Approximate Matching

COMP90049 Knowledge Technologies

1 Closest match

Closest match

A common computational task is to find the nearest match to a given string:

- Spelling correction, name matching, query repair, phonetic matching, data cleansing, genomics.

Further challenge: find strings with the nearest match to any of their substrings. (Why is this harder?)

Methods that address this task are "knowledge technologies" as defined in this subject.

- Thorough or exhaustive solutions (in some contexts) require unrealistic computational resources, so we need to find workable approximations.
- Correctness isn't well-defined.

Another perspective: this is *search with uncertainty*.

Search with uncertainty

Consider a file of names such as

K. Many Chandy, Jayadev Misra
Susanne Graf, Joseph Sifakis
D. Harel, A. Pnueli
Koichi Furukawa, Katsumi Niita, Yuji Matsumoto
Z. Farkas, P. Szeredi, E. Santane-Toth
Agneta Eriksson, Anna-Lena Johansson, Sten-Ake Tarnlund
Luis Moniz Pereira, Antonio Porto
Lynette Hirchmann, Karl Puder
Yuji Matsumoto, Hozumi Tanaka, Masaki Kiyono
J. Darlington, A.J. Field, H. Pull
Masahiko Sato, Tatakfumi Sakurai
P.A. Subrahmanyam, Jia-Huai You
Joseph A. Goguen, Jose Meseguer
Yonathan Malachi, Zohar Manna, Richard Waldinger

How to find lines in this file, if the precise spelling is unknown?

Search with uncertainty

Context matters: not just language and alphabet, but the reason that the search is taking place.

In designing or choosing a search method, three factors need to be considered:

- What the user is trying to achieve.
- The source of reference material for "correct" strings.
- Tractability and interactivity – is a user waiting for a response? How will the solution behave as the uncertainty increases?

Tools: ispell, agrep, web search interfaces.

Example: the `agrep` tool is an implementation of an algorithm for matching with errors.

Dictionaries

To be able to correct errors in strings, there needs to be a known 'truth': a *reference* against which comparisons can be made.

For spelling correction, this is (obviously?) some form of dictionary. But what is in the dictionary?

Commonest words in a sample of English web data:

```
the of and to a in is for on that with be or
by are you as this it at from an will not
have your can was all which has modified one
more if but their we other about they any use
no time out may up also do our only new he
his would been who its than when some so name
should like were first there line used into
two these follows year my what them such over
had must each me then list here get per
through make see how most work
```

Dictionaries

Some common (but not the commonest) names in a register of companies:

```
lucy whitehead miranda sheffield fletcher
mccoy donner ernest weston hooks sy kaiser
grams fink quinn petr ngo redmond walton
hernandez lam kin clever hy thrust ferrari
garrett carey lester limb livermore doyle
gallagher convey barber porte benson zhang
brock gomez jazayeri hare mare blessing valle
dover anton lage zurich
```

Some names have a great deal of variation in spelling: `karen karin kerryn caryn`... are "the same"

... though probably not `kerry derryn`... despite the overall similarity.

Spelling correction in context

In principle, it might seem appealing to use grammar or morphological structure to help guide general spelling correction.

But there are significant obstacles:

- Relatively few spelling errors can be resolved by context.
- The cost of building the dictionary, and its size, would be prohibitive.

There are good contextual spelling checkers for typical errors, based on both machine learning and manual analysis of syntax and error patterns. These are most effective for short words.

There are no spelling checkers that use broad context, such as the category or topic of a document (however this might be defined). We are probably stuck with order-0 (context-free) correction for most words. But we can try and be smarter about what we try to correct . . .

Aside: Stemming and spelling correction

Two kinds of dictionaries are widely used in English spelling correction.

- *Literal:* a set of strings that must be matched exactly.
- *Root+stem:* a set of root words, a set of suffixes, and a set of rules for applying suffixes to roots.

Root: `fill`, `apply`, `invoke`.

Word:
`filled`, `filling`, but not `fillication`
`applied`, `application`, but not `apple`
`invoked`, `invoking`, `invocation`, but not `invokery`

Stemming is the process of stripping suffixes automatically. It allows a dictionary to be generalized, but many root-suffix combinations are not correct spellings.

Observe that such technologies are highly language-dependent.

Dictionaries

Web search tools offer corrections to query spellings, typically one or zero alternatives per query.

Commonest words in a sample of 2005 queries for which the top match was any `.gov` website:

```
of in and the department for state hurricane
michigan county national california to
federal us weather map lottery fema court a
on states form united washington center
health indiana katrina service new office act
how ohio wisconsin code services board
government 2005 food security time what city
forms oregon tax social education missouri
park utah is statistics world drug program
names codes labor dept school bureau district
information virginia medical irs public with
child kentucky nyc jobs radar nasa noaa
cancer water unemployment medicare cia va
report
```

Conclusion: use of web data to correct errors in queries is very unlikely to work. Search engines use query logs (see “britney spears query correction”).

2 Matching with errors

Pattern-matching tools and error correction

The motivating use of pattern matching tools such as `agrep` is to identify likely matching lines in a file. No assumptions are made about meaning of characters.

The aim of spelling correction is to:

- Parse text into “words”.
- Ignore the words that are found in a dictionary.
- Find the best matches for the remainder.

```
...  proceed if you can see no ther option
...
```

Matches with `agrep -1`:
`ether`, `her`, `other`, `thar`, `the`, `thee`, `their`, `them`, `then`, `there`, `therm`, `thew`, `they`, `tier`

Pattern-matching tools and error correction

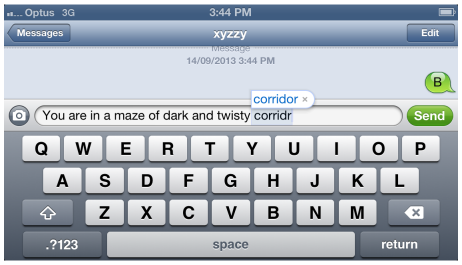
A straightforward process, but it does expose underlying concepts:

- *Efficiency* and *ranking*.

Ranking. “Best match” is relative. Whether a string is a match to a query is not a binary (yes/no) decision, and the goal should be to present matches according to perceived likelihood of their being correct.

Observe that the task inherently requires human participation.

Pattern-matching tools and error correction



You are in a maze of dark and twisty corridr

Pattern-matching tools and error correction

Efficiency. Exhaustive file search is $O(n)$ in time, for say a dictionary of n entries.

Most words should be found in the dictionary; there are plenty of data structures that provide effective $O(1)$ confirmation of whether a particular string is an exact match (e.g., hashing or tries).

Ballpark: 10^6 to 10^7 searches per second on a laptop.

$O(n)$ may be unsatisfactory for the approximate search task. Perhaps more importantly, processing a file includes overheads such as parsing text into lines – doing so for each word to be checked is a waste of resources.

Pattern matching tools aren't customized for parsing; they are not efficient at eliminating exact matches; and don't rank results. Conclusion: Despite the task similarity, they are not suitable for spell checking.

Neighbourhood search

Given a query string that is not in the dictionary, the task is to find the nearest neighbours.

One approach is to generate all neighbours, and see if they are present.

- Enumerate all of the single-character deletions, insertions, and replacements.

This generates all strings whose distance from the query is 1.

For example, for an alphabet of four letters {`e h r t`}, the word `thr` has the immediate neighbours {`hr tr th ethr hthr rthr tthr tehr thhr trhr tthr ther thhr thrr thtr thre thrh thrr thrt ehr hhr rhr ter trr ttr the thh tht`}

How many neighbours? Alphabet size A, query length n:

$$N_1 = (n-1) + A \cdot (n+1) + (A-1) \cdot n$$

Ballpark: $A = 26$, $n = 8$, so $N_1 = 441$ and $\gg 1000$ unknown strings can be processed per second.

Looks feasible! This is how `ispell` works.

Costs

The cost of neighbourhood search is $O(n)$ since the alphabet is fixed.

What happens if we want to increase the distance from the query?

$$N_2 = (n-1)(n-2) + A^2 \cdot (n+1)(n+2) + (A-1)^2 \cdot n(n-1) + \cdots$$

This does not include combinations of deletion, replacement, and insertion.

Ballpark: $N_2 \gg 50,000$, and cost is $O(n^2)$ with a high (≈ 600) constant factor.

This combinatoric analysis indicates that exhaustive search beyond the immediate (distance$=1$) neighbourhood is at the upper limit of feasibility – and usability.

Neighbourhood size

Query is `ther`. What is going on here?

```
aer aether ahem anther aver bather beer bier bother
char cheer chef chert chew chez cither daer deer
dither doer dyer either etcher ether ever ewer
father fer gather ghee goer he hear heir hem hen
hep her herb herd here herl herm hern hero hers hes
hew hex hey hither hoer jeer kier lather leer
lither mither mother nether oer omer other others
otter over oyer peer per phew pier pother rather
rhea seer she shear shed sheer sherd shes shew
shier shoer shyer steer suer taed tael tahr taker
taler tamer taper tar tater taxer tea tear tee teed
teem teen tees ten term tern tether thaler than
thar that thaw the thebe theca thee theed thees
theft thegn their theirs them theme then thenar
theory there therm therms these theta thew thews
thewy they thief thin third this tho thorn thorp
thou three threw thru thud thug thus thy tie tied
tier tiers ties tiger tiler timer tither toe toea
toed toes toner toper tor torr toter tother tour
tower tree tref trek tret trey trier truer tsar
tuber tuner twee twerp tzar user usher utter veer
weer wether whee when where whet whew whey whir
wither yer zither
```

There is a trade-off between *false negatives* and *false positives*. That is, there is a need to consider *effectiveness*.

Trie search with errors

Consider search in a trie for a query $q = c_1 c_2 c_3 \ldots$ allowing k mismatches.

In the current node, there are $|\Omega|$ subtries.

- In the subtrie T_c rooted at the node pointed to by A_c, where $c = c_1$ is the first character in q, we need to search with $q' = c_2c_3 \ldots$ allowing k errors.
- In the other subtries $T' \in \{T_{c'} \mid c' \in \Omega, c' \neq c\}$, we need to search with $q' = c_2c_3 \ldots$ allowing $k - 1$ errors; if $k = 0$ the search stops (failure).

There is also a simple extension that allows the number of mismatches to grow: by placing all 'failure' nodes in a queue, they can be revisited with increased k if no matches are found with the initial k.

(With care – the number of failure nodes grows exponentially with k.)

Tries can similarly be extended to deal with insertions and deletions.

Combinatoric search

Another practical technique is to use variants of binary search, where:

- Strings are sorted, that is, grouped by prefix.
- Strings are inspected character-by-character until it is clear that their distance from the query is greater than k.
- Due to the sorting, groups of strings that share a prefix can be considered simultaneously.

Since the string space is extremely sparse (of the possible A^n strings, only a tiny fraction are actually observed), and there are good special-purpose data structures (e.g., suffix arrays), the cost in practice is around that of neighbourhood search at distance 1.

The distance limit k must be chosen, or repeated search is required. Additional work is required for ranking.

Because there is no sorting of a dictionary that ensures that similar strings are close together, it is not obvious what the best solution to this search problem is.

Edit distances

To compare strings: take the query string and each candidate match, and find the *edit distance* between them.

The simplest edit distance is the number of insertions, deletions, and substitutions needed to turn one string into another.

The smaller the edit distance, the closer the match.

```
Gorbach–ev
     |  |
Gorbechyov
```

```
Connell
      |
Con–al–
```

Measures such as edit distances are attractive because matches are scored, so answers can be ranked.

Edit distances

Assume an array F of size $|q| \times |t|$, to be used to compute the *global* edit distance between q and t.

```
lq = strlen(q); lt = strlen(t);
for( i=0 ; i<=lq ; i++ ) F[0][i] = i;
for( j=0 ; j<=lt ; j++ ) F[j][0] = j;

for( i=1 ; i<=lq ; i++ )
    for( j=1 ; j<=lt ; j++ )
        F[i][j] = min3(
            F[i-1][j] + 1,
            F[i][j-1] + 1,
            F[i-1][j-1] + equal(q[i-1], t[j-1]));
```

Here, `equal()` returns 0 if equal, 1 otherwise.

The value `F[lq][lt]` is the minimum number of edits (replacements and indels) between between q and t.

This method is known for historical reasons as dynamic programming.

Edit distances

Computing the minimal global alignment between `madoda` and `dada`:

		m	a	d	o	d	a
	0	1	2	3	4	5	6
d	1	1	2	2	3	4	5
a	2	2	1	2	3	4	4
d	3	3	2	1	2	3	4
a	4	4	3	2	2	3	**3**

Thus we find that 3 edits is the minimum required.

The edits required can be seen by tracing back through the matrix: delete `m` and the first `a`, and replace `o` with `d`, for example.

```
madoda
     |
––dada
```

Edit distances

The *local* edit distance is similar.

```
lq = strlen(q); lt = strlen(t);
for( i=0 ; i<=lq ; i++ ) F[0][i] = 0;
for( j=0 ; j<=lt ; j++ ) F[j][0] = 0;

for( i=1 ; i<=lq ; i++ )
    for( j=1 ; j<=lt ; j++ )
        F[i][j] = max4(
            0,
            F[i-1][j] - 1,
            F[i][j-1] - 1,
            F[i-1][j-1] + equal(q[i-1], t[j-1]));
```

Here, `equal()` returns $+1$ if equal, -1 otherwise. (But there are many choices of scoring schemes, as we discuss later.)

The value of `F[i][j]` with the *highest* value represents the best alignment.

Edit distances

Computing the minimal local alignment between `pononk` and `sonny`:

		p	o	n	o	n	k
	0	0	0	0	0	0	0
s	0	0	0	0	0	0	0
o	0	0	1	0	1	0	0
n	0	0	0	2	1	2	1
n	0	0	0	1	1	2	1
y	0	0	0	0	0	1	1

The edits required can be seen by tracing back through the matrix. The longest regions of the highest similarity are `onon` and `onn`.

```
pononk
|   |
son–ny
```

N-grams

Let $G_n(s)$ be the substrings of length n of s.

Example: G_2(`Gorbachev`) is `Go`, `or`, `rb`, `ba`, `ac`, `ch`, `he`, `ev`

The n-gram distance of t and s is

$$|G_n(s)| + |G_n(t)| - 2 \times |G_n(s) \cap G_n(t)|$$

Example: $s =$ `Gorbachev`, $t =$ `Gorbechyov`, distance is $8 + 9 - 2 \times 4 = 9$.

The smaller the n-gram distance, the closer the match. Cost is approximately $O(|s| + |t|)$, compared to $O(|s| \cdot |t|)$ for an edit distance.

A useful variant, but not much cheaper than an edit distance in practice – it still requires an exhaustive search of the candidates.

Seems unlikely to produce useful scores for long strings or small alphabets.

N-grams

A feature of n-grams is that they parse the string into tokens.

In principle, this means that, for each n-gram, we can have a list of the strings containing that n-gram.

Finding the close matches to a query means that we need only check the strings with an n-gram in common with the query.

This is a real saving (though more distant matches are missed). Consider 3-grams:

- About 10,000 different n-grams in dictionary entries; less than 10 n-grams per entry; so only 1/1000 of entries per n-gram.
- Less than 10 n-grams in a query; so inspect less than 1/100 of entries to resolve a query.

This is still $O(N)$ for a dictionary of N entries, and requires a reasonably large data structure.

Aside: Query correction

As to what is actually being done by the major search engines . . . they haven't told us (as far as I know).

Some guesses:

- Around 10^9 queries per day, 10^8 distinct queries. All of these are used as reference data.
- 'Failed' (reformulated) queries are noted and added to a dictionary using an offline process that considers successful queries.
- The search space is reduced by assuming n-grams in common between query and target, probably with quite high n.
- Trie methods for query completion (most common query with a matching prefix).
- Lots of computational resources – running a broken query is likely to be much more expensive than fixing the query first.

3 Phonetic matching

Phonetic matching

Many databases include vocabularies of words or names:

- Surnames in the telephone directory.
- Distinct words occurring in a text database.
- Place names in a gazette.

Phonetic matching techniques allow searching for a name or word when the exact spelling is unknown:

- A user may only know a name by its sound.
- A name can be misspelt in a database.
- Some names have variant or indeterminate spelling.

Example

Spelling of names is highly variable:

> Lho, Lo, Loan, Loe, Loew, Lough, Low, Lowe, . . .

In ≈1 Gb of newspaper articles the name "Gorbachev" had $>$ 20 variants including:

> Gorbachev, Gorbacahev, Gorbahev, Gorbatchev, Gorbechev, Gorbachov, Gorachev, Gorbacheva, Gorbechyev, Gorbacev, Gorbachyov, Gorabchev, Grobachev, . . .

Why?

Simple case: Schröder → Schroder or Schroeder?

Soundex

The Soundex sound-alike matching technique is well-known, widely used, simple to implement.

Transforms a string into a 4-character code that represents its sound:

- Replace all but the first letter by one of seven single-digit codes:

a e h i o u w y → 0	b f p v → 1
c g j k q s x z → 2	d t → 3
l → 4 m n → 5	r → 6

- Remove doubles, then remove 0s.
- Truncate to four symbols.

There are 8,918 distinct Soundex codes.

Soundex examples

```
     king   kyngge
⇒    k052   k05220
⇒    k052   k0520
⇒    k52    k52

     knight   night
⇒    k50203   n0203
⇒    k50203   n0203
⇒    k523     n23

     Loan   Loew   Lough   Lewicks
⇒    l005   l000   l002    l000222
⇒    l05    l0     l02     l02
⇒    l5     l      l2      l2
```

There are also (failed) variants, such as Phonix.

Lexicographic methods

Strings that sound alike are often spelt alike.

Example: Gorbachev, Gorbechyov, . . .

It is plausible to suppose that lexicographic string comparison methods will be effective at finding strings that sound alike.

Candidates include neighbourhood search, edit-distances, and n-gram measures.

Perhaps these would be enhanced by combining them with approaches that embody language knowledge.

Refining edit distances

Based on lexicographic information alone, matches such as "Crews" and "Kroose" would not be highly ranked.

Can edit distances be modified to allow for phonetic similarity somehow?

- Repeat letters don't usually change the sound; "Conel" sounds like "Connell".
- Sometimes different letters give rise to the same sound; "Kodi" sounds like "Cody".
- But sometimes they don't; "like" doesn't sound like "lice".

"Editex"

Exploring some possibilities . . . can edit distances be phonetically tweaked?

My `editex`: Like the standard edit distance, but instead of having a fixed penalty for insertion, deletion, and substitution, have two penalties:

- High for letters that are never similar, such as “d” and “m”.
- Low for letters that can give rise to similar sound, such as “m”–“n” and “c”–“k”.

Ten	1. a e i o u y	2. b p	3. c k q	4. d t	5. l r
groupings:	6. f p v	7. s x z	8. c s z	9. m n	10. g j

(The silent letters “w” and “h” are handled as a special case.)

```
Crews–
≀ ≀ |
Kroose
```

But what about phonetics?

Letters don’t represent sounds: consider “t” in “trim” versus “think”.

“th” represents a sound: the phoneme “Θ” in the International Phonetic Alphabet.

Every name has a pronunciation, or, more precisely, can be represented as a string of phonemes. Perhaps these can be used for matching in some way.

Two problems:

- Deciding what pronunciation corresponds to a given string.
- Comparing pronunciations.

“Ipadist”

- Use a text-to-sound algorithm to generate the likely pronunciation of each string.
- Use some form of edit distance to compare pronunciations.

Crews	→	krjϱs
Kroose	→	kr–ϱs

A more principled solution but:

- Text-to-sound algorithms are highly reliable only if context is available, for example to distinguish “in a minute” from “it is minute”.
- The pronunciation of names is much less predictable than the pronunciation of words.

Partial table of consonants

Phonetic symbols can also be compared for similarity:

T1	T2								
	b	*l*	*d*	*a*	*p*	*r*	*P*	*v*	*g*
p	p b			t d				k g	
n	m			n		N		ŋ	
t				r					
f	Φ	f v	Θ *d*	s z	*s x*				h
F				ɬ ɮ					
a							j		
A				l					

p	plosive	*f*	fricative	*F*	lateral fricative
n	nasal	*a*	approximant	*A*	lateral approximant
t	trill				

b	bilabial	*a*	alveolar	*P*	palatal
l	labiodermal	*p*	postalveolar	*v*	velar
d	dental	*r*	retroflex	*g*	glottal

4 Measurement of effectiveness

Performance

Measurement of efficiency is straightforward.

But we also need to know whether the method is useful. Measurement of *effectiveness* involves human assessment of the quality of the results.

For each task we need to find an independent method that measures effectiveness in a reliable way.

- Choose a test lexicon and a set of query strings.
- For each query and each word in the lexicon, use relevance assessors to decide whether the word and query match, that is, the word is *relevant* to the query.
- Apply the matching technique to the lexicon and each query to get a list of possible answers, and measure effectiveness by examining how many of the answers are relevant.

Relevance

Applying `agrep -1 davis` to a file of names drawn from internet news postings:

```
avis, danis, david, davie, davies, davis,
daviss, davys, lavis, navis, ravis
```

Using a human to judge which of these matches are probably correct, also considering some other matching techniques and some other names, yields:

barlow	bulow balo barlow
bloom	bloom blume bluhm
clark	klerk kluch clack clerc clarke clark
davis	daviss davyes davys davis
farah	faraj farah farra farrall farrar
hahn	hahm hann hahn hanne

Using relevance information

Given a query, a matching technique A returns a ranked list of answers.

Given relevance judgements, a top-20 ranking from A might be represented as

♢ ♢ ○ ♢ ○ | ○ ○ ♢ ♢ ○ | ○ ♢ ? ♢ ○ | ♢ ○ ? ? ○

where ♢ is "correct", ○ is "incorrect", and ? is unjudged. Assuming that unjudged matches are incorrect, this ranking is equivalent to

♢ ♢ ○ ♢ ○ | ○ ○ ♢ ♢ ○ | ○ ♢ ○ ♢ ○ | ♢ ○ ○ ○ ○

Using relevance information

Reminder: A's ranking is:

♢ ♢ ○ ♢ ○ | ○ ○ ♢ ♢ ○ | ○ ♢ ○ ♢ ○ | ♢ ○ ○ ○ ○

Suppose that method B returns the ranking

○ ♢ ♢ ♢ ♢ | ○ ○ ○ ♢ ○ | ○ ○ ♢ ○ ○ | ○ ♢ ○ ♢ ♢

To decide which method has the greater effectiveness, we need to agree on a method for scoring a ranking.

There is no objective criteria that can be used to assess a scoring method! Choice of measure is based on arguments from properties such as expected user behaviour.

Using relevance information

The traditional measures of effectiveness are precision and recall.

Precision: The proportion of the retrieved matches that are correct.

Reminder – the rankings were:

♢ ♢ ○ ♢ ○ | ○ ○ ♢ ♢ ○ | ○ ♢ ○ ♢ ○ | ♢ ○ ○ ○ ○
○ ♢ ♢ ♢ ♢ | ○ ○ ○ ♢ ○ | ○ ○ ♢ ○ ○ | ○ ♢ ○ ♢ ♢

In the top 20, A has precision 40% and B has precision 45%.

The limit of 20 was an arbitrary cut-off; we might instead say, for example, that P@10 for both A and B is 50%.

Using relevance information

Recall: The proportion of the correct matches that are retrieved.

Reminder – the rankings were:

♢ ♢ ○ ♢ ○ | ○ ○ ♢ ♢ ○ | ○ ♢ ○ ♢ ○ | ♢ ○ ○ ○ ○
○ ♢ ♢ ♢ ♢ | ○ ○ ○ ♢ ○ | ○ ○ ♢ ○ ○ | ○ ♢ ○ ♢ ♢

We do not know how many correct matches there are for this query. If we assume that 14 strings have been judged correct (not all correct were judged), then A has recall 57% and B has recall 64%.

Using relevance information

As users tend to proceed down a ranking examining answers, in turn, an alternative is use measures such as average precision (AP) – the average of the precisions at each correct matches retrieved. Correct matches that are not retrieved are assigned a precision of 0.0. AP for A is 0.387, and for B is 0.363.

Formally, if r_i is 1 (respectively, 0) if the ith item in a ranking is relevant (respectively, irrelevant), there are d items in the ranking, and there are in total R known relevant items, average precision is defined as

$$AP = \frac{1}{R} \sum_{i=1}^{d} \left(\frac{r_i}{i} \cdot \sum_{j=1}^{i} r_j \right)$$

A detailed discussion of this measure is in the readings.

Using relevance information

Different performance measures are not necessarily consistent with each other.

Given the diversity of retrieval methods used by different search engines, it is easy to have $A > B$ for one query and $A < B$ for another. For this reason, we use a sample of queries (10 is too few, 50 is adequate, more is preferable) and average the per-query effectiveness.

We then need to use a statistical test to ensure that the two samples are indeed different (to some level of confidence), rather than simply different by chance.

There are many other measures – F-score, error rate, "received operator curve" ROC and "area under ROC", discounted cumulative gain, rank-biased precision …

The importance of baselines

We use *baselines* are needed to establish whether any proposed method is doing better than "dumb and simple" – "dumb" methods often work surprisingly well.

Baselines are also valuable in getting a sense for the intrinsic difficulty of a given task – sometimes a simple method is so good that there isn't much scope for a clever method to do better.

Example: identifying what language a document is written in. Just need to check which languages have the document's commonest words.

In formulating a baseline, we need to be sensitive to the task and how the method is to be used. Example:

- In spelling correction it is helpful to give a lot of options; only one of them will be correct.
- In query correction, an option is only shown if there is strong evidence that it is correct.

The baseline should be plausible – there is no point in comparing to the worst method available.

Baselines vs. benchmarks

Baseline – naïve or straightforward method that we would expect a rich (or principled, or …) method to better.

Benchmark – established "best practice" technique that we are pitching our method against.

The word "baseline" is often used as an umbrella term for both meanings.

Building a test collection

Names from the "From:" lines of Internet news articles, hand-edited to standardise format and eliminate obvious rubbish (Darth Vader, GVR 101187-9456, The American Psychological Association).

Queries selected by generating random page numbers in the range 80–1800 and using them as an index into the Melbourne White Pages.

With each of the 125 (!) phonetic matching techniques implemented, fetch the best matches and form a pool.

Using a pair of relevance assessors to judge the pooled answers to each query (one speaking, one listening).

Three teams ⇒ three sets of relevance judgements.

Retrieval performance

Average precision (in parentheses, number correct at 200). Each column is a different speaker–assessor team.

Method	Set of judgements		
	A	B	C
Editex	23.1 (17.8)	28.5 (4.3)	26.7 (6.9)
Ipadist	23.2 (16.4)	23.2 (3.8)	24.5 (6.1)
Edit distance	20.4 (15.4)	25.1 (3.7)	22.3 (6.3)
N-gram	20.0 (16.5)	21.5 (3.5)	22.2 (6.2)
Agrep −1	16.5 (3.5)	18.7 (1.2)	18.3 (2.1)
Agrep −best	12.1 (0.8)	20.3 (0.6)	15.2 (0.8)
Soundex	9.3 (6.0)	6.9 (1.8)	9.5 (3.2)

The orderings given by the teams are nearly identical, despite the different scores achieved.

Independence of methods

Consider the query `crews` on the two best methods:

rank	Ipadist	Editex
1.	`crews`	`crews`
2.	`krewe`	`cress`
3.	`kreuser`	`clews`
4.	`crew`	`drews`
5.	`drews`	`crew`
6.	`clews`	`creps`
7.	`kruse`	`kress`
8.	`kroose`	`cross`

That is, measurement based on recall and precision can be similar even if the outcomes are very different.

As mentioned earlier, it is necessary to use a good number of queries to be confident that one method is better than another.

Summary

- Approximate matching techniques provide search with uncertainty.
- Here we have focused on spelling correction, as an example of a simple knowledge technology where effectiveness is important.
- Most methods use some form of distance measure, or do exhaustive search within an error bound.
- For the specific case of phonetic error, special-purpose methods are superior to purely alphabetic methods.

- Testing of methods requires a test data set; test systems or methods, including a baseline; human assessment of outcomes; an effectiveness measure.
- Such testing methods underpin investigation of areas such as text search and document classification.

Readings

`en.wikipedia.org/wiki/Bitap_algorithm`

`en.wikipedia.org/wiki/Spell_checker`

`www.googleguide.com/spelling_corrections.html`

“Learning a Spelling Error Model from Search Query Logs” (LMS).

“Phonetic String Matching: Lessons from Information Retrieval” (LMS).

Manning et al., chapters 3 & 8.