so on.

Faced with this variety of formats, web designers tend to reach for something resembling the dependable boundaries of the printed page: the number of webpages which claim 'Best viewed at 1024×768' shows no sign of decreasing, though this may have more to do with a reliance on precariously-perched raster graphics than it has to do with a desire for carefully managed typography.[1] Once the futility of demanding specific resolutions begins to set in, what usually follows is the 'bargaining' phase of webdesign, where text is placed into columns of a fixed width likely to be smaller than the average screen.

Narrow columns on the web were of course pioneered by `suck.com`, with its single 291-pixel column. A few years later the templates that came with blogging software tended towards narrow columns, possibly because template designers were following the advice of typographic authorities like Spencer and Bringhurst. Although this mainstay of traditional typography has seen some acceptance on the web, the hyphenation which usually goes with short line-lengths has been notably absent: properties for controlling how a user-agent might hyphenate the text on a page are only slated to appear in CSS3.[2] However, the way web browsers may break the text at their convenience makes it untenable to correct hyphenations by hand, which is the usual practice in print.

Originally, typographers' recommendation of narrow columns may have been informed by the experimental results of Tinker & Paterson 1929 (which are referenced by Herbert Spencer). After summing up the previous research on the hygiene of reading, they indicate that the fastest reading speed was found in the condition with 10-point type set in lines that were 80 mm long.

More recent studies are showing that shorter line-lengths may not be all that helpful when reading from a screen: in Shaikh 2005, the fastest reading speed was obtained in the condition with 95 characters per line, and Ling & van Schaik 2005 also found that longer lines were better for scanning quickly. Contrary to these results, Beymer et al. 2005 showed increased speed and comprehension when lines were short, with the tradeoff that readers tended to skip over the ends of longer paragraphs.

While webpages remain unhyphenated, the WWW is host to an increasing mass of text which comes from print, or has been extracted from file formats that target print; as often as not, this text is hyphenated. Usually, the material has been digitized to make it available via search engines, and for this as well as for other applications, hyphens introduced by line-breaking produce mistakenly split words, leaving gaps in the index. This has a disproportionately large impact on longer exact-phrase queries: although in theory, a probability p of encountering a word corrupted by hyphenation only results in a likelihood of spoiling an

---

[1] "Best viewed at 1024 x 768" turns up 11.3 million hits on Google, and 49.8 million hits on Yahoo (7 April 2011).

[2] See http://www.w3.org/TR/2011/WD-css3-text-20110215/#hyphenation

individual query which follows the cumulative probability function equivalent to successive die-rolls, in practice the probability often goes up more steeply together with the length of the search string—in texts featuring even line-lengths and frequent hyphenations, the probability of bumping into a hyphenation usually approaches 1 with increasing substring length.

The problem is aggravated by the way most current search platforms get bogged down in ambiguity: just listing all the possible hyphenations of a string containing n optional breakpoints results in a search space which shows $O(2^n)$ exponential growth. It follows that the approach of simply spelling out all the permutations quickly becomes intractable, especially when processing text set in narrow columns—newspapers in particular tend to be relatively permissive with hyphenation, which means that each single letter has to be considered as a potential breakpoint. In this case, the n mentioned previously becomes the number of letters in the string rather than the number of syllables in the word.

Extraction of plaintext from paper-centric file formats is probably going to become less cumbersome eventually. Quite likely though, legacy documents from OCR are going to be with us for some time to come, and with them, several applications for which it can be helpful to make a guess at the precise form of the underlying text. Dehyphenation is one part of this picture.

1.1 Motivation

When breaking words across lines, the fragmented word is usually divided with a hyphen. Manning & Schütze 2000 calls this a line-breaking hyphen, by contrast to the lexical hyphen which occurs naturally in words like *co-operate* and dephrasals like *day-to-day*. Most of the problems in dehyphenation emerge from the simple fact that words which already contain a lexical hyphen can be broken along that hyphen without any additional marks; Manning and Schütze call this an instance of haplology. The polysemy introduced by these ambiguous hyphenations can become a notable source of noise, especially when processing text from multiple columns.

If this looks like an unnecessary distinction, it might be worth going over the hyphenation practice of Polish and Portuguese, where the lexical and line-breaking hyphens are sometimes realized differently. For both Portuguese and Polish text, there are some ways of breaking a word across a line that have much greater potential for being misread than what's usual; to mark hyphenations that would otherwise be too confusing, hyphens are printed both at the end of the line and at the beginning of the next line.

In the following quote, it's also worth noting that the lexical hyphen seems to be the more marked form of hyphenation.

> The same convention is used in Portuguese, where the use of hyphens is common, because they are mandatory for verb forms that include a pronoun. Homographs or ambiguity may arise if hyphens are treated incorrectly: for example, "disparate" means "folly" while "dispara-te" means "fire yourself" (or "fires onto you"). Therefore the former needs to be line broken as
>
> dispara-
> te
>
> and the latter as
>
> dispara-
> -te.
>
> A recommended practice is to type <SHY, NBHY> instead of <HYPHEN> to achieve promotion of the hyphen to the next line. This practice is reportedly already common and supported by major text layout applications.[3]

(SHY = *soft hyphen,* NBHY = *non-breaking hyphen*)

In the example quoted above, ‘dispara-♮-te’ leaves no room for confusion: at the end of the line we find a line-breaking hyphen, and the hyphen at the beginning of the next line has to be lexical——a distinction reflected in coding schemes which divide the ‘soft’ from the ‘hard’ hyphens. It’s a rare occasion to encounter hyphens that are so clearly marked, though: when passing over a hyphenation like ‘day-to-♮day’ it’s strictly speaking impossible to be completely sure wether it should be read ‘day-to-day’ or ‘day-today’ (that is, if it’s intended as a lexical or a typographical hyphen).

The only relevant piece of advice given by style guides on this topic is to warn against especially eye-catching hyphenations like *wee-knights, ex-acting, the-rapist* (Keary 1991, Liang 1983). This might be a sign that readers don’t lose any more sleep over hyphenated words than they do over all the other ambiguities of natural language——presumably people use their considerable background knowledge to sort out hyphenated words as much as they need to, however a workable simulation of this semantic–morphological background in a computer system isn’t currently feasible. The lexical part of background knowledge is much easier to approximate, using a word list; but even for a relatively unproductive language like English, a precompiled dictionary for dehyphenation won’t help in deciding on ambiguous dephrasals like ‘day-to-♮day’ since both ‘day’ and ‘today’ are independent words.

Modifying this dictionary approach slightly, using a frequency word list that’s gathered from a relatively freeform corpus of text, and that includes dephrasals, it becomes possible to make an educated guess for any item that occurs

---

[3] Freytag/Heninger, Unicode Standard Annex #14: ‘Unicode line breaking algorithm’, §5.3 (Rev. 24, 2009-09-18) – http://www.unicode.org/unicode/reports/tr14/#Hyphen

often enough. In this scheme, 'day-to-day' would be selected when it occurs at a higher frequency than 'day-today', which it usually does. A setup like the above will be developed and tested here as the 'lexicographic' method of dehyphenation.

1.2 Materials

For English text, it happens that dehyphenation at an acceptable quality can be achieved simply by using a reasonably comprehensive dictionary; this probably goes for other analytic languages as well, such as French. However, in the synthetic languages that incorporate more productive processes of word-formation, like compounding, the concept of a 'complete' word list can go from being problematic to nearly meaningless, depending on how much that particular language suffers from vocabulary explosion.

Although the experiments and discussion mostly revolves around dehyphenating Norwegian text, the focus here will be on methods that can be applied to the general group of alphabetic languages which are sensible to hyphenate in print. The Norwegian language makes for a convenient guinea pig here, given that it has a highly productive process of compounding: it's especially the productivity of N+N composites that interferes with top-down lexicography (Bungum 2008). As this is a kind of Germanic compounding, the impact on the lexicon is similar to what's found in Dutch and German.

In common with the other Scandinavian languages, Norwegian marks both definiteness and plurality of nouns using suffixes (*-en, -et, -a, -ene, -er;* see Table 2 on p. 33 for an example in context). This means that, relative to English, a simple dictionary gathered from comparable text tends to have more word-forms in it. At the same time, Norwegian morphology is not as hyperactive as is the case in agglutinative or highly-inflecting languages such as Finnish, which makes it a suitable middle ground to survey the problem from: quite likely, there will be at least a few languages which require morphological analysis for tasks like dehyphenation, but in practical terms it holds some interest to estimate just how much can be gained from robust methods that are relatively language-independent.

The material used for testing comes from a subset of the documents in NORA, the Norwegian Open Research Archive, in the state that it was in circa mid-2009.[4] These texts are representative of a kind of document which is becoming increasingly common, that is, 'mid-quality' text typeset by computer but still prepared with print first in mind, usually without passing through the hands of professional proofreaders or copyeditors. The documents in NORA are typically Master's-level theses or equivalent, from a wide range of academic disciplines.

---

[4] Currently accessible online through http://www.duo.uio.no/englishindex.html

Note that most of these documents are typeset 'ragged-right', typically because they originate from Microsoft Word®: these texts will only be hyphenated along preexisting hyphens (e.g. *once-over*). Since these documents never favour the null hypothesis, it may skew the results somewhat towards methods which identify more lexical hyphens. The material is still representative, in the sense that it's a real collection of academic documents which could be indexed more comprehensively with dehyphenation. And since there is no watertight method for automatically detecting which documents feature an uneven right margin, it's useful to measure performance in this kind of situation where documents arrive in an unordered mix.

The dataset contains around 1.5 gigabytes' worth of plaintext, of which ≈1 GB is English and the remainder mainly Norwegian, with a few Danish and Swedish texts thrown in; French and German number only about 40 texts each, and a single document is marked as Sami language. Most of the English texts were not written by native speakers, hence this part of the material can't claim the level of authenticity offered by the Brown or Leverhulme corpora; this shouldn't affect dehyphenation performance very much, though.

The conversion from PDF to plaintext was carried out using PDFBox 0.8.0, customized to mark up footnotes and title text.[5] The language of individual documents was detected with statistical *n*-gram analysis using software adapted from Apache Nutch.[6] A word-count on the text comes out to about 250 million words all in all, making for an average word length of 5 characters plus space, which is within the expected range of 4½ to 5. Copies of the actual text files that were used are available on request.

1.3 Acknowledgements

Thanks first of all to my advisor Jan Tore Lønning, for his considerable patience and support even when work on what should have been a simple thesis dragged on beyond all reason. Credit is also due to my co-conspirators on the WeScience$_0$ project, who wrote most of the code for interfacing with PDFBox to extract the text from PDF documents. And finally, thanks to everyone in and around the University of Oslo research group in Logic and Natural Language who made it a uniquely stimulating environment for learning.

---

[5] Available from http://pdfbox.apache.org/

[6] http://nutch.apache.org/

1.4 Outline

There are three main parts to this thesis: chapters 1–3 give background information, chapters 4–6 describe possible methods for dehyphenation and evaluate them against each other, and the appendages feature technical details.

This chapter gave a justification for studying dehyphenation and described the corpus of text that will be used for testing.

The second chapter will provide theoretical background and some flailing attempts to clear up the status of the hyphen and related marks in current punctuation theory.

Chapter 3 gives an outline of previous work. It's only two pages long, reflecting the fact that there hasn't been much previous work directly related to dehyphenation.

Chapter 4 presents possible empirical methods for dehyphenation, concentrating on one lexicographic and four morphological methods. This constitutes most of the significant research that's contributed here.

Chapter 5 contains supplemental methods which typically provide small adjustments. These tend to be more language-specific and less flexible relative to the empirical methods.

In chapter 6, the methods presented in the previous two chapters are evaluated on a sample of hyphenated words.

Chapter 7 sums up and makes recommendations for dehyphenation practice and further work.

Finally, Appendix A describes some of the techniques for removing noise from the corpus that were used or considered for the text material; Appendix B lists the code used for evaluation; and the bibliography is in Appendix C.

CHAPTER 2

Background

Manning & Schütze sort hyphens into four primary kinds: line-breaking hyphens, which are placed on the page for typographical reasons; the lexical hyphens that are properly part of words like *to-morrow;* and the pre-modifier grouping hyphens which create modifiers from several words, as with *once-quiet.* No name is provided for the final type of hyphen, possibly because it's somewhat vaguely defined: it occurs 'where a phrase is seen as in some sense quotative'. The examples given are *child-as-required-yuppie-possession, "take-it-or-leave-it",* and *90-cent-an-hour*, which all fit the description of *ad hoc* phrasal compounds. Depending on the material, these quotative one-off uses can make up a substantial share of the hyphened words that go against the null hypothesis (they usually shouldn't have any hyphens removed; see section 6.3). As such, the hyphen that appears in ad-hoc phrasal compounds will be examined in section 2.2.

These distinctions might seem more fine-grained than what's strictly needed; after all, the only job a dehyphenation algorithm has is to erase the line-breaking hyphens and leave all the other kinds. Though intuitively appealing, this approach is hindered by the fact that the line-breaking hyphen is by far the most difficult to detect, occurring as it does basically at random. Barring the prospect of building a system for classifying something unsystematic, this suggests an eclectic strategy for detecting line-breaking hyphens: basically, consigning a dash to the dustbin as a typographic hyphen is only done as a last resort after every other possibility has been exhausted. With this approach, it becomes the case that being able to tell all the types of hyphen apart might improve accuracy, which provides a convenient excuse to go hyphen-spotting.

2.1 Sublexical hyphens

Although punctuation has increasingly come to be accepted as a valid area of research since the 1990 publication of Nunberg's *The Linguistics of Punctuation*, apostrophes and hyphens have been left out in most of the major approaches (including Meyer 1987, Jones 1996 and Say 1998). As Jones puts it, the reason behind this is that they're *sub-lexical* and cannot influence the syntax:

> the sub-lexical marks change the meaning of the words that contain them, wheras the inter-lexical and super-lexical marks change the manner in which the words combine to produce an overall meaning or purpose...

Or, in Nunberg's wording (on p. 68):

> The hyphen, for example, can be regarded as an affix that attaches to a word-part, rather than to a word, and as such it does not interact with any of the indicators of syntactic categories...

When it's used to hyphenate words across lines, the hyphen does typically separate syllables instead of words; however, the intersyllabic form of the hyphen seems to attract more attention in the more exotic literary use where it indicates stuttering or slowly enunciated speech (Meyer 1987; Partridge 1953; Skelton 1949, p. 124). The hyphen also appears placed between syllables in printed examples of 'expletive infixation' such as *abso-bloody-lutely* (McMillan 1980).

The study of sublexical phenomena, meanwhile, has been left mostly to the odd morphologist, and experimental cognitive psychologists such as Hyönä & Pollatsek 1998, Liversedge & Blythe 2007 and so on. The lack of interest in the sublexical level may be in part due to the common simplifying assumption that words are atomic, as spelled out in the Lexical Integrity Hypothesis: this in turn may stem from the theoretical fiction that a language is an inert object of study which can be completely known (as critiqued in Harris 1981).

2.2 Hyphens in dephrasals

Above the sublexical level, the inter-lexical hyphens are divided mostly between the pre-modifier grouping hyphens (as in *once-quiet*) and the 'quotative' usage in rank-shifted phrases, ad-hoc phrasal compounds, or dephrasal 'nonce-uses' such as *do-it-yourself, pay-and-display.*[7] The more a unit is like a phrase, the higher the probability that it contains internal function-words; this will have practical consequences for dehyphenation in section 5.3.

Ad hoc phrasal compounds are discussed in some detail in Meibauer 2007, where, among other things, he uses German case agreement to argue that they're slightly transparent to anaphoric binding: this runs counter to the position of

---

[7] Examples from Nunberg et al. 2002.

Ackema & Neeleman 2004, that insertion of a phrase from syntax into morphology results in an opaque lexical unit.

Meibauer also makes mention of something else which sets these dephrasals apart: ‘Incongruity on the word level means that it is unusual to combine a phrasal meaning with a word meaning.’ This type of incongruity shows up at its most obvious in outlier ‘stunt words’ such as ‘general getting-ready-to-fly-to-Canberra-tonight-iness’ or ‘de-Prince-Charming-from-Shrek-ify myself’.[8] This way of hitching phrases to patterns like X-*iness* is also demonstrated by relatively respectable words like *get-at-able* and *come-at-able*, so presumably it’s the exaggerated length of the inserted phrase which makes the former pair of examples stand out. This might indicate the existence of a continuum between acceptable and marginal phrase-compounds.

Ad-hoc dephrasals are uniquely suited to one-off uses since they have the freedom to simply provide a gloss of the sort of thing they refer to:

> Or is this bad logic, fit only for cultural theory seminars and Buffy-the-Vampire-Slayer-as-Postmodern-Signifier conferences?[9]

They can also contain personal pronouns; the context of the following Norwegian quote from Tore Rem is also strikingly familiar to the preceeding quote from Tom McCarthy.

> Eller på om ei fagbok kommer ut på Universitetsforlaget eller hos Forlaget Kjøp-deg-en-bokutgi-velse-hos-oss-så-får-du-betalt-i-universitetenes-tellekant-systemer.[10]
>
> (‘Or wether a textbook is brought out on the University Press or with the Publishing House of Buy-Yourself-a-Vanity-Publication-Redeemable-in-University-Credit.’)

It’s possible for the personal pronouns to outnumber the other function-words in a dephrasal, making it conspicuously quote-like. That seems to be the case in this example, quoted from the short story *Bridgehead* by Frank Belknap Long:

> The weapon in Eddie’s clasp looked as though somebody had been sweating holes in the Government’s post-war priority programme. Apparently a lot of valuable new metals had gone into it, along with some very tensile mental haywire. It had a startling you’ll-never-guess-where-*I*-came-from look.[11]

This also illustrates the point of Meibauer (from p. 244) about the relative ease of anaphorical binding into phrasal compounds—here the pronoun ‘I’ refers to the weapon, if it refers to anything in particular. Especially when the personal pronoun appears stressed inside the dephrasal, it serves to highlight the quote-like nature to the point of straining credibility as a modifier; it makes the weapon seem like it’s on the verge of participating in the conversation.

---

[8] Collected from Twitter by Mark Peters; quoted from the archives of *Wordlustitude* (http://wordlust.blogspot.com/)

[9] Tom McCarthy, *Tintin and the Secret of Literature*, p. 11. ISBN 1-86207-831-4

[10] Tore Rem, Sakprosakritikk på norsk. PROSA 04/07. www.prosa.no

[11] Frank Belknap Long, ‘Bridgehead’. Originally in *Astounding Science Fiction*, August 1944.

It's perhaps notable that Long is the only one of the quoted writers who opted not to wall off the sentence with the phrase-compound in it; unlike the other two, he seems to be less concerned with preventing the phrase-compound from interacting with its context. This blatant disregard for hierarchies of seriousness may be one of the stylistic choices making his text easier to dismiss as 'subliterature', even though Rem actually has more personal pronouns in his chatty phrasal compound than Long does ('buy *yourself* a book with *us* then *you* will get paid in the universities' counting·edge systems').

(Both Rem and McCarthy downplay the shock of the overly long phrase-compound by isolating it behind a kind of double-glazed window of alternate possibilities nested two deep——the relevant sentences both begin with 'or' and move gradually towards the less likely alternatives. Not only does this single out the thing described by the phrase-compound as just a foil to the more serious option, it also reduces the number of possible anaphoric bindings that could propagate upwards to interfere with the meaning above the sentence-level. If there is such an interaction between stylistics and the use of phrasal composites, it probably isn't relevant here, however.)

So far, *ad hoc* phrasal compounds have received less attention in linguistics than they have in lexicography, where they're usually grouped with the 'nonce-words'; given that they're defined as primarily one-off coinages, they would be less popular with system-minded theorists precisely because it's difficult to make very strong generalizations about them.

On the other hand, calling something 'ad hoc' might sound out of place when it's applied to idiomatic phrase-compounds that get used as often as most words, such as *up-and-coming* or *well-to-do*. In instances like these, what seems to take precedence is that the phrase retains a high degree of *transparency:* i.e., this kind of phrase-compound doesn't normally undergo semantic bleaching or turn morphologically opaque, unlike what usually happens when words go through lexicalization.

Deviations from this trend are mostly found in cases where the dephrasal nonce-formation is run together to form a new word: for instance, *wannabe* can be used as a free-standing noun, whereas *would-be* can only act as a modifier. This has the snag that running words together tends to get the freshly-coined word classified as slang, with only a few exceptions——e.g., *ampersand* is actually a slurred-together version of 'and per se and', but was still adopted by dictionaries (possibly, it sounded like it ought to be a technical term while its real origin was obscure).

With the internal structure smoothed over like this, a word can finally turn more opaque and eventually become susceptible to semantic bleaching. To take the word 'hand·kerchief' as an example, it may be reanalyzed in ways not possible with the contracted form 'hanky': a cartoon fish referring to its 'finkerchief' might be a workable pun in many contexts where a word like '*finky' on the other hand would fail to register. With dephrasals, this moment of opaqueness typically arrives when the lexeme no longer contains any obvious function-words that can support easy reanalysis.

The way that long dephrasals often wind up relegated to the outskirts of expressive or 'marginal' language may in fact be caused by having internal function-words on display: also, since rank-shifting a phrase requires a long-throw insertion all the way from syntax into morphology, this could provide an explanation why they stand out among other types of words. If the reanalysis provoked by encountering word-internal function-words triggers a drastic change in reading strategy while mid-sentence, it may be the case that overly complicated dephrasals have potential for disrupting the rhythm of a sentence which is so great that it comes into conflict with the ideal rate of information transfer (which might be constant, cf. Genzel & Charniak 2002).

Meibauer also highlights that ad-hoc phrasal compounds often are seen as *witty;* if phrase-compounds tend towards an aberrant information–transfer rate it may contribute to an impression of amusing marginality due to the way humour typically depends on surprising the audience (see Ritchie 2004, ch. 4). This in turn is easily dismissed as childish, since it allows an immodest degree of expressive freedom.

Beyond mere expressiveness, it's possible to find groupings constructed with interlexical hyphens that go so far as to violate the structure of grammar, as in the following non-technical exposition on the syntax of Japanese modifiers:

> Thus, whereas in English one says, "the delicious chestnuts" but "the chestnuts on the table" and "the chestnuts that I ate yesterday," the order in Japanese is (consistently) "delicious chestnuts," "on-the-table chestnuts" and "I-yesterday-ate chestnuts."[12]

Here, the context makes it clear that the hyphenated lumps are standing in for Japanese modifiers of various kinds, but calling it quotative begins to take on strange connotations when the phrase comes from a different language family. In this case, the internal syntax of the 'quote' is completely separate from what surrounds it, constituting basically an instructive toy language constructed on the spot.

This metalinguistic use shows that, in the limit, hyphens can force almost any conceivable group of words into being a constituent. Since constructing dephrasals can often be an arbitrary and highly self-conscious route of word-formation, this makes it less unforced and 'natural', more artificial and affiliated with language play.

---

[12] Peter Sharpe, *Kodansha's Communicative English-Japanese Dictionary*, p. 1149. ISBN 4-7700-1808-8

2.3 Theories of punctuation

Nunberg 1990 concentrates on the notion of the underlying structure that motivates punctuation, being primarily occupied with a *text grammar* which ties into syntax, as specified on p. 21:

> I should stress that I am using the term ″text grammar″ here in a relatively literal sense, in distinction to the way the term is used in much of the literature on discourse analysis. … the term ″grammar″ is to be understood as a set of rules that determine syntactic relations among explicit formal elements (as opposed to describing essentially semantic or pragmatic relations of ″coherence″ and the like).[13]

This stands in sharp relief to Meyer 1987, who concludes in §1.4 that 'American punctuation is ultimately pragmatic'. A statement superficially similar to Meyer's can be found in M. B. Parkes' 1992 *Pause and Effect: An Introduction to the History of Punctuation in the West*, which briefly mentions punctuation acting as pragmatics (while discussing exclamation marks, on p. 2):

> … The writer employs the symbol here to encourage readers to draw on their own experience so that it may contribute to the assessment of the message of the text. By invoking behavioural experience in this way punctuation becomes a feature of the 'pragmatics' of the written medium. In spoken language such contributions to a message can be conveyed in various ways both linguistic and paralinguistic – such as a repertoire of intonations, or gestures and facial expressions – which can be employed because an interlocutor is present.

The above sense of 'pragmatic' most resembles kinesics: where Meyer argues that a writer's choice of punctuation is primarily stylistic, Parkes sees a parallel between the punctuation of a text and the use of body movement and intonation in face-to-face conversation, which is probably the more relevant line of thinking here. As channels of communication go, both punctuation and kinesics are relatively independent, in that they can both be said to be set apart from the words that accompany them by their closer attachment to the physical circumstance of communication; this holds true wether it's the condition of being pyhsically present or the precondition of having left marks on a surface.

One fairly obvious link between punctuation and kinesics is the insertion of "air quotes" in conversation. Another example, in which the connection to kinesics could be said to flow the other way (from gesture to page-based writing), appears in the following quote from a personal webpage:

> LCD displays emit polarized light, which is usually waving in approximately
> diagonal direction…polarizing sunglasses only pass vertically polarized light.
> So when you look through polarizing sunglasses on LCD display, at right angle,

---

[13] (See ch. 8 of Jones 1996 for an overview of the later integration of Nunberg's theory into discourse semantics.)

> no light goes through, because display gives out \ diagonal polarized light, and glasses at that angle only pass / polarized light.[14]

In the last sentence, an ASCII backslash and a forward slash are included for their graphic form—–they're used in a graphically immediate way, simply as lines or makeshift pointing arrows. Although acting as modifiers, the slashes form a close analogue to the use of kinesics in conversation: when reading this passage out loud, the moments corresponding to the slashes are likely to be accompanied by the speaker tilting their hand or arm to visualize the slant of polarization.

The intuitive visual sign adds some much-needed immediacy to an explanation of a difficult concept in optics (it probably sounds strange to most people that a ray of light actually 'vibrates' instead of travelling in a straight line). Notably, this is less like the use of emoticons in online conversation and more like the sort of miniature inline illustrations which are proposed in Tufte 1983 under the name 'dataword'.

2.4 Punctuation and convention

Punctuation only settled into standardized shapes after the printing-press arrived. Before Gutenberg, the sets of pointing marks varied by region and were often idiosyncratic to a community of scribes (Parkes 1992). Now that computer layout 'removes the difference between letter and image', in the words of Schwemer-Scheddin 1998, the stage could be set for a more fluid interaction between the graphical and the lexical. Although this development is still fledgling, there are faint signs of a resurgence in the kind of processes of graphic conventionalization that existed in manuscript culture (Parkes 1992, p. 58):

> Because the *diple* was used to indicate quotations from authorities it became one of several methods employed to identify gnomic utterances or *sententiae*, … For the same reason the *nota* acquired emphatic significance, and, like italic type, was employed for emphasis even where there was no quotation.

As described, it seems as though the graphic sign > underwent semantic bleaching through repeated use, not unlike the contemporary erosion of the word 'literally' as it's currently being sanded down to a simple intensifier. This kind of conventionalization of a graphic symbol is rather similar to the lexicalization of words through frequent use, and the process seems to operate similarly wether it's a symbol that arose from lexical writing, as in a ligature or a shorthand, or from a standardized drawing on the page.

Many of the same traits of transparency and reanalysis show up for symbols as well as for words: for instance, substituting an @ for the letter A to give the

---

[14] Dmytry Lavrov, *Strain patterns in plexiglass.* http://dmytry.blogspot.com/2009/07/strain-patterns-in-plexiglass.html · Quoted by kind permission of the author.

impression of being up-to-d@te clearly depends on an audience who can recognize the *a* embedded in the graphic shape, just as the old-fashioned practice of writing '&c.' as shorthand for 'et cetera' requires readers that are able to associate the graphic symbol & with the lexeme 'et'.

The hyphen can't be said to have an obvious phonological realization, and in fact is a slightly difficult punctuation mark to talk about (see for instance the discussion in Nunberg et al. 2002 over what to call the various kinds of dashes, en-rules and so on). Possibly the difficulty arises because, like the marks \ and /, the hyphen is a straight line, giving it a fairly immediate graphical presence; it's also the most common punctuation mark to trace a spatial relation (left↔right) directly on the page.

Since the typographical hyphen is the one that relates most closely to the space of the page, to round out the background for dehyphenation it might be helpful to take a closer look at the intersection between typography and semiotics.

2.5 Surface-oriented approaches

Turning to the field of information design, Waller 1980 describes typography as 'macro-punctuation', providing a link between punctuation marks and the layout of space on the page:

> Punctuation is the single aspect of written language, for which grammatical rules exist, that does not represent words themselves but the spaces between them. It is, then, an organizational system at the micro-text level functioning in much the same way as typographic signals and the use of space at the macro-text level.

Notably, punctuation tends to look out-of-place on the title pages of books, a place where designers have long had plenty of leeway to use space, varying typefaces and text size to indicate structure; here, it might be the case that when punctuation is used to convey what's already communicated through graphical layout it results in a text that comes across as pedantic due to unneeded redundancy.

For this sort of redundancy to even register with readers at all, some overlap must exist between the roles of punctuation and graphical layout. (There's a more hands-on discussion of this to be found in McLean 2000—in particular, p. 21 features a reproduction of what he calls the 'less well-designed title page' of '*The Emigrant, and Other Poems.*' from 1833. In that book, the comma and the period of the title are actually printed on the title page, which gives an immediately awkward impression.)

Semiotic approaches to punctuation have emphasized the two-dimensional nature of writing surfaces, rather than the ideally one-dimensional progression of a perfectly well-behaved text; Nunberg 1990 basically glosses over this point by postulating a set of 'pouring rules' that place the words onto the page in some unspecified way. In contrast, Harris 1995 argues (on p. 46) that 'There simply is no counterpart in speech to the use of a surface, which is the commonest way in writing of articulating spatial relations.'

The way surfaces can be used to show spatial concepts at fewer removes is also something that takes centre stage in Waller 1987:

> A dichotomy emerges between a linear model of written language in which a relatively discreet typography 'scores' or notates the reading process for compliant readers, and a diagrammatic typography in which some concept relations are mapped more or less directly on the page for access by selfdirected readers. Typographically complex pages are seen as hybrid forms in which control over the syntagm (used here to mean the temporal sequence of linguistic events encountered by the reader) switches between the reader (in the case of more diagrammatic forms) and the writer (in the case of conventional prose). Typography is thus most easily accounted for in terms of reader-writer relations, with an added complication imposed by the physical nature of the text as artefact: line, column and page boundaries are mostly arbitrary in linear texts but often meaningful in diagrammatic ones.

In light of this, hyphenation can be seen as a way to prevent readers from being misled into unintentional *diagrammatic readings*——it does this by letting typesetters fill in any 'holes' or 'rivers' of vertical whitespace that might otherwise leave the paragraph in a tattered shape with potential for deluding the eye into turning the text on its side, as if it were an acrostic or some sort of crossword puzzle. Used like this, the hyphen acts as a micro-typographical bridge between the sublexical level and the macro-typography of the page.

At last, this finally provides a likely answer why text still makes it to print with both margins flush: when lines range up to give the visual impression of an overall column-shape, that column constitutes a very simple kind of text-diagram with only a single function, to facilitate reading prose at a steady pace. Following this assumption, maintaining an even right margin should become less important when there are several columns on the page; and as it happens, many typographers advise against fully justifying text across multiple columns. A line-breaking hyphen at the margin is then understood intuitively as a typographical element relating to the text-diagram of the column itself rather than the prose text contained in it.

When there's only a single column on a page, the column-shape usually isn't outlined: instead, the borders lie implicit in the overall layout of the text, appearing naturally to the reader in the unfocused peripheral vision. The use of hyphens in dephrasals might conceivably be similarly motivated by the visual impact of bundling words together with dashes——this produces the optical effect of making the bundle show up as a single lexical unit in the peripheral vision of the reader, which then resolves into individual words when it's looked at directly.

If use of the hyphen is motivated by optical pragmatics, placing it somewhere between the spatial and the semantic marks, it might explain why it's so difficult to pin down (as well as why Partridge 1953 treated it under the heading 'hyphens and oddments'). On the space of the page, among the most common punctuation marks it's - which most directly traces a line. Its flexibility flows naturally from this—the expressive potential of the hyphen is simply the expressive potential of the horizontal line.

CHAPTER 3

Previous work

3.1 Grefenstette & Tapanainen

Grefenstette & Tapanainen 1994 describes an experiment where, as a preliminary to tokenization, they ran the Brown corpus throughthe UNIX typesetting program `nroff(1)` and then joined all of the hyphenated words, omitting end-of-line hyphens by default. The workhorse of their setup was the Flex rule

```
[a-z]-[ \t]*\n[ \t]* { printf("%c", yytext[0]); }
```

This appears to distinguish only the case where the single character preceeding the hyphen isn't lowercase alphabetic. Presumably this scheme preserves the hyphen of 'initialisms' like *D-night* and *X-rays*, although it may not work with a compound like *L-5-vinyl-2-thio-oxazolidone* (if it isn't broken after the L, the 5, or the 2). Note that their rule never peeks backwards to see if the word begins with a capital letter; this may have something to do with backtracking in a regular expression being prohibitively expensive in 1994.

The fact that they don't attempt any other analysis of case may also be in part because they were opposed to doing tokenization at an early stage:

> Here, if one had access to a dictionary and morphological package at this stage, one could test each of the 12473 cases by analyzing the constituent parts and making more informed decisions, but such a mechanism is already rather sophisticated, and its construction is rarely considered for such a preliminary stage of linguistic treatment. One may consider the 615 errors (out of 1 million words) as so many unknown words to be treated at some later stage, or just accept them as noise in the system.

Now that eight cycles of Moore's Law have come and gone, the task of preliminary tokenization is beginning to look considerably less involved. Since G&T's study is the most significant mention of dehyphenation I've been able to find in the literature, their experiment will be replicated in section 6.1.

3.2 Liang (TEX)

The hyphenation algorithm designed for TEX has some interest here, since it's the second most popular source of documents in the dataset. The fundamental algorithm which has been in use since the release of TEX82 is of course described in Liang 1983, but even though this algorithm is thoroughly documented and completely determinstic, it's still a difficult beast to second-guess.

Liang's algorithm works with patterns indicating likely breakpoints in substrings; for example, `e3fine` covers both of the hyphenations *de-fine* and *re-fine*. However, even if we knew the set of patterns that was used to hyphenate a text this wouldn't be much help in reversing the process. Taking `e3fine` as an example, even if we encounter the substring '...e-♮fine' there's no guarantee it was broken because of the `e3fine` pattern; the word might simply have been split along a naturally occurring lexical hyphen, as in *large-fine* or *double-fine*.

The problem stems in part from the fact that a hyphenation routine running at the time of document generation can depend on the original text to constrain it, whereas a dehyphenation program is trying to discover what that original text was in the first place. More generally, this is a case of A⟶B not necessarily implying B⟶A: the possibility that a substring might have been broken by a pattern is no guarantee that it actually was. Since every decent hyphenation routine will try to break along lexical hyphens, attempting to run the TEX algorithm 'in reverse' is doomed to failure.

3.3 Commercial products

'xcorrect' is the only commercial dehyphenation package I've been able to find. It seems to work primarily with German text, which is something of a special case since all the nouns in the language are supposed to be capitalized. This paves the way for a straightforward approach of keeping the hyphen only if it appears before a capital letter—a method which can be implemented using a single regular expression such as the following:[15]

```
s/-\n([a-zäöß])/\1/
```

With this method, we get Bahnhof-♮Kaffee⟶Bahnhof-Kaffee,

Echtzeit-♮strategiespiele
⟶Echtzeitstrategiespiele,

Kurz-♮nachrichten-Dienst
⟶Kurznachrichten-Dienst.

---

[15] The three examples displayed at http://download.xeebion.com/xcph/htm/unhyphen.htm are consistent with this simple dehyphenation strategy.

CHAPTER 4

Empirical dehyphenation

In the wording of Grefenstette and Tapanainen, hyphens which are introduced by line-breaking are only *circumstantial*, related to the width of the page and not to the meaning of the text. Circumstantial hyphens might be expected to appear in a much more random pattern than the naturally occurring lexical hyphens, since they can be placed between any pair of syllables. This means that the potential noise introduced by circumstantial hyphens tends to be dispersed over a bigger number of forms: for instance, the four possible breakpoints in *con-cate-na-tion* makes four possible hyphenated variants. Given this, we might expect to be able to sort the signal of lexical hyphens from the noise of the circumstantial ones; however, it's difficult to gather enough data from hyphenated words alone.

One obvious source of additional words is the text itself: in the NORA dataset, only 1 out of 600 words are fragmented by hyphenation. Making the uncontroversial assumption that the divided and undivided words are both sampled from the same general population of words, this allows us to build a word list for dehyphenation from the very mass of text that is being processed.

Given the ergodic property of language, the accuracy of this method is likely to stabilize on a sufficiently large collection of documents that have some degree of internal consistency and a reasonably low level of noise. In this report, the dehyphenation methods will be tested on plaintext extracted from PDF files that likely contain much less noise than your average webpage; as mentioned in the introduction, this is a type of document which is becoming increasingly more common. Just how sensitive the lexicographic method is to noise will be discussed further in section 7.4.

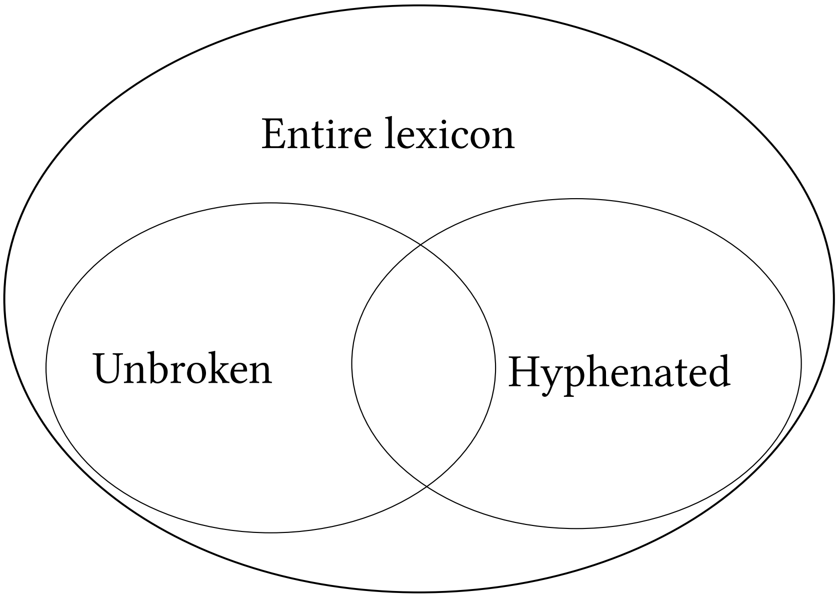

Figure 1: Hyphenated and unhyphenated words are both subsets of the same overall lexicon. The lexicographic algorithm exploits the overlap between the two subsets.

4.1 A lexicographic dehyphenation algorithm

An implementation of the empirical lexicographic algorithm is included in section B.9 on p. 71; this makes no assumptions about specific languages except indirectly through the set of allowable characters.

Words are picked out in the simplest way possible: any stretch of alphabetic characters gets taken in as a lexical item.[16] This eliminates the need for any special-case handling of punctuation, nonstandard characters and so on. No attempt is made to discover sentence boundaries, which means there's no way to distinguish the capitalized words which are proper nouns from the ones which just happen to begin a sentence. The algorithm still appears to be robust enough to achieve results despite this and other sources of noise. The words found with this procedure are compiled into a frequency dictionary, kept in a hashtable keyed by the string resulting from lowercasing the word and stripping it of hyphens. Under each entry, separate frequency-counts are kept for all the variant capitalizations and hyphenations.

---

[16] That is, the longest possible contiguous sequence of only alphabetic letters and hyphens.

online	12817	OnLine	20
Online	5766	onLine	12
on-line	897	on-Line	3
On-line	382	ONline	1
On-Line	43	oNLine	1
ONLINE	40		

Table 1: Frequencies for the word "online".

The resulting frequency word-list allows picking out the most popular form of a dictionary entry. For example, under the key "online" we might find the frequencies listed in Table 1. The capitalized form *Online* is unusually common here due to the fact that the word is quite popular in titles, while the 40 instances of *ONLINE* were probably harvested from index pages typeset in uppercase. It's still clear that the lowercase regular form of a word tends to be the most frequent, despite the noisy environment.

Given these frequencies and asked to dehyphenate 'on-♮line', the algorithm would choose the most frequent of the possibilities (which is *online* without a hyphen). If the word were hyphenated like 'on-li-♮ne' instead, the variant form *online* wouldn't be in the running as a possible dehyphenation; so, the algorithm would pick *on-line* instead.

Before counting word frequencies, lexemes which contain uncertain hyphens (due to having been hyphenated across lines) are split off into their own set. This results in a 'broken' and an 'unbroken' population: the lexicographic algorithm runs on the basic premise that the hyphenated words are sampled from the same general population as all other words. Under this same assumption, when a full dictionary is unavailable or unfeasible the unbroken words can stand in for the entire lexicon. Then, the probability that a hyphen at the margin is lexical can be estimated by finding the proportion of lexical hyphens in the subset of the lexicon that the line-broken string could possibly have been sampled from; this set is defined by the string with and without its uncertain hyphen.

In the actual implementation, there's an additional complication. When the most popular variant is a regular word without a hyphen, the algorithm doesn't actually recommend it, but instead refrains from making a decision: although it's possible to use the frequency word-list to make negative verdicts, the implementation presented here only delivers positive and neutral evidence.[17] This is a feature which originally crept in due to a programming error, but we'll see at the end of section 6.6 that including negative verdicts in the lexicographic method may actually be detrimental to its performance.

---

[17] It returns nil instead of false; as seen in Fig. 5 on p. 47, neutral evidence eventually becomes negative evidence unless it's overturned.

The method outlined above has the advantage that it's fairly robust: it doesn't normally need very fine-grained language detection, and it can process relatively heterogenous material as long as there isn't an overwhelming amount of noise (such as crosstalk between different languages, or rampant misspellings). The major weakness of the simple lexicographic method is that performance depends completely on coverage: inspecting the Venn diagram of Fig. 1, it should be obvious that the coverage can't be made greater than the intersection between the hyphenated and the unhyphenated words without involving some sort of external dictionary. This will come back to haunt us in section 6.4, where we'll see performance limited by the fact that the NORA test set contains 34 517 unique divided tokens (that is, hapaxes which happen to be hyphenated). The impact of this is that the word-list approach has to pass on at least 8.27% of the hyphenations, since there is no other possible word they can be compared to.

4.2 Morphological analysis

Since it works by simply consulting a frequency dictionary, the lexicographic method tends to suffer from a lack of predictive power: it's not much of a 'learning' algorithm, seeing as it can only predict anything as a side-effect of describing it. On the face of it, the tools with the greatest potential for improving on this would seem to be automatic morphological analyzers, especially ones that detect bound forms. This might provide some much-needed negative evidence, for example, in those cases where a bound form appears after a dash-and-newline (typically a sign that we're dealing with a line-breaking hyphen which should be deleted, as in *process-♮ing* or *refresh-♮ment*.)

As a rule, statistical morphological analyzers tend to overgenerate morphs, but this is less of a concern here than it might be elsewhere. Pinpoint precision is in fact less important for dehyphenation than what's usual in most linguistic analysis, since there's normally little harm done if an analyzer hallucinates a few extra morphemes into existence.

At runtime, there's a safety-net implicit in the fact that only two dehyphenations are possible for a given string, which means there's not much there for spurious morphs to have an effect on unless they happen to fall at a word boundary; the biggest concern in this department would be mistaking full words for bound forms.

Although the robustness is convenient for this particular application, the reader should beware that the results reported below may not generalize to other areas.

4.3 Linguistica

Goldsmith 2001 describes an algorithm for the unsupervised learning of morphology from a corpus using minimum description length analysis: in this scheme, substrings are unified into morpheme hierarchies depending on the potential savings in information-theoretic bits. The algorithm is implemented in the software package Linguistica, which is freely available under the GPL.[18]

Linguistica was used to identify suffixes by reading 500 000 tokens from the given corpus. Separate corpora were prepared for Norwegian and English by taking the word-lists from the NORA dataset and rejecting all entries with capital letters or hyphens in them. (This is relatively unproblematic given that the morphological analyzer is looking for the most common patterns in regular words, making the outlier tokens less interesting.) The word lists were ordered alphabetically; in this case, the fact that we're looking for suffixes mitigates any skewing this might introduce towards the beginning of the alphabet.

Similar amounts of suffixes were detected between English and Norwegian (7 374 resp. 7 224). As displayed in the inset of Fig. 2, the amount of suffixes detected is similar at each corpus count, especially when it approaches 1. For higher corpus counts, there are some conspicuous peaks in which more suffixes were detected for Norwegian, conceivably reflecting its somewhat more active morphology. There are no obviously similar differences in the low-frequency suffixes, that have the least basis in data. This could result from the low-count morphs being mostly noise; alternately, it could just be that the morphology of the two languages looks very similar at low frequencies.

The sharp exponential rise towards the low end motivates discarding the low-count 'suffixes' as noise in the system: for the evaluation in section 6.6 the cutoff value was set to 2, meaning to reject suffixes based on a corpus count less than three.

4.4 Morfessor

Morfessor 1.0, described in Creutz & Lagus 2005, is a system for unsupervised morpheme segmentation that works with unannotated text. Instead of minimum description lengths, it uses maximum a posteriori estimates (MAP).

Morfessor was invoked with the command

```
nice -19 perl morfessor1.0.perl -data lexicon-no
```

using perl 5.10.1 for Linux.

The lexicon file for Norwegian was a list of token frequencies. To reduce the influence of noise and also to minimize running time, the word-list was cleaned

---

[18] See http://linguistica.uchicago.edu/. The package used here was v4.0.2 for Linux, downloaded in October 2010.

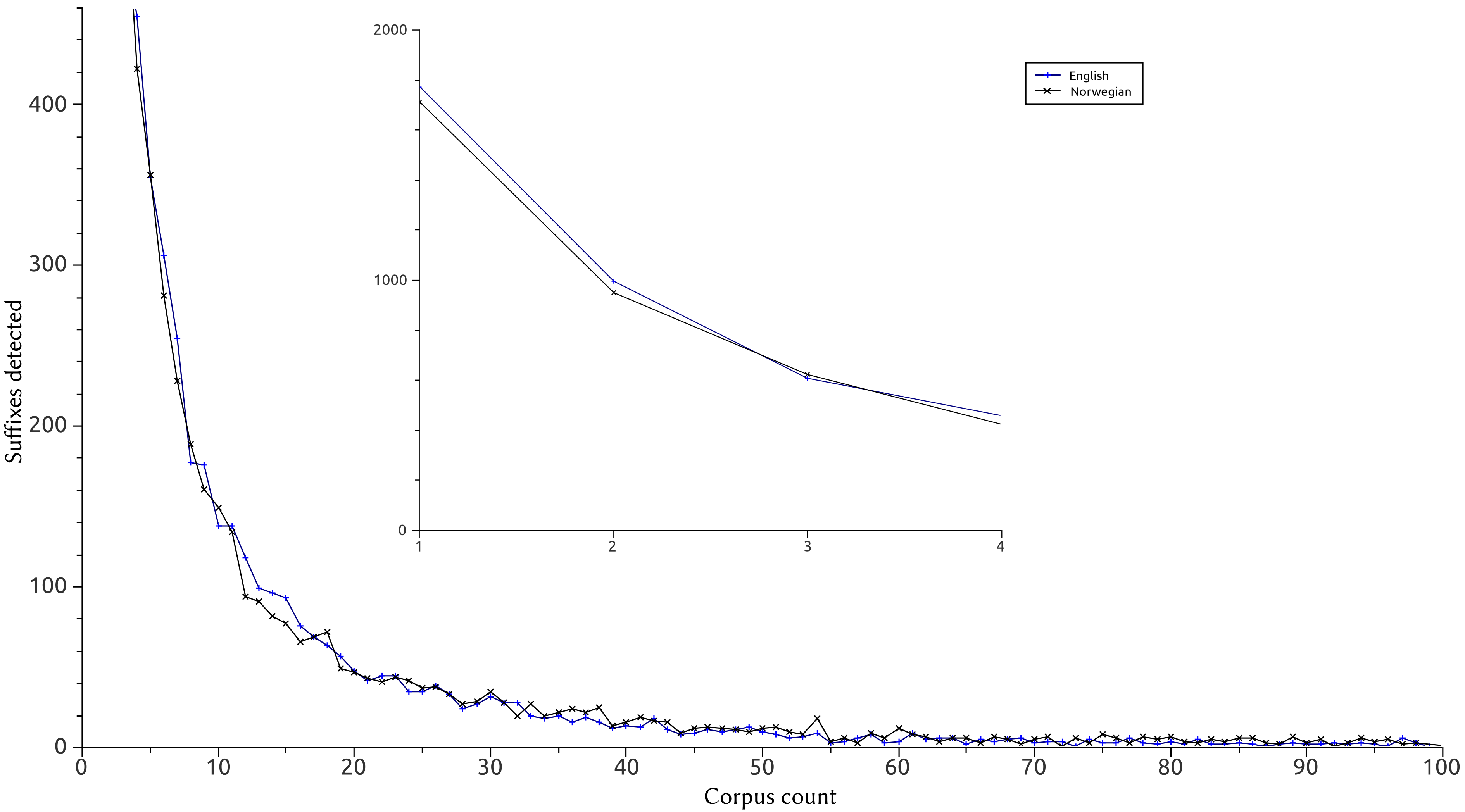

Figure 2: Suffixes found by Linguistica at different corpus counts (not cumulative).

of hapaxes and suspicious-looking tokens (ones that contained numbers, 'mixedCaps' and so on). Scrubbing the word list in this manner whittled it down from 2 231 227 tokens to 619 123, a reduction to 27.7% of the original size.

Unlike the other morphological packages investigated here, the model output by Morfessor shows the detected morphemes in context, which means the structure of compound words can be read directly from the data file (see example in Table 2). This has special value for dehyphenation, since the fact that a given word or stem often forms the first part of a compound can provide some much-needed negative evidence (in favour of deleting the hyphen).

The class `MorfessorDehyphen`, listed in section B.3, implements a simple way of using a Morfessor model. It picks out the stems that appear at the beginning of two or more compounds and uses them to gauge wether a string forms the first part of a compound; if it looks like it does, the class decides to delete the hyphen.

bedrageri + bestemmelsen
bedrageri + bestemmelsene
bedrageri + ene
bedrageri + er
bedrageri + et
bedrageri + et + s
bedrageri + handling
bedrageri + handlingen
bedrageri + sak
bedrageri + saker

Table 2: Compounds and inflections analyzed by Morfessor, formed from a stem meaning 'fraud'.

4.5 Affisix

Hlaváčová & Hrušecký 2008 introduces the affix recognition tool Affisix, which allows a user to assemble a custom prefix/suffix recognizer by freely combining its various functions, adjusting tolerance thresholds, and learning from a corpus.[19]

For the current purpose, the most relevant methods offered by Affisix are the ones based on the 'difference entropy', calculated by subtracting the entropy value of a segment from the entropy value of its preceding segment; that is, it singles out the segments where the growth in entropy is especially rapid. Since we're using it to look for suffixes, the measure we're primarily interested in is the one based on 'backward' entropy, where entropy is counted starting at the back of the word. The sites with the greatest positive or negative growth are selected as the most likely morpheme boundaries. There's also the additional requirement of two or more 'left-alternatives'; to be considered as a suffix, a substring must appear in at least two different contexts.

---

[19] Available from http://affisix.sf.net/. The release used here was v2.1.99.

elsesaktivitet
elsesaktiviteten
elsesalternativet
elsesanalyse
elsesanalysen
elsesansvar
elsesapparat
elsesapparatet
elsesarbeid
elsesarbeidet
elsesarena
elsesargumentet
elsesaspekt

Table 3: A bound morpheme, awkwardly sandwiched.

The invocation used for the Norwegian text was

```
nice -18 affisix --recognize suffix -i lex-no
-o affis-no.txt -c ’&(>(dbentr(i);0.25);>(lalt(i);2))’
-s ’fentr(i);bentr(i);dbentr(i)’ -v
```

—which results in the accumulation of suffixes shown in Fig. 3. The eye-catching drop-offs around 0.64 and 1.1 indicate bundles of suffixes that were detected at the exact same level of difference entropy.

Splitting morphs by this method causes two major problems: first, the algorithm finds the individual words of compound words as well as bound morphemes (something that can be corrected for by using a dictionary to filter out whole words). Second, the algorithm also finds bound morphemes that occur in the middle of a compound word, as shown in Table 3—here the bound morpheme *-else* comes from words like *bedervelses·aktivitet*. Trying to eliminate these occurences by filtering the word list with itself is somewhat more problematic, since we’re often dealing with very short strings which frequently appear as substrings without necessarily being proper morphemes.

In this case, we can spring for a two-pass method, using prefix recognition to find the sandwiched suffixes in the output from the first round of suffix recognition. (This kind of fishing around for substrings requires some tailoring to the specific language being processed, though this is still general enough that it might work on related languages, like Dutch.)

Now that we’re looking for prefixes, the interesting measure becomes not the backward but the forward difference entropy, which is the entropy growth-rate when moving to the right. For prefixes, the additional requirement is that the affix should have a minimum of two ‘right-alternatives’.

Affisix was invoked as follows:

```
    nice -18 affisix --recognize prefix -i affis-no.txt
-o sandwich-no.txt -c ’&(>(dfentr(i);0.25);>(ralt(i);2))’
-s ’fentr(i);bentr(i);dfentr(i)’ -v
```

The number of sandwiched suffixes found is shown in Figure 4.

When detecting sandwiched suffixes in the second pass, the linking elements or *Fugenmorpheme* between words in a compound also show up. This is immedi-

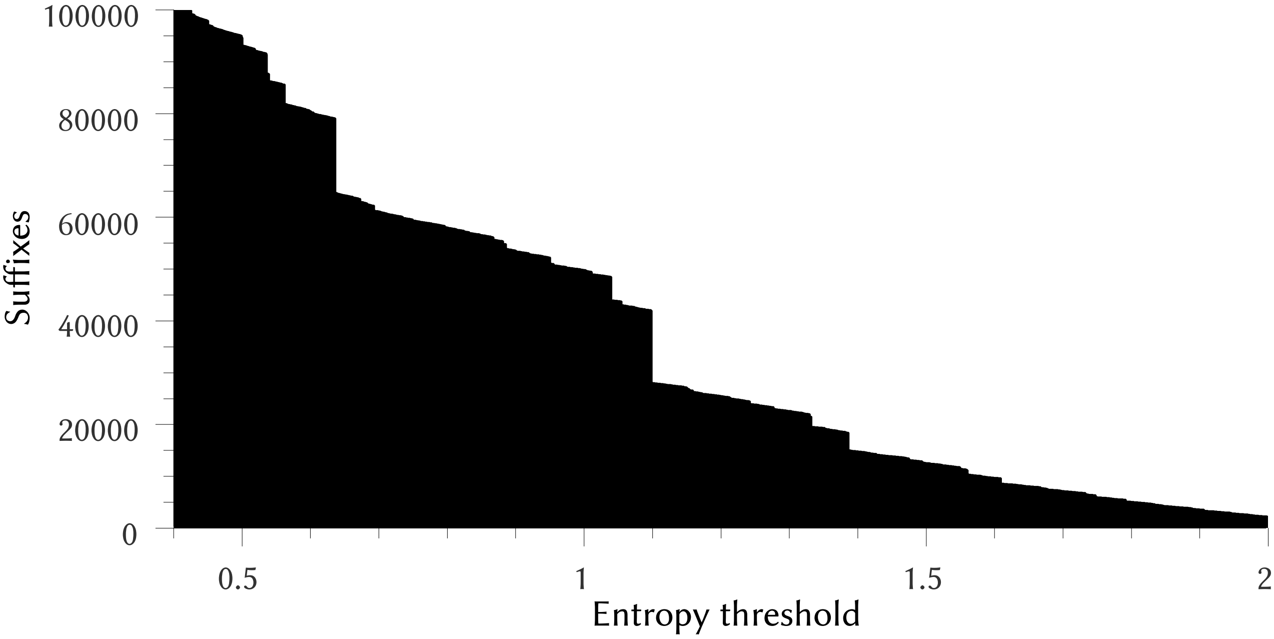

Figure 3: Suffixes found by Affisix as a function of the entropy threshold (Norwegian text).

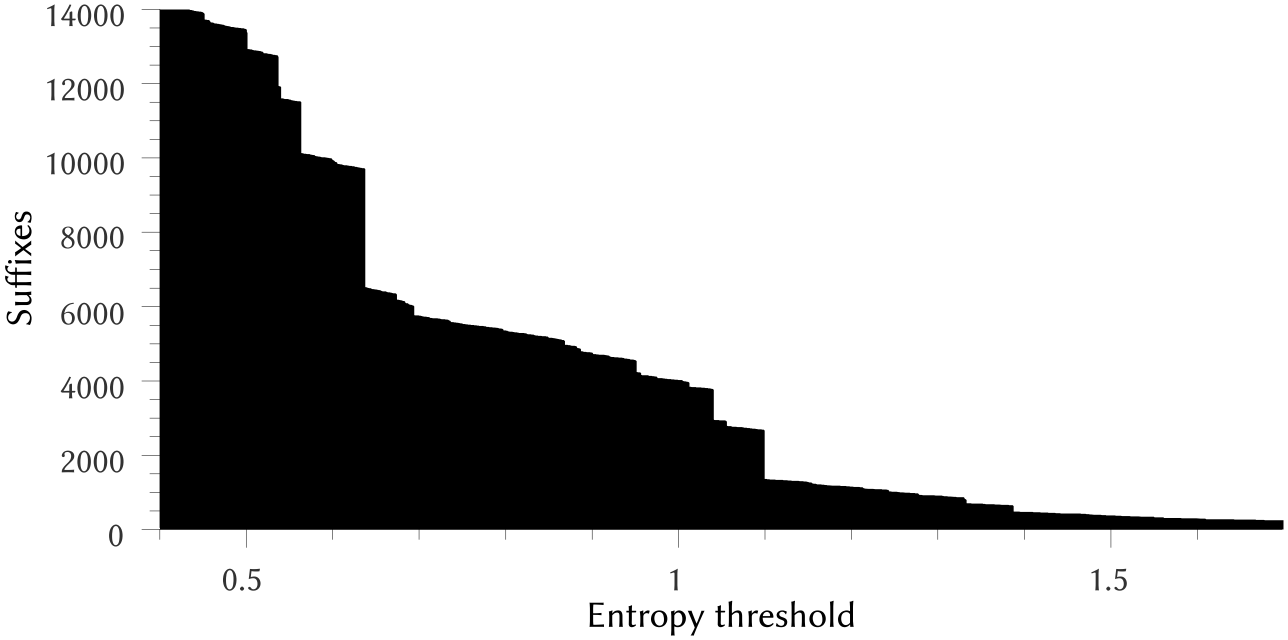

Figure 4: Sandwiched suffixes, amount detected as a function of the entropy threshold.

ately useful for dehyphenation, since a linking element that appears after a suffix never takes a hyphen (e.g., *-ings-*, *-elses-*), and in Norwegian this tends to happen in neighbourhoods where there are relatively few false friends—the only one that comes to mind for these examples is *blings*.

For the evaluation in section 6.6, the cutoff entropy threshold for both suffix- and sandwich-detection was set arbitrarily to 1.05, a value that was arrived at basically by eyeballing the graphs reproduced here.

A note on the convergence of entropy values

The 'continental shelves' that can be seen in figures 3 and 4 are eye-catching and somewhat peculiar; they appear where a large cluster of affixes is found at exactly the same entropy threshold. Manually inspecting the affixes found within these sudden growth spurts doesn't turn up an obvious difference between them and the affixes found at other thresholds, however. One possible explanation for this may come from the common assumption that natural language (and communication in general) tends towards a constant rate of entropy.[20] If this constancy applies down to the morphological–orthographical level, then chunks of letters might flock towards the same entropy values simply because this happens to be a naturally occurring property of language. An additional constraint is the narrow range of valid morphotactic transitions in a language.

Another thing to keep in mind when reading this data is that the measure used is *difference* entropy, that is, the result of subtracting one entropy value from the next; the same difference entropy can be derived from any number of absolute entropies. This means that the quantum leaps aren't due to common entropy values but rather common intervals *between* entropies.

Intriguing though it may look, investigating this in more detail here would be straying too far.

---

[20] See, for instance, Genzel and Charniak 2002.

CHAPTER 5

Supplemental methods

These methods are mostly stateless and have a fairly small impact; they were investigated mainly in the hope that they might shave off a few errors after the major approaches have run. They do not make many assumptions about the target language.

Jumping the gun a little, it's worth mentioning that in the final evaluation (section 6.6) most of these methods did not improve performance, with the exception of 'capitals' and the method described in section 5.3, 'Sandwiched function words'.

The source code implementing the stateless methods is collected in section B.7 on page 69.

5.1 Doubled consonant

Known affectionately as 'doublecons'.

For many languages, hyphenation is preferred between double consonants (as in *prof-fered, ap-plication, ak-kedere*). These are also loci where hyphens never occur naturally, which means that it's one of the surest indications of where it's safe to delete the hyphen.

The implementation of this method always returns **false** when there's an identical consonant on each side of the hyphen (c-c, d-d, f-f, g-g and so on).

In both English and Norwegian there are a few words where the double consonant should not be split, such as *engrossment* and *spissfindighet*, but joining one of these that has been incorrectly split (as *?engros-sment*) will never introduce an error.

5.2 Capitalization

If the capitalization of the text can be trusted to be mostly correct, the orthography may provide some clues for dehyphenation.

The simplest kind of orthographic rule is to look at a word's capitalization, checking wether it's in all-caps, mixedCaps, or has an initial capital letter. This is implemented as the 'capitals' function, and is similar to the method used in Grefenstette & Tapanainen 1994. Note, however, that this function differs somewhat from their Flex rule, which only inspected the single character before the hyphen-and-linebreak. The 'capitals' method which is implemented here considers every letter in a word.

5.3 Sandwiched function words

Nicknamed 'noncepart', this looks for strings from a dictionary in order to detect lexical hyphens within dephrasals such as *dyed-in-the-wool.* If a word is found on either side of the hyphen, the method returns **true**, indicating that the hyphen is lexical and should be kept.

This is strictly speaking language-dependent, since the dictionary will be specific to a language. However, given Zipf's Law, the collection of the most popular function-words in a language tends to be very small, to the point where it has little practical importance wether you compile the list by hand or collect the words automatically; and the longer dephrasals tend to contain a fair share of function-words.

Note that this method is somewhat sensitive to noise: for instance, if the dictionary is contaminated with the frequent misspelling '*alot' (contraction of *a lot*), then the algorithm could be led to the conclusion that the hyphenation *alot-*
♮*tment* contains a lexical hyphen.

The implementation in B.6 on p. 68 actually does not limit itself only to function-words, but uses the whole dictionary compiled from the NORA dataset. This failed to hurt performance very much in the final evaluation—adding the method barely raised the number of faults.

5.4 Rhyming compounds

This method looks for words like *helter-skelter* or *chaotic-determinstical*, where both parts end in the same substring. (Subtler rhymes like *pie-in-the-sky* are left as an exercise for the reader.)

The function only looks at the last two characters of each potential word-part, and, if they're identical, returns **true**. If no rhyme is found, the function tries stripping common suffixes (such as ’s’, ’es’, ’al’, ’ic’, and ’ical’) and goes again to see if its luck improves.

Looking at more than the last two characters is not currently implemented, as it has no impact on the decision; this would only be useful in a setup which returned a confidence score or something similar.

5.5 Word length

Returns **true** if either side of the hyphen is a string that is only one character long.

This catches ‘intialisms’ like *X-ray* and *level-9* which have been hyphenated incorrectly (a hyphenation is supposed to have a minimum of two characters before the linebreak and three characters following it).

5.6 Unimplemented possibilities

Hyphenated numbers

It’s extremely rare for numbers to be split across lines, but it can be found occasionally, for instance in early editions of Daniel Boorstin’s *The Image*. This phenomenon will not be treated here, as it never occurs in the NORA test set.

Verb-based compounds

Compound phrases ending in verbs, such as *system-powered* and *far-reaching*, are a relatively productive group in English. In the terminology of Manning & Schütze, these would be pre-modifier grouping hyphens.

The verb in these compounds is likely to be some sort of participle, which may be identified coarsely simply by looking for *-ed, -ing* suffixes. The parallel case in Norwegian is less problematic for purposes of dehyphenation, as compounds acting as verbs usually do not take a hyphen, and omitting the hyphen is our null hypothesis.

This possible method wasn’t prioritized since it’s a bit language-specific.

Syntactic analysis

Certain terms, like *hellige tre kongers-♮fest*[21] can only be recognized as the larger units that they are by using a more comprehensive analyzer that can work with both the syntactic and morphological level. Implementing this kind of comprehensive analyzer would go too far beyond the scope of this study.

---

[21] Literally ‘holy three-kings feast’, that is, the Epiphany. From Karianne Bjellås Gilje, ‘Stort nabolag’, in *Dagbladet,* 2 April 2011.

For English terms of this type, such as *film festival–audiences,* the attachment to the multi-word unit is supposed to be marked by using an en-dash instead of a hyphen. This, however, is a distinction rarely made by people who aren’t copyeditors, and is also quite likely to be lost in OCR.

CHAPTER 6

Experiments and evaluation

6.1 Replication of Grefenstette & Tapanainen

Since the tools used in their study—namely the Brown corpus, `nroff` and `flex`—are still in use, it was possible to attempt replicating the results of Grefenstette & Tapanainen 1994 using the GNU `troff` reimplementation, `groff`.

Grefenstette and Tapanainen report 101 860 lines of formatted text, of which 12 473 ended in hyphens. The 615 errors reported for G&T's baseline dehyphenation worked out to 4.9% of the hyphenated words. I was unable to come closer than 101 902 lines at a page width of `6.0535i`: of these, 21 151 lines ended in a hyphen. Applying the Flex pattern given in section 3.1 yielded 520 dehyphenation mistakes; several of the errors mentioned by G&T make an appearance (*rockcarved, satincovered*), while others do not (*science-fiction, rock-ribbed*), or are mangled in a different way (*rookieof-the-year, ring-around-therosie*).

Although 520 vs. 615 errors is only 15% off the mark, the proportion of errors in the replicated experiment works out to 520⁄21151 = 2.5% of the hyphenated words, which is only half of G&T's 4.9% (615⁄12473). The precise number of hyphenations may depend on very minute details of the `nroff` invocation and how the text is prepared, especially given that the Brown corpus is processed as a single continuous paragraph—this leads to a sort of cascading effect where the smallest change can get the snowball rollling, eventually causing a vast difference towards the end.

(Grefenstette & Tapanainen)

To the extent that the replicated experiment is reliable, it does indicate that the majority of the errors happen on *ad hoc* phrase-compounds such as *mailed-fist-in-velvet-glove* and *low-level*. When this is the case, it's only to be expected that the number of dehyphenation errors would be highly variable, given that the figure then depends mostly on how frequently these phrase-compounds happen to straddle the margin.

6.2 Preparing a sample

As far as I know, no gold standard for hyphenation has been assembled before. To generate one, 4 266 hyphenations were selected at random from the overall NORA test set, resulting in a mix of English and Norwegian plus the occasional Swedish word. Sampling was carried out by picking documents randomly, then collecting around 20% of the hyphenations from one document at a time by skipping ahead a random number of items.[22] This approach replicates in the sample something of the overall tendency that certain words tend to recur throughout a given document (typically because they relate to the topic under discussion; for example, in this document the words 'hyphenation' and 'dehyphenation' appear much more frequently than usual).

To produce a gold standard, each hyphen was judged by hand as either lexical, typographical, or part of a garbled non-word. Words which were nonsense or noise were deleted; in doubtful cases, I exercised personal judgement.

An important caveat here is that I do not have any experience in hyphenating text that's going to press. A better way of doing this might be to compare the judgement calls of several different typesetters, which would also allow identifying grey areas where judgement differs. These grey-area items basically constitute 'don't care' values which have no effect on precision scores, since either possibility could be said to be correct. Singling them out might be beneficial if a dehyphenation system approaches very high precision, but the methods evaluated below are likely to be coarse enough that it would have little effect.

6.3 Estimating overlap and ordering

When comparing the dehyphenation methods described in the previous two chapters, it can be difficult to estimate just how they will affect each others' performance, or even to what degree they will take an interest in the same words. One way to gauge this is to survey how much ground each method covers; in machine-learning terms, this is similar to their recall score. As with most things in dehyphenation, it's difficult to point to a single collection of text that can serve as a yardstick for everything else: as a result, the use of just one random sample from the NORA test set is more than a little impressionistic.

In the system used here, the string to be dehyphenated is sent through a preset chain of methods or 'deciders'. As graphed in Fig. 5 on page 47, a decider can settle on one of three return values: **true** for keeping the hyphen, **false** for deleting it, or **nil**, which works as a blank vote. When the result is nil, the main loop proceeds to consult the next method in the chain. If the program already is

---

[22] The skip-ahead was set to maximum ten items and minimum one, for an average of 1 in 5 lines; the script that was used for the sampling is included in section B.12 as `HyphenSample.rb`

at the last step in the chain, it defaults to false, which reflects that deleting the hyphen is the null hypothesis.

Since the bottom rung of the chain defaults to false, functions which only ever return false or nil (which includes all the methods from ch. 4 based on morphological analysis) can only be useful as a corrective—that is, they can only improve on the final result when placed before functions that produce too many false positives.

Method	**Coverage**	**Proportion**
Lexicog	894/4266	21%
Lingustica	1269/4266	30%
Morfessor	1600/4266	38%
affisandwich	506/4266	12%
affisuffix	1019/4266	24%
capitals	386/4266	9%
doublecons	301/4266	7%
noncepart	86/4266	2%
rhyming	97/4266	2%
wordlength	83/4266	2%

Table 4: Coverage of different dehyphenation methods.

The coverage scores for the various dehyphenation methods are listed in Table 4 above. ‘Coverage’ here means that the method settles on a decision, either true or false. For instance, the fact that the lexicographic method returns a decision on 894 items also implies that it returns nil for everything else, 3 372 items. The above data gives a rough idea of how many items are processed by each method, but, besides coverage, the other major issue is how the various methods will interact when they’re combined to form a hybrid-approach dehyphenation system. The scope of this problem can be gauged by looking at the amount of overlap between the sets of strings covered by the different methods. These intersection numbers, laid out in Table 5 on p. 44, roughly estimate just how much the different methods may step on each others’ toes. The percentages given are fractions of the entire sample: for example, the 7.7% overlap between ‘Morfessor’ and ‘Lexicog’ means that the lexicographic method and the Morfessor-based one can make conflicting decisions on at most 330 out of 4266 words in the sample.

The relative overlap between methods is mapped out in Table 6. Here, the denominator is constant when scanning columns downward and the numerator is constant when reading along a row. So looking up the ‘Morfessor’ row in the ‘affisandwich’ column gives the proportion of Morfessor’s overlap against affisandwich relative to the total coverage that affisandwich has, which is 363 / 506 = 71.74%. Swapping the row and column yields that overlap as a fraction of Morfessor’s much larger coverage, giving 363 / 1600 = 22.69%.

	Lexicog	Lingustica	Morfessor	affisandwich	affisuffix	capitals	doublecons	noncepart	rhyming	wordlength
Lexicog	20.96% (894)									
Lingustica	2.86% (122)	29.75% (1269)								
Morfessor	7.74% (330)	10.78% (460)	37.51% (1600)							
affisandwich	1.24% (53)	4.43% (189)	8.51% (363)	11.86% (506)						
affisuffix	6.68% (285)	9.19% (392)	11.67% (498)	5.70% (243)	23.89% (1019)					
capitals	7.31% (312)	0.75% (32)	0.33% (14)	0.02% (1)	2.79% (119)	9.05% (386)				
doublecons	0.42% (18)	2.16% (92)	3.19% (136)	0.66% (28)	0.94% (40)	0.00% (0)	7.06% (301)			
noncepart	0.82% (35)	0.26% (11)	0.33% (14)	0.16% (7)	0.38% (16)	0.45% (19)	0.14% (6)	2.02% (86)		
rhyming	1.01% (43)	0.40% (17)	1.10% (47)	0.26% (11)	0.75% (32)	0.14% (6)	0.21% (9)	0.05% (2)	2.27% (97)	
wordlength	1.34% (57)	0.40% (17)	0.14% (6)	0.05% (2)	0.42% (18)	1.17% (50)	0.02% (1)	0.09% (4)	0.00% (0)	1.95% (83)

Table 5: Overlap in coverage between the various dehyphenation methods (on a sample of 4266 mixed English/Norwegian words).

	Lexicog	Lingustica	Morfessor	affisandwich	affisuffix	capitals	doublecons	noncepart	rhyming	wordlength
Lexicog	**100% (894)**	9.61% (122)	20.62% (330)	10.47% (53)	27.97% (285)	80.83% (312)	5.98% (18)	40.70% (35)	44.33% (43)	68.67% (57)
Lingustica	13.65% (122)	**100% (1269)**	28.75% (460)	37.35% (189)	38.47% (392)	8.29% (32)	30.56% (92)	12.79% (11)	17.53% (17)	20.48% (17)
Morfessor	36.91% (330)	36.25% (460)	**100% (1600)**	71.74% (363)	48.87% (498)	3.63% (14)	45.18% (136)	16.28% (14)	48.45% (47)	7.23% (6)
affisandwich	5.93% (53)	14.89% (189)	22.69% (363)	**100% (506)**	23.85% (243)	0.26% (1)	9.30% (28)	8.14% (7)	11.34% (11)	2.41% (2)
affisuffix	31.88% (285)	30.89% (392)	31.12% (498)	48.02% (243)	**100% (1019)**	30.83% (119)	13.29% (40)	18.60% (16)	32.99% (32)	21.69% (18)
capitals	34.90% (312)	2.52% (32)	0.88% (14)	0.20% (1)	11.68% (119)	**100% (386)**	0.00% (0)	22.09% (19)	6.19% (6)	60.24% (50)
doublecons	2.01% (18)	7.25% (92)	8.50% (136)	5.53% (28)	3.93% (40)	0.00% (0)	**100% (301)**	6.98% (6)	9.28% (9)	1.20% (1)
noncepart	3.91% (35)	0.87% (11)	0.88% (14)	1.38% (7)	1.57% (16)	4.92% (19)	1.99% (6)	**100% (86)**	2.06% (2)	4.82% (4)
rhyming	4.81% (43)	1.34% (17)	2.94% (47)	2.17% (11)	3.14% (32)	1.55% (6)	2.99% (9)	2.33% (2)	**100% (97)**	0.00% (0)
wordlength	6.38% (57)	1.34% (17)	0.38% (6)	0.40% (2)	1.77% (18)	12.95% (50)	0.33% (1)	4.65% (4)	0.00% (0)	**100% (83)**

Table 6: Relative overlap in coverage between the dehyphenation methods.

Another way to read the data is to see the numbers in Table 5 as the size of the intersection of strings covered by both the methods in row A and column B, i.e. A ∩ B. Then, the percentages in Table 6 at row A and column B gives the probability of getting 'true' or 'false' out of method A given that this string already provoked a true-or-false from method B, that is, $P(B|A)$.

From the data presented in these tables, we can see that there's a relatively low degree of interaction between the different approaches; this lets us make an independence assumption, judging the methods mostly in terms of how they affect the final result.

6.4 Initial evaluation

To compare the lexicographic algorithm with the stateless method used by Grefenstette & Tapanainen, I first adapted the hyphenation 'gold standard' described earlier in this chapter: after discarding duplicate items, 3 634 hyphenations remained.

The lexicographic algorithm was trained on the entire NORA dataset and then compared with the 'G&T' reimplementation, yielding these error rates:

Lexicographic method	403⁄3634 = 11.1%
Grefenstette & Tapanainen	601⁄3634 = 16.5%

There were 350 hyphenations which neither algorithm got right; 53 items were only correctly dehyphenated by the G&T algorithm (these were typically compounds containing acronyms, such as *S-kortisol* and *AM-radioen*).

For the lexicographic algorithm, an error rate around 10% is only to be expected given that it's limited by the amount of hapaxes in the data (previously described in section 4.1). This in turn raises the question of wether the empirical algorithm will do worse on a smaller dataset. To test this, I ran a follow-up experiment where the empirical algorithm was trained on just the hyphenated version of the Brown corpus that was prepared in section 6.1. Performance was then compared against the reimplemented version of the G&T algorithm, resulting in these error rates (note that since duplicates were removed, this is per type, not per token):

Lexicographic method	277⁄11372 = 2.4%
Grefenstette & Tapanainen	423⁄11372 = 3.7%

In this run, there were 273 hyphenations which neither algorithm got right, and just 4 items where only the G&T reimplementation succeeded.

These results should go towards showing that the empirical word-list approach can give competitive results even when working with a relatively small collection of words—in this last run, 1 million tokens. It also demonstrates that

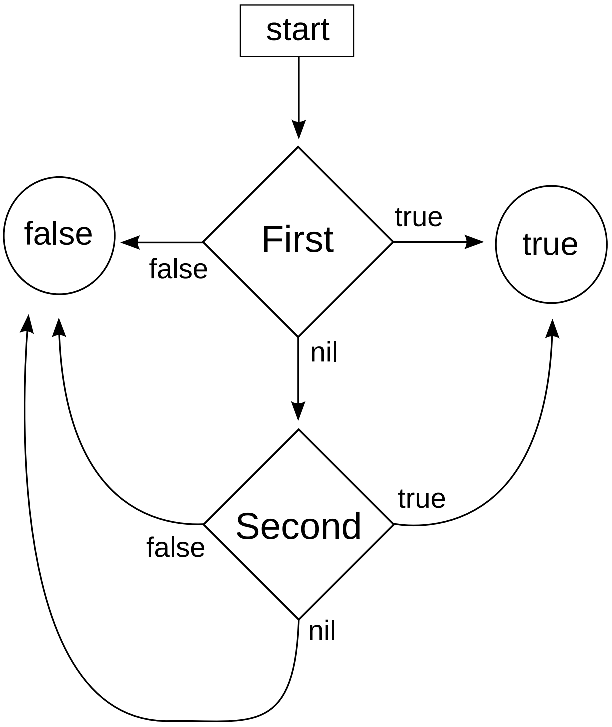

Figure 5: Flowchart for an example pipeline containing just two methods. A return value of true results in keeping the hyphen, false in deleting it.

even though the precise error rate for dehyphenation will show considerable variation depending on circumstances, it's still possible to see broad trends in the relative performance of different methods.

6.5 Experimental setup

It's not especially realistic to evaluate procedures in isolation: a practical implementation is free to use any eclectic combination of methods as long as it gets results. With this in mind, the dehyphenation classes were written in a modular way so they could be chained together to form a cascade or pipeline. To facilitate chaining, each method can return any one of **true**, **false** or **nil**. A return value of nil means 'no decision' and signals that the queue should proceed to the next method in the chain; a state diagram of the possible progressions is shown in Figure 5. In the event that every method returns nil, the cascade defaults to the null hypothesis, returning false to signal that the hyphen should be deleted.

As outlined in section 6.2, the sample of 4 266 hyphenations was prepared by selecting Norwegian and English documents at random from the NORA test set, then picking around 20% of the hyphenated words.

The ‘bynames’ for the various dehyphenation methods are the ones used internally by the evaluation pipeline. They correspond to detailed descriptions in sections throughout this document as laid out in the table directly below.

Method	Section	Page
Lexicog	4.1	28
Lingustica	4.3	31
Morfessor	4.4	31
affisandwich	4.5	33
affisuffix	4.5	34
capitals	5.2	38
doublecons	5.1	37
noncepart	5.3	38
rhyming	5.4	38
wordlength	5.5	39

6.6 Final evaluation

In the current implementation the methods based on morphological analysis only return false or nil, never true; therefore, most of their verdicts are identical with the null hypothesis which serves as the default at the bottom of the cascade. Subsequently, if the morphological methods are placed on the bottom rung they blend in completely with the background, causing no appreciable difference in the final result. Since it follows that the morphological modules are going to have to be added on top of others, the first methods placed into the experimental setup become the lexicographic and stateless ones, by default.

The major stateless contender is ‘capitals’, which examines only the capitalization; note that this method is quite similar to the one used by Grefenstette & Tapanainen (section 3.1), with the added tweak that this module examines the whole word and not just the letter on either side of the hyphen. Results of evaluating on the sample prepared earlier are listed in Table 7.

The item ‘none’ at the top shows the result of always deleting the hyphen, ‘capitals’ is the stateless method similar to Grefenstette & Tapanainen 1994, and ‘lexicog’ is the lexicographic method described in section 4.1. Several methods may fire in sequence: ‘capitals + lexicog’ means that ‘capitals’ was run first, followed by the lexicographic method. The combination capitals+lexicog produces an error rate of only 8.26%, which becomes the figure to beat.

In the second round, it becomes obvious that simply plugging in morphological analyzers upstream of the lexicographic method gives a sharp increase in the number of errors. An alternate approach that might correct for this is to divide the lexicographical method into *two passes,* with a first pass that ignores the words and quasi-words at frequency 1, which is where most of the noise in the

Module sequence	Misses	Faults	Error rate
none	1358	0	30.55%
capitals	875	3	21.32%
lexicog	387	23	9.96%
capitals + lexicog	315	25	**8.26%**
affisix + lexicog	251	420	16.29%
linguistica + lexicog	323	199	12.68%
morfessor + lexicog	234	480	17.34%
affisandwich + lexicog	331	126	11.10%
lexicog-nohapax + capitals	378	24	9.76%
lexicog-nohapax + capitals + lexicog	315	25	**8.26%**
the above + morphological analysis...			
affisix	194	166	8.74%
linguistica	252	97	8.47%
morfessor	165	204	8.96%
affisandwich	260	87	8.43%
or orthographical analysis...			
wordlen	310	35	8.38%
doublecons	299	42	8.28%
rhyming	303	67	8.98%
noncepart	290	30	7.77%

Table 7: Performance of different constellations of methods (on the sample of 4 266 English/Norwegian words). 'Misses' are items where the hyphen should have been retained, but every method in the queue returned nil. A 'fault' means that the pipeline settled on the wrong decision, wether true or false. The total number of errors is the sum of 'misses' and 'faults'.

corpus comes from. When placed after this first pass, the morphological methods might possibly act to block out some of the noise which comes from the hapaxes.

To be on the safe side, we first have to verify that discounting hapaxes does not actually wind up improving performance (and it doesn't). Second, we have to make sure that springing for two passes of lexicography doesn't upset things too much otherwise: running a test with a pipeline of 'lexicog-nohapax⟶capitals ⟶ lexicog' gives a reassuring result with exactly the same number of errors as before. Below this point in the table, the morphological method up for testing is run before the final lexicog-with-hapaxes—so, for example, the test with Affisix is queued up as 'lexicog-nohapax⟶capitals⟶affisix⟶lexicog'. Used this way, the morphological methods don't cause nearly as much trouble, but they still fail to improve on the simple combination capitals+lexicog.[23]

---

[23] Note that, after nohapax and capitals have run, there are only 24 outright faults left that the morphological methods could possibly correct, leaving them very little to work with.

Next up are the small-coverage stateless methods which mostly look at orthography; all of them introduce as many errors as they correct. They would seem not to be worth including in dehyphenation queues except under special circumstances.

Finally, adding the 'noncepart' method which tries to detect ad-hoc phrasal compounds like *out-of-order* actually gives a slight increase in accuracy.

Lexicography with negative verdicts

The lexicographic methods tested above only return true or nil, never false; if the more popular wordform lacks a hyphen, the algorithm simply returns nil to signal 'no decision' (rather than giving a decision to delete the hyphen, as could be expected). There's no obvious reason why this should be so, since the frequency dictionary enables making negative verdicts that are as definitive as the positive ones.

For the sake of completeness, a version which also returns false is tested below, with and without being augmented by 'capitals'. It's easy to see that performance actually deteriorates slightly when this feature is added, but at the moment I'm unable to explain why exactly this happens.

Variant	–	+ capitals
nohapax	12.24%	9.91%
lexicog	9.96%	8.40%

Table 8: Error rates for a lexicographic method which returns both true and false.

CHAPTER 7

Conclusion

7.1 Recommendations

On the sample used in section 6.6, going from the null hypothesis to the lexicographic method cut the error rate in half. This would indicate that even this very crude approach to lexicography can do a lot for dehyphenation.

In the interests of fairness, it should be noted that the data used for testing may have been slightly biased against the null hypothesis, favouring methods which lean towards returning true (like the lexicographic one). A lot of the documents in the dataset happened to be typeset 'ragged-right', and so were only hyphenated along lexical hyphens. This still counts as a realistic test, however, since the material was a representative collection of real academic texts of the kind which is a prime candidate for dehyphenation.

7.2 Morphology in dehyphenation

For the specific gold standard used here, none of the morphological methods improved on the simple lexicographic method. More careful attention to morphology might have yielded better results, but the salient point here was to investigate the sort of performance that could be achieved with relatively hands-off automatic analysis: dehyphenation of documents will probably only get included as an afterthought, in real-world systems, which makes it unlikely that anyone will spend much time on finer points such as tuning of language-specific parameters.

On the other hand, when run after the lexicographic method without hapaxes several of the morphological methods didn't make things much worse, either: this offers some hope that morphology might improve performance when dehyphenating languages that are morphologically hyperactive, such as Finnish.

Note that the results from evaluating the morphological methods in the previous chapter are not useful as a 'bake-off' type competitive comparison of the morphological packages themselves—each one was used for a fairly different purpose, mostly depending on what kind of information was the least trouble to dig out of their output. This difference can easily be seen in the lack of overlap between the different morphological methods, laid out in Table 5 and 6.

7.3 Directions for future work

The 'gold standard' prepared here can only provide a fairly coarse measure of dehyphenation performance: besides the fact that it contains at least one outright error, there's also the problem that it only works with black-or-white classifications. Each item must be judged as either a lexical or an accidental hyphen, with no room for doubt. As mentioned at the beginning of chapter 6, a more sophisticated way of doing this might involve pooling the judgements of several people (preferably experienced proofreaders). With a range of opinion, it would be possible to identify statistically the grey areas where judgement is essentially arbitrary and either decision would be acceptable.

The other major issue for dehyphenation is that error rates can be seen to fluctuate wildly, being highly dependent on the material. For estimating the variation in the error rate, it's possible to use statistical techniques such as resampling or cross-validation, giving variance scores which indicate how sturdy the results are. This was not done here, partly due to time constraints and partly because including these scores would have made the results more difficult to understand.

7.4 Improving robustness

The lexicographic method developed here should already be fairly robust even while processing several languages mixed together.

When dehyphenating collections of documents that are especially noisy, it's worth noting that modifying the lexicographic method to ignore hapaxes resulted in just a slight dip in performance (in Table 7, this is the run labeled 'lexicog-nohapax + capitals'). Accordingly, lexicography might provide acceptable results even when processing text which is so noisy the hapaxes can't be trusted at all.

7.5 Improving efficiency

In the current implementation, the frequency word-list is stored straightforwardly in a hash table. After gathering words from 1.5 GB of text, the table grows to around 80 MB: although this poses no problem on most current platforms, in situations where space is tight the hash could be replaced by a Bloom filter.

Bloom filters are bit-arrays of superimposed hashed values, providing a compact data structure to check for the existence of a string. The tradeoff is that they're prone to producing false positives (a risk which can be alleviated by using a larger array). The filters have the additional property that they never produce a false negative, making them especially well-suited to dictionaries; however, unlike hash tables they can't be used for storing frequencies, and are only useful to test for the existence of a particular string.

As mentioned in the description of the two-pass lexicographic approach on p. 48, the most important distinction in the frequency dictionary is between the uncertain hapaxes, with a frequency of one, and the more certain words which occur at least twice. This distinction can in fact be maintained without the upkeep of a full frequency word-list—simply keep one Bloom filter with hapaxes and use a separate Bloom filter to hold the words that have frequency ≥ 2.

A drawback of using Bloom filters instead of hashes is that an item can't be deleted, which means that if faulty words are put into the data structure the only way to correct it is to recompute the whole thing. Additionally, since a word can't be deleted from the 'hapax' filter it becomes necessary to check both filters to see if a word is a hapax or not. Since a Bloom filter never produces a false negative and its false positives are distributed randomly, this 'dual-Bloom' approach is relatively robust.

The possibility of false positives is a calculated risk when using Bloom filters. In the scheme outlined here, it would make sense to keep a larger filter for hapaxes if most words are counted as a hapax as some point; when the 'hapax' and 'twoplus' filters are of unequal size, this would make it less likely for the filters to produce identical errors, reducing the risk of a false positive on both ends. The truth-value table below also shows off the possibility for built-in error correction through the 'impossible state' where a word matches in the 'twoplus' filter but not the 'hapax' filter. In this case, the frequency would be set to zero.

hapax	twoplus	frequency
false	false	0
true	false	1
true	true	≥ 2
false	true	(error)

Table 9: Possible truth values and corresponding word frequencies.

APPENDIX A

Noise reduction

> PostScript makes no distinction between text and graphics.
>
> —*PostScript Language Tutorial and Cookbook*

PostScript, and by extension, PDF, allows considerable freedom in how a string is printed. Crucially for purposes of plain-text extraction, this means that any given file is under no obligation to store a recognizable SPACE character; being defined purely negatively in terms of blank space relative to its surrounding characters, this can make it a difficult task to decide what exactly counts as a word division on the page. Because of this, even plaintext extracted from files that haven't gone through OCR may feature entire lines without spaces, as in

> PostScript(anditsoffspring,PDF)allowsconsiderablefreedominhowastringis
> printed.Cruciallyforpresentpurposes,thismeansthatanygivenfileisundernoobli-

This is a major source of noise in the NORA test set, affecting up to 1% of the documents. It's especially troublesome when trying to construct a dictionary from the text, since the far end of the word-length spectrum gets saddled with a lot of long strings that aren't properly compounds.

Two strategies for identifying noisy documents immediately present themselves: one is to examine average word length and single out the documents where it deviates significantly from the norm; the other, to look for function words that intrude on other words (as in 'means**thatany**givenfile**is**under'). Both of these strategies have their problems. The trouble with going by word length is that there is no magical threshold which reliably separates the noise from the authentic words, and the problem with going by function words is that there is a high chance for false positives, e.g., mistaking the word *into* for a corruption of the syntactic construction 'in to'.

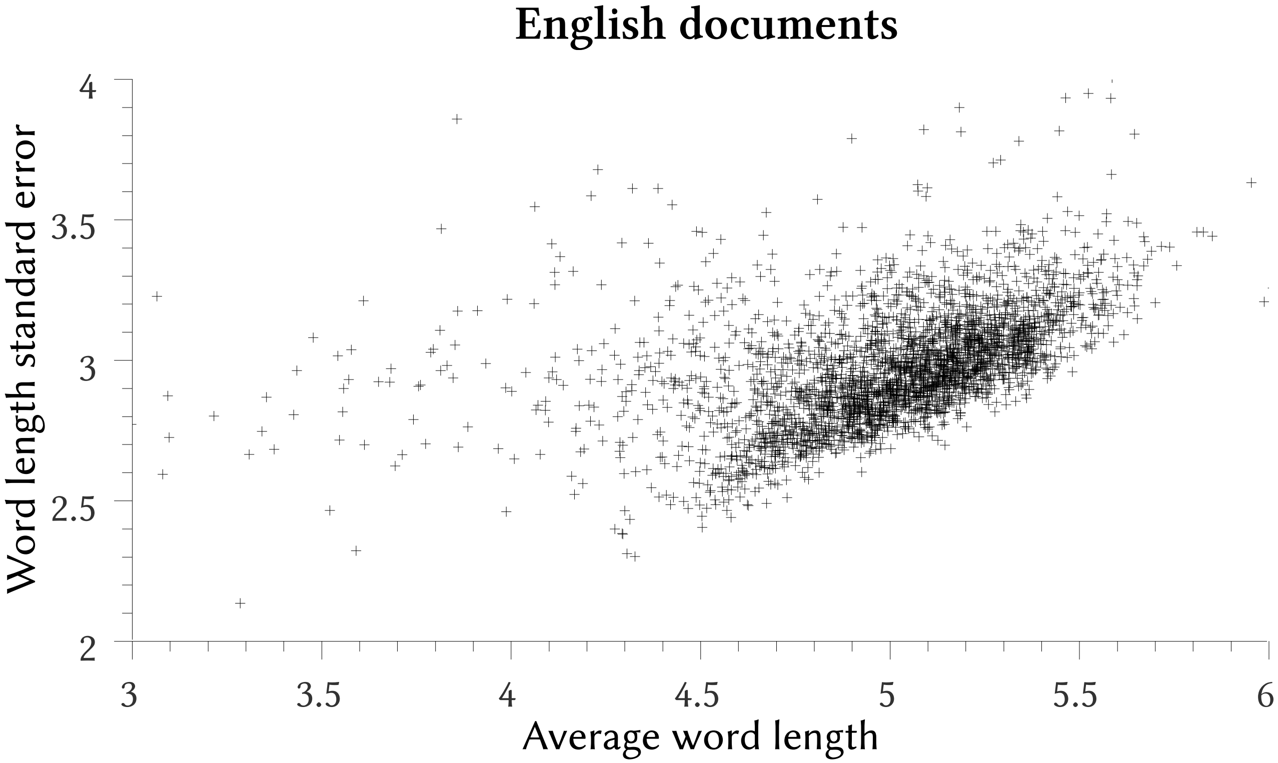

Figure 6: Average word length vs. standard error (variance) of wordlengths for the 3 080 English documents. Some outliers not shown.

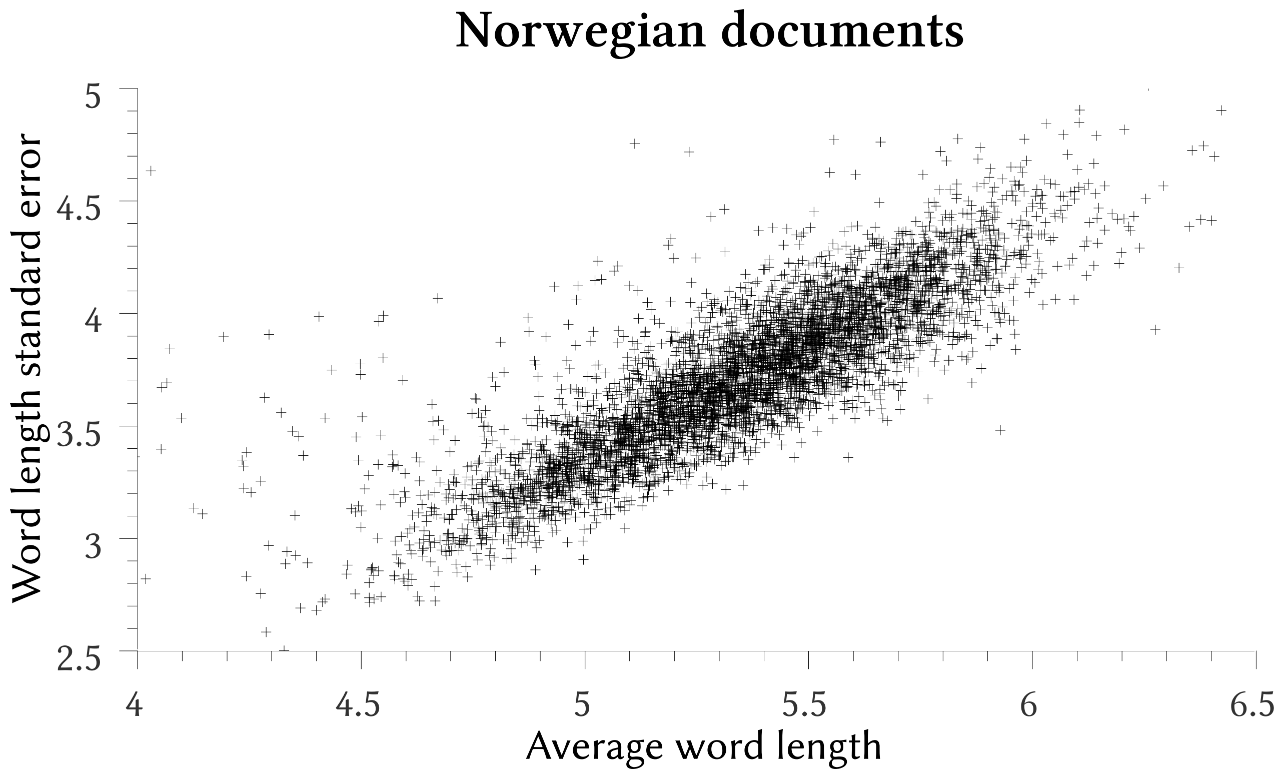

Figure 7: Average word length vs. standard error of wordlengths for the 5 296 Norwegian documents. Some outliers not shown.

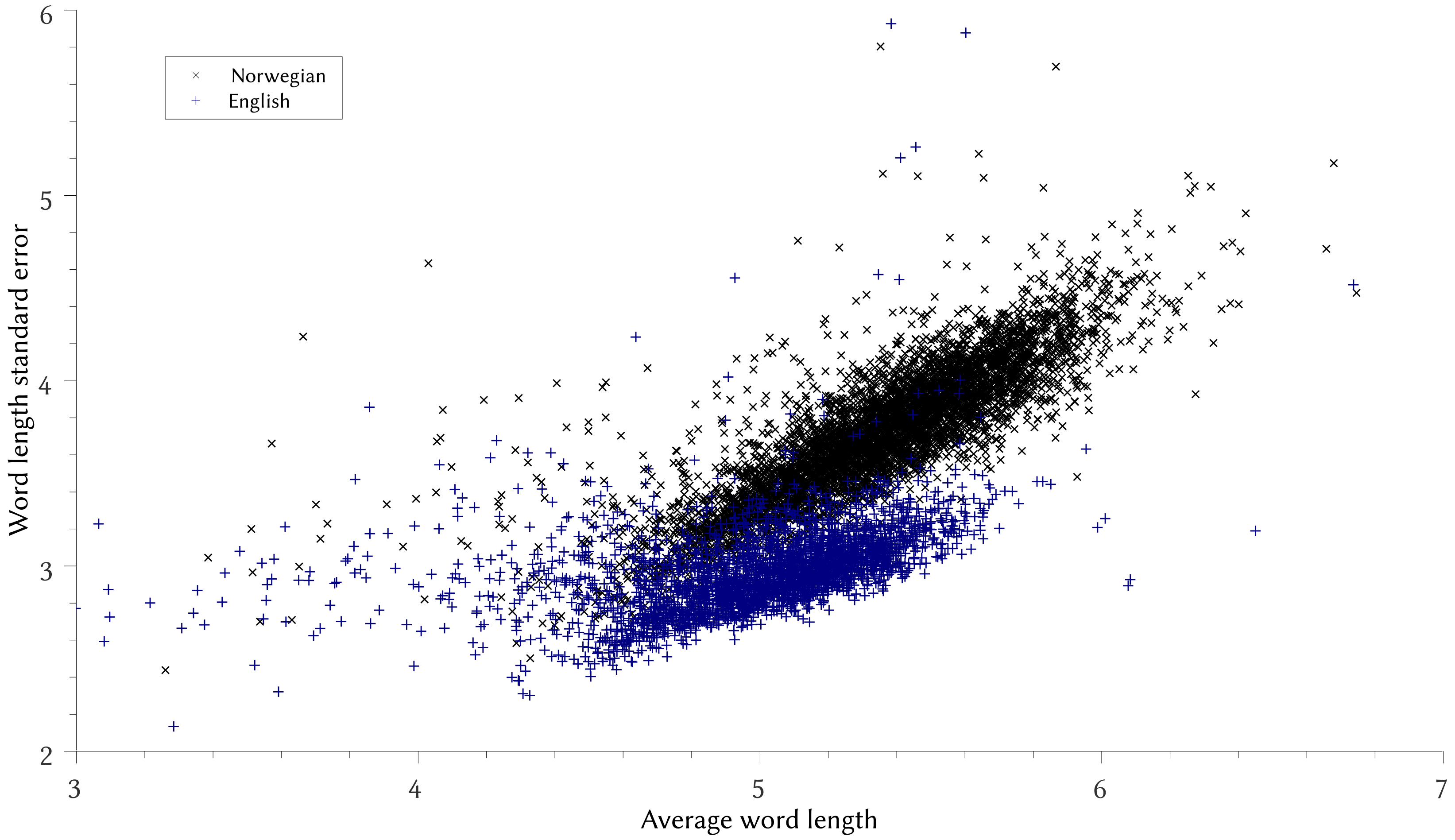

Figure 8: Distribution of Norwegian and English documents, in the same coordinate system.

A.1 Variance of word length and frequency

Examining the length of words has the advantage that it's one of the easiest things to measure in a text. Figure 8 shows the relation between average word length and the corresponding standard deviation for the separate languages in the test set: the correlation between the two is readily apparent, especially for Norwegian. This fact is unsurprising, given Zipf's law: since a large proportion of the words in a normal text always will be the short function words, a text with more and longer compound words will also have larger variance—the function words pin down the short end of the scale, so to speak.

The overlap between the languages in the low range is unproblematic, since it's relatively easy to guess the language from the stop-words in the text. Within each language, measuring the ratio between word length and variance allows us to identify the outlier texts which are most likely to contain a high proportion of noise. It's especially useful for finding documents which are missing a lot of spaces—in these cases, the function words are run together with everything else, and this shows up in an unusual proportion between the mean word length and the variance of word lengths.

With this scheme, marking documents for clean-up is just a matter of calculating a regression line and adjusting the tolerance threshold for how far a document is allowed to stray from the typical wordlength/variance ratio. Since the correlation betwen average word length and its variance is stronger in Norwegian, it may be the case that this measure is more precise for languages where compounds tend to be written flush.

A.2 Stop word detection

Function words in a compound usually mark it as dephrasal; in both English and Norwegian, the glue-words have to be separated by hyphens, as in *well-to-do, man-in-the-middle, flue-på-veggen, i-og-for-seg*. This is precisely because writing them flush without hyphens is too likely to confuse a dephrasal compound with a regular compound; accordingly, lexicalization of these items seems to happen at a very slow pace (although it does happen, as with *to-day* and *to-morrow*).

The upshot of this is that function words are surrounded either by spaces, or by hyphens. If what looks like a lot of function words are found written flush, as in 'topoftheshelf', this is a good sign that the document in question is missing a lot of spaces. This creates an opportunity for distinguishing between syntactic and morphological combination without getting into an entire system for deep analysis, since syntactic combinations are likely to have a much higher content of function words, combining more freely than they do in morphology.

Function words that are only a single letter are the least useful here, as the occurrence of a single letter is fairly uninformative. More generally, we can estimate how well a given word indicates missing spaces by how rarely it occurs as a substring of other words: for example, ‘of’ is a substring of ‘often’, making it a relatively poor indicator.

A.3 Mangled ligatures

Another source of noise are typographical ligatures such as *fi, fl, ffi,* etc.; these *presentation forms*, as Unicode calls them, often go unrecognized by PDF-to-plaintext conversion software. (The reach of this problem can easily be seen by running an exact-phrase web search on a word that’s missing a ligature, such as "de nitely".) Especially in documents that have been prepared with TEX, it’s difficult if not impossible to predict exactly which glyph maps to which ligature.

To repair mangled ligatures in the NORA test set, however, I had the luxury of being able to make assumptions about the language of a document (it was either English or Norwegian). For each of the two languages, a dictionary was prepared of words which contain ligatures (*de**fi**nitely, e**ffi**cient* and so on); the unrecognized glyph that appeared most often in a likely position was chosen as the missing ligature.

This method succeeded in repairing around a hundred-odd documents, at the cost of a few false positives; in one case the program incorrectly assumed that the Polish Ł (‘L with stroke’) was the ‘fi’-ligature, which goes to show that the method can’t be used for blanket application across all languages.

The implementation simply stores the relevant words in a hash; this means that any one lookup happens in O(1) constant time. When there’s only one lookup per word in the text, this guarantees O(n) performance on a text with n words.

Note that since the ligatures of the documents themselves couldn’t be trusted, the word list for this task had to be assembled from other sources. For English, I used the Linux version of `/usr/dict/words`.

APPENDIX B

Program listings

The following code requires Ruby 1.9 for Unicode support; it assumes that input files are in UTF-8 or ASCII.

The classes which do the actual dehyphenation have an extremely simple interface:

- A constructor. If applicable, the single argument is usually a filename to read data from.
- A method `decide?(before,after)`, which takes the word-fragments from before and after the linebreak and returns either true (keep the hyphen), false (delete the hyphen), or nil (undecided).

Note that using copy-and-paste from the PDF version of this document may cause errors due to issues with character sets; see the HTML version for links to the original source files.

B.1 Evaluating different pipelines

This is the script used for the final evaluation in section 6.6, demonstrating how all the dehyphenation classes were used. The class DeciderQueue is listed in section B.2 on p. 64.

File `EvalMethods.rb`:

```
# -*- coding: UTF-8 -*-

require './DeciderQueue.rb'
require './StatelessDehyphen.rb'
require './MorphoData.rb'
require './MorfessorDehyphen.rb'
require './AffisixDehyphen.rb'
require './HyphenStat.rb'
require './NoncePart.rb'
require './RhymingDehyphen.rb'

pipe = DeciderQueue.new

#~ affixguesser = HyphenedAffixGuesser.new(5)
#~ affixguesser.load_tsv('words-all.txt')
linguistica = MorphoData.new(2)
linguistica.readSuffixes('nosuffixes2.txt')
morfessor = MorfessorDehyphen.new('no01.morfessor')
affisuffix = AffisixSuffix.new('affis-suffix-no.txt')
affisuffix.threshold = 1.05
affisandwich = AffisixSandwich.new('affis-sandwich-no.txt')
affisandwich.threshold = 1.05
rhyming = RhymingDehyphen.new(['e', 'er', 'en', 'sk', 'ske', 'ig', 'ær'])
lexicog = HyphenStat.new
lexicog.loadwords('words-all.txt')
noncepart = NoncePartDehyphen.new(lexicog)
# ^ taps into the dictionary loaded with HyphenStat


# ---- uncomment/rearrange lines below to evaluate different methods ----

# pipe.add() { |before, after| affixguesser.decide?(before,after) }

pipe.add() { |before, after| lexicog.decide_freq(before, after, 2) }
 # lexicography without hapaxes

pipe.add() { |before, after| CapitalsDehyphen.decide?(before,after) }
# pipe.add() { |before, after| WordLengthDehyphen.decide?(before,after) }
# pipe.add() { |b, a| DoubleConsonantDehyphen.decide?(b,a) }
 # pipe.add() { |before, after| rhyming.decide?(before,after) }
# pipe.add() { |before, after| morfessor.decide?(before,after) }
# pipe.add() { |before, after| affisuffix.decide?(before,after) }
# pipe.add() { |before, after| affisandwich.decide?(before,after) }
# pipe.add() { |before, after| linguistica.decide?(before,after) }

# pipe.add() { |before, after| lexicog.decide?(before,after) }

 #pipe.add() { |before, after| noncepart.decide?(before,after) }
```

```
misses = []
faults = []
correct = 0
total = 0
File.open('dehyphend1.txt', "r:UTF-8") do |borkfile|
  borkfile.each_line do |line|
    parts = line.chomp.split("\u{00ad}")
    if parts.length==2
      total += 1
      before = parts[0].sub(/-$/, '')
      after = parts[1]
      result = pipe.run(before, after)
      hyphenated = (parts[0] =~ /-$/)
      if not result.nil?
        if (result and !hyphenated) or
          ((!result) and hyphenated) then
          faults.push(parts[0]+"."+parts[1])
        else
          correct += 1
        end
      elsif (result and !hyphenated) or
          ((!result) and hyphenated) then
        misses.push(line.chomp)
      end

    end
  end
end

overall = misses.length+faults.length

print faults, " = ", faults.length, " faults\n"
print "Misses: ", misses.length
print " . Correct: ", correct
print " . Overall ", sprintf("%.2f", (100*overall).to_f/total), "% wrong"

print "\n"
```

B.2 Dehyphenation queue

This class queues up dehyphenation classes and evaluates them, following the state diagram in Fig. 5 on page 47.

File `DeciderQueue.rb`:

```ruby
# -*- coding: UTF-8 -*-

class DeciderQueue
  def initialize()
    @queue = []
  end

  def add(&block)
    block.kind_of? Proc or raise ArgumentError, "Code block required"
    @queue.push(block)
  end

  # returns the result of the first block in the queue
  # that returns something different than nil
  def run(*input)
    ret = nil
    @queue.each do |block|
      ret = block.call(*input)
      break if ret != nil
    end
    ret
  end

end
```

B.3 Morfessor-based method

The constructor for this class takes the name of a file which is the output from a run of Morfessor 1.0.

File `MorfessorDehyphen.rb`:

```ruby
# -*- coding: UTF-8 -*-

class MorfessorDehyphen
  def initialize(filename)
    @stems = Hash.new
    rawdata = Hash.new
    # rawdata is only used while reading the Morfessor model;
    #  @stems is the sorted dictionary that sees further use

    filename.kind_of? String or raise ArgumentError
    File.open(filename, "r:UTF-8") do |infile|
      item = ""; parts = []
      infile.each_line do |line|
        item = line.gsub(/^\d+\s+/, ’’)
        item.chomp!
        parts = item.split(’ + ’)
        next if parts.length < 2  # unanalyzed item
        if parts[0].length > 2
          # not interested in tiny prefixes here
          if rawdata[parts[0]]
            @stems[parts[0]] = true
            # we’ve seen the stem before, which means it’s reliable
          else
            rawdata[parts[0]] = true
            # this is the first time we’ve seen this stem
          end
        end
      end
    end
  end

  def decide?(before, after)
    if @stems[before]
      return false
    else
      return nil
    end
  end
end
```

B.4 Linguistica-based method

The constructor for this class takes an optional tolerance threshold, where higher values are more restrictive: see section 4.3 for details. To load data into an instance of the class, use the method `readSuffixes`.

Note that Linguistica outputs its results in UTF-16 format, which needs to be recoded into UTF-8 before it can be read by this class.

File `MorphoData.rb`:

```
# -*- coding: UTF-8 -*-
```

```
def readfields(filename)
  File.open(filename, "r:UTF-8") do |fh|
    fh.each_line do |line|
      fields = line.split(/\s+/)
      if fields.length > 0
        yield fields
      end
    end
  end
end

class MorphoData
  def initialize(cutoff=0)
    @cutoff = cutoff
    @suffixes = Hash.new(0)
  end

  def readSuffixes(filename)
    readfields(filename) do |fields|
      next if fields.length != 5
      @suffixes[fields[1]] = fields[3].to_i
      # The statistic we're storing is the 'corpus count',
      #   not the use count (the latter is always lower).
    end
  end

  def decide?(before, after)
    if @suffixes[after] > @cutoff
      return false
    else
      return nil
    end
  end

end
```

B.5 Affisix-based methods

These classes implement both the suffix detector and the sandwiched-suffix detector.

File `AffisixDehyphen.rb`:

```
# -*- coding: UTF-8 -*-

class AffisixDehyphens
  # Superclass for the two different methods using Affisix data.
  # Holds the common bits for parsing data and setting parameters.

  def initialize(filename)

    @threshold = 0.5
    @affixes = Hash.new(0.0)

    File.open(filename, "r:UTF-8") do |infile|
      infile.each_line do |line|
        line.match(/f: ([\d.]+), b: ([\d.]+), d: ([\d.]+) - (.*) \(/) { |m|
          # inside this block, m[x] holds the text of the x’th paren-match
          @affixes[m[4]] = m[3].to_f
          # stores the difference entropy under a key of the suffix
        }

      end
    end
  end

  def threshold=(thresh)
    # Sets the threshold entropy value.
    # Lower values are more permissive.
    @threshold = thresh
  end
end

class AffisixSuffix < AffisixDehyphens
  def decide?(before, after)
    if @affixes[after] > @threshold
      return false
    else
      return nil
    end
  end
end

class AffisixSandwich < AffisixDehyphens
  def decide?(before, after)
    (2..before.length).each do |len|
      cand = before[-len..-1]
      # looks at last len characters of the before-part
      if @affixes[cand] > @threshold
        return false
      end
    end
    return nil
  end
end
```

B.6 Detecting sandwiched words

This class depends on an external dictionary object, which must respond to the method `found?(string)` with a boolean. (`HyphenStat` will do for this purpose; see `EvalMethods.rb` for an example of usage.)

File `NoncePart.rb`:

```
# -*- coding: UTF-8 -*-

class NoncePartDehyphen
  def initialize(dictionary)
    @dict = dictionary
    # dictionary object must respond to .found? method
  end

  def decide?(before, after)
    before = before.sub(/-$/, '') # delete final hyphen
    if before.include? '-' or after.include? '-'
      beforeparts = before.split('-')
      afterparts = after.split('-')
      before = beforeparts.pop
      after = afterparts.shift
      beforeparts += afterparts
      founds = beforeparts.collect do |piece|
        @dict.found? piece
      end
      # founds is now an array of boolean values
      trues = founds.count(true)
      #if trues > (founds.length / 2) and trues > 0
        # the parts don't look like just noise
      if true
        if @dict.found? before+after
          return false  # the parts constitute a word; join the strings
        elsif @dict.found? before or @dict.found? after
          return true  # keep the hyphen
        else
          return nil  # undecided
        end
      end
    end
    return nil
  end

end
```

B.7 Stateless methods

File `StatelessDehyphen.rb`:

```
# -*- coding: UTF-8 -*-

class CapitalsDehyphen
  def initialize()
    # completely stateless, making instantiation pointless
  end

  def self.decide?(before, after)
    before = before.sub(/-$/, ’’) # delete final hyphen
    if (before =~ /[^[:lower:]]$/) or (after =~ /^[^[:lower:]]/)
      return true
    else
      return nil
    end
  end

  def decide?(before, after)
    CapitalsDehyphen.decide?(before, after)
  end
end

class DoubleConsonantDehyphen
  def self.decide?(before, after)
    if before =~ /[bcdfghjklmnpqrstvwxyz]$/
      if before[-1,1] == after[0,1]
```

```
        return false
      end
    end
  nil
  end

  def decide?(before,after)
    DoubleConsonantDehyphen.decide?(before, after)
  end

end

class WordLengthDehyphen
  def self.decide?(before, after)
    before = before.sub(/-$/, '')
    befores = before.split('-')
    afters = after.split('-')
    if befores[-1].length <= 1 or afters[0].length <= 1
      true
    else
      nil
    end
  end

  def decide?(before, after)
    WordLengthDehyphen.decide?(before,after)
  end
end
```

B.8 Rhyme detection

File `RhymingDehyphen.rb`:

```
# -*- coding: UTF-8 -*-

class RhymingDehyphen
  def initialize(suffixes=['s', 'es', 'al', 'ic', 'ical'])
    @suffixes = suffixes
  end

  # are the last two letters the same?
  def rhyming?(one, other)
    if one[-2..-1] == other[-2..-1]
```

```
      return true
    else
      return false
    end
  end

  def decide?(before, after)
    # pointless for strings shorter than 3 chars
    return nil if ([before.length,after.length].min < 3)
    if rhyming?(before,after)
      return true
    else @suffixes.each do |suffix|
      # look for rhymes, disregarding certain suffixes
      suffixrex = Regexp.new(suffix+'$')
      if rhyming?(before, after.sub(suffixrex, ''))
        return true
      end
    end
    end
    return nil
  end

end
```

B.9 Lexicographic algorithm

This code is split into two classes. `HyphenTally` is a support class which keeps a hash containing the word frequencies.

The actual dehyphenation is done by an instance of `HyphenStat`: call its `gobble` method with each filename you want to gather words from. The resulting frequency dictionary can be stored to disk with `savewords()`, and loaded quickly with `loadwords()`.

`savebroken()` is used to store the hyphenated words that were encountered, which is mostly useful to get data for evaluation.

`HyphenTally.keystrip()` normalizes a string into a hash key (by stripping out hyphens and oddball characters).

To simplify processing, the code assumes that newlines after hyphens which have been deleted are marked with an `<eol/>` XML entity; this can be changed via the `eol` parameter to the constructor (see line 85, below).

To make this class pass negative verdicts (as at the end of section 6.6), uncomment the three lines ending in '`ret = false`'.

File `HyphenStat.rb`:

```
# -*- coding: UTF-8 -*-
$lower = [ "ÆØÅÖÄ", "æøåöä" ]
def keystrip(str)
  return "" if (str.nil? or str.length < 1)
  striprex = Regexp.new(’[^’+@alphabet+’]’)
  key = str.gsub(striprex, ’’)
  key.downcase!
  key.tr( $lower[0], $lower[1] )
end

class HyphenTally

  def initialize(alphabet=’a-zA-ZæøåÆØÅäöÄÖ’)
    @alphabet = alphabet
    @tally = {}
  end
  def self.preproc(word)
    return word
  end

  def add(word)
    word = HyphenTally.preproc(word)
    key = keystrip(word)

    if not @tally.has_key? key
      @tally[key] = [ word, 1 ]
    else
      if idx = @tally[key].index(word)
        @tally[key][idx+1] += 1
      else @tally[key].push(word, 1)
        # ^ simply uses an array alternating between word and frequency;
        #    most keys will  have <4 entries, so springing for a
        #    hash of hashes would be overkill
      end
    end
  end

  def found?(word)
    key = keystrip(word)
    @tally.has_key? key  and @tally[key].index(word)
  end

  def decide?(before, after)
```

```
    decide_freq(before, after, 1)
  end

  # Only make a decision if a word occurs at least minfreq times
  #  call with minfreq=2 to disqualify hapaxes
  def decide_freq(before, after, minfreq=1)
    before = before.sub(/-$/,’’) # erase final hyphen if present
    key = keystrip(before+after)

    ret = nil  # default is to delete the hyphen
    if @tally.has_key? key
      scores = @tally[key]
      unhyphened = 0
      hyphened = 0
      # scores has format [word, freq, word, freq]
      0.step(scores.length-1, 2) do |i|
        if scores[i] == before+after
          unhyphened = scores[i+1]
        elsif scores[i] == before+’-’+after
          hyphened = scores[i+1]
        end
      end
      if hyphened > unhyphened and
        (hyphened + unhyphened) >= minfreq then
        ret = true
      # elsif hyphened < unhyphened and
      #  (hyphened + unhyphened) >= (minfreq+2) then
      #  ret = false
      end
    elsif before.include? ’-’ or after.include? ’-’
      # this might be a nonce-word
    end
    ret
  end

  def put(key, item)
    @tally[key] = item
  end

  # dump to tab-separated values
  def to_sepval(sep="\t")
    out = ’’
    entry = ’’
    @tally.each do |k,v|
      entry = ’’ + k
      v.each do |subitem|
        entry += sep + subitem.to_s
      end
      if block_given?
        yield entry
      else
        out += entry + "\n"
      end
    end
    out
  end

end

class HyphenStat
  def initialize(eol="<eol/>", alphabet=’a-zA-ZæøåÆØÅäöÄÖ’)
    @eol = eol  # end-of-line marker
    @alphabet = alphabet + "\u{0308}" + "\u{030a}"  # ah, Unicode
    @words = HyphenTally.new
    @hyphened = HyphenTally.new
  end

  def gobble(filename)
       food =  ’[-’ + @alphabet + ’]’
    nonfood = ’[^-’ + @alphabet + ’]’
    nonfoodre = Regexp.new(nonfood)
    eolre = Regexp.new(food +’+’+ @eol + food+’+’)

    IO.readlines(filename, encoding:’UTF-8’).each do |line|
      while eolre.match(line) { |m|
          parts = m.to_s.split(@eol)
          parts.each { |p| p.gsub!("\u{00ad}", ’’) }
            # delete soft hyphen if already present
          broken = parts[0] + "\u{00ad}" + parts[1]
            # then mark the breakpoint with a soft hyphen
          @hyphened.add(broken)
        }
        line.sub!(eolre, ’’)
          # keep hyphenated and unhyphenated populations separate
      end
      line.split(nonfoodre).each do |word|
        next if word.length < 3
        @words.add(word)
      end
    end
  end

  def decide?(before, after)
    @words.decide? before, after
  end

  def decide_freq(before,after,minfreq=1)
    @words.decide_freq(before,after,minfreq)
  end

  def found?(word)
    @words.found?(word)
  end

  def savewords(filehandle)
    savedata(filehandle, @words)
  end
  def savebroken(filehandle)
    savedata(filehandle, @hyphened)
  end
  def savedata(fh, thing)
    thing.to_sepval("\t") { |line| fh.puts(line) }
  end

  def loadwords(filename)
    File.open(filename, 'r:UTF-8') { |filehandle| loaddata(filehandle, @words) }
  end
  def loadbroken(filename)
    File.open(filename, 'r:UTF-8') { |filehandle| loaddata(filehandle, @hyphened) }
  end

  def loaddata(fh, thing)
    key = ''
    arr = nil
    fh.readlines.each do |line|
      items = line.chomp.split("\t")
      next if items.length < 3

      key = items[0]
      arr = []
      (1...items.length).each do |i|
        arr.push( (i%2!=0 ? items[i] : items[i].to_i) )
          # values alternate between strings and integers
      end
      thing.put(key, arr)
    end
  end

end
```

B.10 Method overlap

The following code counts how many items overlap between several functions on a given input sequence, that is, it calculates the degree of mutual judgements between them. Note that since it looks at all possible pairings, performance is necessarily O(n^2). This makes it unwieldy for evaluating more than twenty or so methods at a time.

The code in the classes themselves is not specific to dehyphenation; that gets taken care of in the script section beginning on line 90. The script given here is the one that was used to compute Table 5.

The utility function `permute()` takes an array of elements and returns all possible combinations of them except pairing an element with itself (similar to a Cartesian product). Those combinations are used in `DecidersEval.results()` to examine all possible pairings of the `Proc` objects which encapsulate different methods for dehyphenation. The Proc(ess) objects are callbacks to dehyphenation methods, similar to anonymous functions in Java; their use is demonstrated on line 100 of the script section.

File `DecidersEval.rb`:

```
# -*- coding: UTF-8 -*-

def permute(items)
  ret = []
  items.each do |item1|
    items.each do |item2|
      next if item1==item2
      pair = [item1, item2]
      if not ret.include? pair
        ret.push(pair)
        # ordering is treated as significant
      end
    end
  end
  ret
end

class DecidersEval
  def initialize()
    @deciders = Hash.new
  end

  def addMethod(label, &meth)
    meth.kind_of? Proc or raise ArgumentError
    @deciders[label] = meth
  end

  def results(input)
    parts = []
    result = nil
    results = {}
    @deciders.each do |label, decider|
      results[label] = []
      input.each do |word|
        parts = word.split("\u{00ad}") # Unicode 'soft hyphen'
        result = decider.call(*parts)
        results[label].push(result)
      end
    end
    pairings = permute(@deciders.keys)
    overlap = {}
    pairings.each do |pairing|
      one = results[pairing[0]]
      other = results[pairing[1]]
      overlap[pairing] = []  # note: indexed by arrays of two strings
      one.each_index do |idx|
        overlap[pairing][idx] = nil
        if one[idx]!=nil and other[idx]!=nil
          overlap[pairing][idx] = (one[idx]==other[idx])
            # true or false
        end
      end
    end
    [results, overlap]
  end
  def friendlyOverlap(input)
    ret = results(input)
    results = ret[0]
    overlap = ret[1]
    res = {}
    results.each_key do |methname|
      answers = results[methname]
      nonnil = answers.find_all { |a| not a.nil? }
      print methname, "\t", nonnil.length, "\n"
    end

    overlap.each_key do |pairing|
      res[pairing] = []
      bools = overlap[pairing]
      bools.each_index do |idx|
        if not bools[idx].nil?
          code = (bools[idx] ? ’*’ : ’,’)
          res[pairing].push(input[idx].sub("\u{00ad}",’.’)+code)
        end
      end
      coverages = []
      pairing.each do |method|
        covered = results[method].find_all {|elem| not elem.nil? }
        coverages.push(covered.length)
      end
      #~ greatest = coverages.max
      greatest = coverages[0]
      overlapping = overlap[pairing].find_all {|elem| not elem.nil? }
      overlapped = overlapping.length
      print pairing.join("-"), " ",
    (overlapped.to_f/greatest)*100, "% ", overlapped, "/", greatest, "\n"
      # " overlap: ",
    end

    res
  end
end

# --- script section begins here ---

require ’./StatelessDehyphen.rb’
# require ’./HyphenedAffixGuesser.rb’
require ’./MorphoData.rb’
require ’./MorfessorDehyphen.rb’
require ’./AffisixDehyphen.rb’
require ’./HyphenStat.rb’
require ’./NoncePart.rb’
require ’./RhymingDehyphen.rb’

#~ affixguesser = HyphenedAffixGuesser.new(5)
#~ affixguesser.load_tsv(’words-all.txt’)
linguistica = MorphoData.new(2)
linguistica.readSuffixes(’nosuffixes2.txt’)
morfessor = MorfessorDehyphen.new(’no01.morfessor’)
affisuffix = AffisixSuffix.new(’affis-suffix-no.txt’)
affisuffix.threshold = 1.05
affisandwich = AffisixSandwich.new(’affis-sandwich-no.txt’)
affisandwich.threshold = 1.05
rhyming = RhymingDehyphen.new([’e’, ’er’, ’en’, ’sk’, ’ske’, ’ig’, ’ær’])
lexicog = HyphenStat.new
lexicog.loadwords(’words-all.txt’)
noncepart = NoncePartDehyphen.new(lexicog)
# ^ taps into the dictionary loaded with HyphenStat

measurer = DecidersEval.new
measurer.addMethod("wordlength") { |b, a| WordLengthDehyphen.decide?(b,a) }
# measurer.addMethod("affixguess") { |b, a| affixguesser.decide?(b,a) }
measurer.addMethod("Lingustica") { |b, a| linguistica.decide?(b,a) }
measurer.addMethod("Morfessor") { |b, a| morfessor.decide?(b,a) }
measurer.addMethod("affisuffix") { |b, a| affisuffix.decide?(b,a) }
measurer.addMethod("affisandwich") { |b, a| affisandwich.decide?(b,a) }
measurer.addMethod("doublecons") { |b, a| DoubleConsonantDehyphen.decide?(b,a) }
measurer.addMethod("capitals") { |b, a| CapitalsDehyphen.decide?(b,a) }
measurer.addMethod("rhyming") { |b, a| rhyming.decide?(b,a) }
measurer.addMethod("lexicog") { |b, a| lexicog.decide?(b,a) }
measurer.addMethod("noncepart") { |b, a| noncepart.decide?(b,a) }

input = []
File.open("dehyphend1.txt", "r:UTF-8") do |fh|
  fh.each_line do |line|
    line.chomp!
    next if line.length<=1
    input.push(line.gsub("-\u{00ad}", "\u{00ad}"))
  end
end
```

```
results = measurer.friendlyOverlap(input)
results.each do |key, items|
  print key.join("-"), "\n"
  print items.join(" "), "\n"
end
```

B.11 Generating a gold standard

This is the script used in section 6.4 to generate a gold standard for evaluation. It takes a hyphenated text, compares it to the unhyphenated original, and outputs the gold standard.

(For ease of debugging, the output is a bit noisy; the reader may prefer to pipe standard output to `/dev/null`.)

File `MakeFacit.rb`:

```
# -*- coding: UTF-8 -*-

def usage
  print <<_HEREDOC
Usage: ruby MakeFacit.rb original hyphened standard
Compares original to hyphened, writing the correct solution for each
hyphenated word to standard.  For use with DecidersEval.rb.
_HEREDOC
end
if ARGV.length!=3
  usage(); exit
end

$alphabet="’"+’a-zA-ZæøåÆØÅäöÄÖ’
$nonwords=Regexp.new(’[^-’+$alphabet+’]+’)
$hypwords=Regexp.new(’[^-’+$alphabet+’]+’)

File.open(ARGV[0], "r:UTF-8") do |origfh|
  File.open(ARGV[1], "r:UTF-8") do |afterfh|
    File.open(ARGV[2], "w:UTF-8") do |resultfile|
      afters = []
      prev = nil
      afterfh.each_line do |line|
        line = line.chomp.gsub(/\s+$/, ’’).gsub(/^\s+/, ’’)
        words = line.split($nonwords)
        if prev
          afters.push(prev + "\u{00ad}" + words[0])
          prev = nil
  end
  if line =~ /-$/
    prev = words[-1].sub(/-$/, '')
  end
end
answers = []
originals = origfh.read().split($hypwords)
#~ print originals.join(" ")

origpos = -1
afters.each do |after|
  next if after =~ /-$/
  bits = after.split("\u{00ad}")
  next if not (bits[0] and bits[1])
  next if bits[0].length < 2 or bits[1].length < 2
  cand0 = bits[0] + bits[1]
  cand1 = bits[0] + "-" + bits[1]
  print "\n+", cand1, " "; $stdout.flush;
  cont = true
  while cont
    origpos += 1
    break if origpos > originals.length
    thisExam = originals[origpos]
    print thisExam, " "; $stdout.flush;
    cont = false
    if thisExam =~ /-$/ and thisExam =~ /[^-]/
      if cand0.include? thisExam.sub(/-$/, '')
        cont = false
      else cont = true
      end
        # "one- or two-person" causes enormous problems
    elsif thisExam == cand0
      answers.push(after)
    elsif thisExam == cand1
      answers.push(bits[0] + "-\u{00ad}" + bits[1])
    else
      cont = true
    end
  end
  #~ print answers[-1], " "; $stdout.flush;
end
answers.each do |answer|
  resultfile.print answer, "\n"
```

```
      end
    end
  end
end
```

B.12 Sampling hyphenations

This script reads from all XML files in the current directory, sampling hyphenations that have been marked with an `<eol/>` entity and writing them to a file `sampleeols.txt` (which will be overwritten if it already exists).

The positions to be sampled are chosen by a random walk that only takes positive step values. The `period` argument to `spit_lines()` determines the step size (minus one; in the script below, passing it a step size of 9 means that the average skip-ahead will tend towards 5).

File `HyphenSample.rb`:

```
# -*- coding: UTF-8 -*-

def spit_lines(fns, period=25)

  eol="<eol/>"
  alphabet=’a-zA-ZæøåÆØÅäöÄÖ’ + "\u{0308}" + "\u{030a}"
  food = ’[-’ + alphabet + ’]’
  eolre = Regexp.new(food +’+’+ eol + food+’+’)

  counter = period - rand(period)
  fns.each do |filename|
    IO.readlines(filename, encoding:’UTF-8’).each do |line|
      if line.include? eol
        counter -= 1
        if counter<=0
          counter = period - rand(period)
          if block_given?
            yield line
          else print line
          end
        end
      end
    end
  end
end
```

```
def shuffle(arr)
  elem = nil
  repl = -1
  (0...arr.length).each do |i|
    repl = rand(arr.length)
    elem = arr[i]
    arr[i] = arr[repl]
    arr[repl] = elem
  end
  arr
end

fns = shuffle(Dir['*.xml'])
# spit_lines(fns, 29)
File.open("sampleeols.txt", "w") do |fh|
  spit_lines(fns, 9) { |line| fh.print(line) }
end
```

APPENDIX C

Bibliography

Peter Ackema & Ad Neeleman 2004, *Beyond Morphology: Interface Conditions on Word Formations.* Oxford Studies in Theoretical Linguistics 6. ISBN 0-19-926729-4

David Beymer, Daniel M. Russell & Peter Z. Orton 2005, Wide vs. Narrow Paragraphs: An Eye Tracking Analysis. *INTERACT 2005*, LNCS 3585, pp. 741–752. doi:10.1007/11555261_59

Henry Bradley 1913, *On the relations between Spoken and Written Language, with special reference to English.* From the Proceedings of the British Academy, Vol. VI. http://www.archive.org/details/onrelationsbetwe00bradrich

Lars Bungum 2008, *Automatisk oversettelse av norske substantiv-komposita: En eksperimentell studie.* MSc thesis, University of Oslo. http://urn.nb.no/URN:NBN:no-21030

Mathias Creutz & Krista Lagus 2005, *Unsupervised Morpheme Segmentation and Morphology Induction from Text Corpora Using Morfessor 1.0.* Publications in Computer and Information Science, Report A81, Helsinki University of Technology, March. http://www.cis.hut.fi/projects/morpho/

Dmitriy Genzel & Eugene Charniak 2002, Entropy rate constancy in text. In P. Isabelle (Ed.), *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 199–206.

John Goldsmith 2001, Unsupervised Learning of the Morphology of a Natural Language. *Computational Linguistics* 27(2), 153–198. doi:10.1162/089120101750300490

Bob Gordon 2001, *Making Digital Type Look Good.* Ilex Press. ISBN 0-500-283133

Gregory Grefenstette & Pasi Tapanainen 1994, What is a word, What is a sentence? Problems of Tokenization. In *Proceedings of the 3rd International Conference on Computational Lexicography*, 79–87.

Roy Harris 1981, *The Language Myth.* Duckworth. ISBN 0-7156-1528-9

Roy Harris 1995, *Signs of Writing.* Routledge. ISBN 0-415-10088-7

Jaroslava Hlaváčová & Michal Hrušecký 2008, *Affisix: Tool for Prefix Recognition.* In P. Sojka et al. (eds.): TSD 2008, LNAI 5246, 85–92

Jukka Hyönä & Alexander Pollatsek 1998, Reading Finnish Compound Words: Eye Fixations Are Affected by Component Morphemes. *Journal of Experimental Psychology: Human Perception and Performance* 24(6), 1612–27

Bernard Jones 1996, *What's The Point? A (Computational) Theory of Punctuation.* PhD thesis, University of Edinburgh. http://hdl.handle.net/1842/519

Major Keary 1991, *On Hyphenation - Anarchy of Pedantry.* PC Update Online. http://www.melbpc.org.au/pcupdate/9100/9112article4.htm

Franklin Mark Liang 1983, *Word Hy-phen-a-tion by Com-put-er.* PhD thesis, Stanford University. Report No. STAN-CS-83-977

Jonathan Ling & Paul van Schaik 2006, The influence of font type and line length on visual search and information retrieval in web pages. *Int. J. Human-Computer Studies 64*, 395–404. doi:10.1016/j.ijhcs.2005.08.015

Simon P. Liversedge & Hazel I. Blythe 2007, Lexical and Sublexical Influences on Eye Movements During Reading. *Language and Linguistics Compass* 1/1–2, 17–31. doi:10.1111/j.1749-818x.2007.00003.x

Christopher D. Manning & Hinrich Schütze 2000, *Foundations of statistical natural language processing.* MIT Press. ISBN 0-262-13360-1

Ruari McLean 2000, *How Typography Happens.* The British Library & Oak Knoll Press. ISBN 1-58456-019-3

James B. McMillan 1980, Infixing and Interposing in English. *American Speech* 55(3), pp. 163-83. http://www.jstor.org/stable/455082

Jörg Meibauer 2007, How marginal are phrasal compounds? Generalized insertion, expressivity, and I/Q-interaction. *Morphology* 17(2), 233-59

Charles F. Meyer 1987, *A Linguistic Study of American Punctuation.* American university studies. Series XIII, Linguistics ; vol. 5. Peter Lang Publishing. ISBN 0-8204-0522-1

Geoffrey Nunberg 1990, *The Linguistics of Punctuation.* CSLI lecture notes: no. 18, University of Chicago Press. ISBN 0-937073-46-6

Geoffrey Nunberg, Edward Briscoe & Rodney Huddleston 2002, Punctuation and Text-Category Indicators. In R. Huddleston and G. K. Pullum (eds.), *The Cambridge Grammar of English.*

Wolfgang A. Ocker, A Program to Hyphenate English Words. *IEEE Transactions on Engineering Writing and Speech*, Vol. EWS-14, No. 2, June 1971.

M. B. Parkes 1992, *Pause and effect : an introduction to the history of punctuation in the West.* ISBN 0-85967-742-7, 0-520-07941-8

Eric Partridge 1953, *You Have a Point There: A Guide to Punctuation and Its Allies.* Hamish Hamilton : London.

Graeme Ritchie 2004, *The Linguistic Analysis of Jokes.* Routledge Studies in Linguistics no. 2. ISBN 0-415-30983-2

Bilge Say & Varol Akman 1997, Current Approaches to Punctuation in Computational Linguistics. *Computers and the Humanities* 30(6), 457–469.

Bilge Say 1998, *An Information-based Approach to Punctuation.* PhD Thesis, Dept. of Engineering and Information Sciences, Bilkent University.

Yvonne Schwemer-Scheddin 1998, Broken Images: Blackletter between Faith and Mysticism. In Peter Bain & Paul Shaw (eds.), *Blackletter: Type and National Identity.* ISBN 1-56898-125-2

A. Dawn Shaikh 2005, The Effects of Line Length on Reading Online News. *Usability News* 7(2), July 2005. http://psychology.wichita.edu/surl/usabilitynews/72/LineLength.asp

Reginald Skelton 1949, *Modern English Punctuation.* Second edition. Sir Isaac Pitman & Sons : London.

Hàn Thế Thành 2004, Micro-typographic Extensions of pdfTEX in Practice. *TUGboat* 25(1): Proceedings of the Practical TEX 2004 Conference.

Miles A. Tinker & Donald G. Paterson 1929, Studies of Typographical Factors Influencing Speed of Reading. III. Length of Line. *The Journal of Applied Psychology* 13(3), 205-219.

Edward R. Tufte 1983, *The visual display of quantitative information.* Graphics Press. ISBN 0-9613921-0-x

Robert H. W. Waller 1980, Graphic aspects of complex texts: Typography as macro-punctuation. In P. A. Kolers, M. E. Wrolstad, & H. Bourma (eds.), *Processing of Visible Language*, Vol. II. Plenum Press.

Robert Waller 1987, *The typographic contribution to language: Towards a model of typographic genres and their underlying structures.* PhD thesis, University of Reading.

C. L. Wrenn 1967, *Word and Symbol: Studies in English Language.* English Language Series. Longmans : London.