The Wild Thing!

Kenneth Church **Bo Thiesson**

Microsoft Research
Redmond, WA, 98052, USA
{church, thiesson}@microsoft.com

Abstract

Suppose you are on a mobile device with no keyboard (e.g., a cell or PDA). How can you enter text quickly? T9? Graffiti? This demo will show how language modeling can be used to speed up data entry, both in the mobile context, as well as the desktop. The Wild Thing encourages users to use wildcards (*). A language model finds the k-best expansions. Users quickly figure out when they can get away with wildcards. General purpose trigram language models are effective for the general case (unrestricted text), but there are important special cases like searching over popular web queries, where more restricted language models are even more effective.

1 Motivation: Phone App

Cell phones and PDAs are everywhere. Users love mobility. What are people doing with their phone? You'd think they would be talking on their phones, but a lot of people are typing. It is considered rude to talk on a cell in certain public places, especially in Europe and Asia. SMS text messaging enables people to communicate, even when they can't talk.

It is bizarre that people are typing on their phones given how painful it is. "Talking on the phone" is a collocation, but "typing on the phone" is not. *Slate* (slate.msn.com/id/2111773) recently ran a story titled: "A Phone You Can Actually Type On" with the lead:

> "If you've tried to zap someone a text message recently, you've probably discovered the huge drawback of typing on your cell phone. Unless you're one of those cyborg Scandinavian teenagers who was born with a Nokia in his hand, pecking out even a simple message is a thumb-twisting chore."

There are great hopes that speech recognition will someday make it unnecessary to type on your phone (for SMS or any other app), but speech recognition won't help with the rudeness issue. If people are typing because they can't talk, then speech recognition is not an option. Fortunately, the speech community has developed powerful language modeling techniques that can help even when speech is not an option.

2 K-Best String Matching

Suppose we want to search for MSN using a cell phone. A standard approach would be to type 6 <pause> 777 <pause> 66, where 6 → M, 777 → S and 66 → N. (The pauses are necessary for disambiguation.) Kids these days are pretty good at typing this way, but there has to be a better solution.

T9 (www.t9.com) is an interesting alternative. The user types 676 (for *MSN*). The system uses a (unigram) language model to find the k-best matches. The user selects MSN from this list. Some users love T9, and some don't.

The input, 676, can be thought of as short hand for the regular expression:

```
/^[6MNOmno][7PRSprs][6MNOmno]$/
```

using standard Unix notation. Regular expressions become much more interesting when we consider wildcards. So-called "word wheeling" can be thought of as the special case where we add a wildcard to the end of whatever the user types. Thus, if the user types 676 (for MSN), we would find the k-best matches for:

```
/^[6MNOmno][7PRSprs][6MNOmno].*/
```


Proceedings of the ACL Interactive Poster and Demonstration Sessions,
pages 93–96, Ann Arbor, June 2005.

See Google Suggests[1] for a nice example of word wheeling. Google Suggests makes it easy to find popular web queries (in the standard non-mobile desktop context). The user types a prefix. After each character, the system produces a list of the *k* most popular web queries that start with the specified prefix.

Word wheeling not only helps when you know what you want to say, but it also helps when you don't. Users can't spell. And things get stuck on the tip of their tongue. Some users are just browsing. They aren't looking for anything in particular, but they'd like to know what others are looking at.

The popular query application is relatively easy in terms of entropy. About 19 bits are needed to specify one of the 7 million most popular web queries. That is, if we assign each web query a probability based on query logs collected at msn.com, then we can estimate entropy, H, and discover that H≈19. (About 23 bits would be needed if these pages were equally likely, but they aren't.) It is often said that the average query is between two and three words long, but H is more meaningful than query length.

General purpose trigram language models are effective for the general case (unrestricted text), but there are important special cases like popular web queries, where more restricted language models are even more effective than trigram models. Our language model for web queries is simply a list of queries and their probabilities. We consider queries to be a finite language, unlike unrestricted text where the trigram language model allows sentences to be arbitrarily long.

Let's consider another example. The *MSN* query was too easy. Suppose we want to find *Condoleezza Rice*, but we can't spell her name. And even if we could, we wouldn't want to. Typing on a phone isn't fun.

We suggest spelling *Condoleezza* as 2*, where 2 → [ABCabc2] and * is the wildcard. We then type '#' for space. Rice is easy to spell: 7423. Thus, the user types, 2*#7423, and the system searches over the MSN query log to produce a list of k-best (most popular) matches (*k* defaults to 10):

1. Anne Rice
2. Book of Shadows
3. Chris Rice
4. Condoleezza Rice
5. Ann Rice

   …
8. Condoleeza Rice

The letters matching constants in the regular expression are underlined. The other letters match wildcards. (An implicit wildcard is appended to the end of the input string.)

Wildcards are very powerful. Strings with wildcards are more expressive than prefix matching (word wheeling). As mentioned above, it should take just 19 bits on average to specify one of the 7 million most popular queries. The query 2*#7423 contains 7 characters in an 12-character alphabet (2-9 → [A-Za-z2-9] in the obvious way, except that 0 → [QZqz0]; # → space; * is wild). 7 characters in a 12 character alphabet is 7 $\log_2 12$ = 25 bits. If the input notation were optimal (which it isn't), it shouldn't be necessary to type much more than this on average to specify one of the 7 million most popular queries.

Alphabetic ordering causes bizarre behavior. Yellow Pages are full of company names starting with *A, AA, AAA*, etc.. If prefix matching tools like *Google Suggests* take off, then it is just a matter of time before companies start to go after valuable prefixes: *mail*, *maps*, etc. Wildcards can help society avoid that non-sense. If you want to find a top mail site, you can type, "*mail" and you'll find: *Gmail*, *Hotmail*, *Yahoo mail*, etc..

3 Collaboration & Personalization

Users quickly learn when they can get away with wildcards. Typing therefore becomes a collaborative exercise, much like Palm's approach to handwriting recognition. Recognition is hard. Rather than trying to solve the general case, Palm encourages users to work with the system to write in a way that is easier to recognize (Graffiti). The system isn't trying to solve the AI problem by itself, but rather there is a man-machine collaboration where both parties work together as a team.

Collaboration is even more powerful in the web context. Users issue lots of queries, making it clear what's hot (and what's not). The system constructs a language model based on these queries to direct users toward good stuff. More and more users will then go there, causing the hot query to move up in the language model. In this way, collaboration can be viewed as a positive feedback

[1] http://www.google.com/webhp?complete=1

loop. There is a strong herd instinct; all parties benefit from the follow-the-pack collaboration.

In addition, users want personalization. When typing names of our friends and family, technical terms, etc., we should be able to get away with more wildcards than other users would. There are obvious opportunities for personalizing the language model by integrating the language model with a desktop search index (Dumais *et al*, 2003).

4 Modes, Language Models and Apps

The Wild Thing demo has a switch for turning on and off phone mode to determine whether input comes from a phone keypad or a standard keyboard. Both with and without phone mode, the system uses a language model to find the k-best expansions of the wildcards.

The demo contains a number of different language models, including a number of standard trigram language models. Some of the language models were trained on large quantities (6 Billion words) of English. Others were trained on large samples of Spanish and German. Still others were trained on small sub-domains (such as ATIS, available from www.ldc.upenn.edu). The demo also contains two special purpose language models for searching popular web queries, and popular web domains.

Different language models are different. With a trigram language model trained on general English (containing large amounts of newswire collected over the last decade),

```
pres* rea* *d y* t* it is v*
imp* → President Reagan said
yesterday that it is very impor-
tant
```

With a Spanish Language Model,

```
pres* rea* → presidente Reagan
```

In the ATIS domain,

```
pres* rea* → <UNK> <UNK>
```

The tool can also be used to debug language models. It turns out that some French slipped into the English training corpus. Consequently, the English language model expanded the * in *en * de* to some common French words that happen to be English words as well: *raison, circulation, oeuvre, place*, as well as <OOV>. After discovering this, we discovered quite a few more anomalies in the training corpus such as headers from the AP news.

There may also be ESL (English as a Second Language) applications for the tool. Many users have a stronger active vocabulary than passive vocabulary. If the user has a word stuck on the tip of their tongue, they can type a suggestive context with appropriate wildcards and there is a good chance the system will propose the word the user is looking for.

Similar tricks are useful in monolingual contexts. Suppose you aren't sure how to spell a celebrity's name. If you provide a suggestive context, the language model is likely to get it right:

```
ron* r*g*n → Ronald Reagan
don* r*g*n → Donald Regan
c* rice → Condoleezza Rice
```

To summarize, wildcards are helpful in quite a few apps:

- No keyboard: cell phone, PDA, Tablet PC.
- Speed matters: instant messaging, email.
- Spelling/ESL/tip of the tongue.
- Browsing: direct users toward hot stuff.

5 Indexing and Compression

The k-best string matching problem raises a number of interesting technical challenges. We have two types of language models: trigram language models and long lists (for finite languages such as the 7 million most popular web queries).

The long lists are indexed with a suffix array. Suffix arrays[2] generalize very nicely to phone mode, as described below. We treat the list of web queries as a text of N bytes. (Newlines are replaced with end-of-string delimiters.) The suffix array, S, is a sequence of N ints. The array is initialized with the ints from 0 to $N-1$. Thus, $S[i]=i$, for $0 \le i < N$. Each of these ints represents a string, starting at position i in the text and extending to the end of the string. S is then sorted alphabetically.

Suffix arrays make it easy to find the frequency and location of any substring. For example, given the substring "mail," we find the first and last suffix in S that starts with "mail." The gap between these two is the frequency. Each suffix in the gap points to a super-string of "mail."

To generalize suffix arrays for phone mode we replace alphabetical order (strcmp) with phone order (phone-strcmp). Both strcmp and phone-strcmp consider each character one at a time. In standard alphabetic ordering, 'a'<'b'<'c', but in

[2] An excellent discussion of suffix arrays including source code can be found at www.cs.dartmouth.edu/~doug.

phone-strcmp, the characters that map to the same key on the phone keypad are treated as equivalent.

We generalize suffix arrays to take advantage of popularity weights. We don't want to find all queries that contain the substring "mail," but rather, just the k-best (most popular). The standard suffix array method will work, if we add a filter on the output that searches over the results for the k-best. However, that filter could take O(N) time if there are lots of matches, as there typically are for short queries.

An improvement is to sort the suffix array by both popularity and alphabetic ordering, alternating on even and odd depths in the tree. At the first level, we sort by the first order and then we sort by the second order and so on, using a construction, vaguely analogous to KD-Trees (Bentley, 1975). When searching a node ordered by alphabetical order, we do what we would do for standard suffix arrays. But when searching a node ordered by popularity, we search the more popular half before the second half. If there are lots of matches, as there are for short strings, the index makes it very easy to find the top-k quickly, and we won't have to search the second half very often. If the prefix is rare, then we might have to search both halves, and therefore, half the splits (those split by popularity) are useless for the worst case, where the input substring doesn't match anything in the table. Lookup is O(sqrt N).[3]

Wildcard matching is, of course, a different task from substring matching. Finite State Machines (Mohri *et al*, 2002) are the right way to think about the k-best string matching problem with wildcards. In practice, the input strings often contain long anchors of constants (wildcard free substrings). Suffix arrays can use these anchors to generate a list of candidates that are then filtered by a regex package.

Memory is limited in many practical applications, especially in the mobile context. Much has been written about lossless compression of language models. For trigram models, we use a lossy method inspired by the Unix Spell program (McIlroy, 1982). We map each trigram $<x, y, z>$ into a hash code $h = (V^2 x + V y + z) \% P$, where V is the size of the vocabulary and P is an appropriate prime. P trades off memory for loss. The cost to store N trigrams is: $N\ [1/\log_e 2 + \log_2(P/N)]$ bits. The loss, the probability of a false hit, is $1/P$.

The N trigrams are hashed into h hash codes. The codes are sorted. The differences, x, are encoded with a Golomb code[4] (Witten *et al*, 1999), which is an optimal Huffman code, assuming that the differences are exponentially distributed, which they will be, if the hash is Poisson.

6 Conclusions

The Wild Thing encourages users to make use of wildcards, speeding up typing, especially on cell phones. Wildcards are useful when you want to find something you can't spell, or something stuck on the tip of your tongue. Wildcards are more expressive than standard prefix matching, great for users, and technically challenging (and fun) for us.

References

J. L. Bentley (1975), Multidimensional binary search trees used for associative searching, *Commun. ACM*, 18:9, pp 509-517.

S. T. Dumais, E. Cutrell, et al (2003). Stuff I've Seen: A system for personal information retrieval and re-use, *SIGIR*.

M. D. McIlroy (1982), Development of a spelling list, *IEEE Trans. on Communications* **30**, 91-99.

M. Mohri, F. C. N. Pereira, and M. Riley. Weighted Finite-State Transducers in Speech Recognition. *Computer Speech and Language*, 16(1):69-88, 2002.

I. H. Witten, A. Moffat and T. C. Bell, (1999), *Managing Gigabytes: Compressing and Indexing Documents and Images,* by Morgan Kaufmann Publishing, San Francisco, ISBN 1-55860-570-3.

[3] Let $F(N)$ be the work to process N items on the frequency splits and let $A(N)$ be the work to process N items on the alphabetical splits. In the worst case, $F(N) = 2A(N/2) + C_1$ and $A(N) = F(N/2) + C_2$, where C_1 and C_2 are two constants. In other words, $F(N) = 2F(N/4) + C$, where $C = C_1 + 2C_2$. We guess that $F(N) = \alpha\ \text{sqrt}(N) + \beta$, where α and β are constant. Substituting this guess into the recurrence, the dependencies on N cancel. Thus, we conclude, F(N) = O(sqrt N).

[4] In Golomb, $x = x_q\ m + x_r$, where $x_q = \text{floor}(x/m)$ and $x_r = x \bmod m$. Choose m to be a power of two near ceil(½ E[x])=ceil(½ P/N). Store quotients x_q in unary and remainders x_r in binary. z in unary is a sequence of z−1 zeros followed by a 1. Unary is an optimal Huffman code when $\Pr(z)=(½)^{z+1}$. Storage costs are: x_q bits for $x_q + \log_2 m$ bits for x_r.