Dynamic Ranking

Nicole Hamilton
November 2004

Author's Note (November 2026): *This paper was originally written and circulated internally at Microsoft in November 2004, just after code freeze for the first release in January 2005 of what became the Bing engine. It describes the query compiler and dynamic ranker I invented and wrote for that first release. Popular bag-of-words ranking techniques such as tf-idf [1] and bm25 [2] considered only word frequency, not position. Instead, this ranker identified groups of words called spans and classified them as features—for example, as an exact phrase in the title—in a very efficient single pass. Scoring used a linear combination with bucketing of nonlinear and non-monotonic features like title length, and separate scoring of matches in anchor, URL, title, and body text. This paper also describes training problems we were having with the Flying Dutchman tuner, especially the critical decision to make guesses about unlabeled pages when scoring sets of results. It recommended training only on the ordering of labeled pairs and investigating the use of a neural net instead of a linear combination for scoring. The text was lightly anonymized in 2018 for instructional use in my System Design of a Search Engine course at the University of Michigan, with Appendices A and B added to provide students with the full query language grammar and set of ranking parameters, and a References section added in 2026 for archival context. LinkRank refers to an implementation of PageRank [3]. The Index Stream Readers (ISRs) and constraint solver followed the design introduced at AltaVista [4]. The span algorithm is the subject of U.S. Patent 7,254,576 [5] and the heuristics described here provided the 569 features used in Burges et al., "Learning to Rank using Gradient Descent" (ICML 2005) [6].*

Ranking is the process by which a search engine decides the order of results. This paper will outline how that's done in a modern search engine and describe one particular algorithm and heuristic scoring method investigated by the author.

Search engine ranking is typically considered as two parts, static ranking and dynamic ranking. Static ranking is an estimate of the importance or quality of the page *without regard to the query*. For example, the New York Times home page is certain to be a high-quality result if it matches at all and we can recognize that with a hand-curated list of important sites on the web, as Yahoo! did in its early days, or by using algorithmic means to identify important sites, e.g., by noticing that many other sites link to it. Other static attributes include the length of the URL, the title or the page, the domain, e.g., .gov or .edu versus .biz, whether it contains images, lots of links, invisible text, pornographic content, the vocabulary level, spelling errors, and so on. Any data that would inform on the quality of a page independent of a query might be a candidate to be collected for static ranking.

Dynamic ranking, by contrast, is all about making an estimate of the quality of the page as a result *for a particular query* through some means that considers both the static information about the page and the quality of the match between the query and the page content. Though done by an algorithm running in a computer, the objective is to order the results the way a human would.

Basic Dynamic Ranking

The dynamic ranker in a typical search engine is given a query string and a handle to an inverted word index of some number of pages on the web. Typically, an entire engine is composed of thousands of machines, e.g., ***m*** rows of ***n*** machines, each with an index covering 1/***n***-th of the pages known to the engine. As queries arrive from the web service front end, a row is selected, perhaps randomly or by queue length, and the query is broadcast to all the machines in that row. The results from the entire row are collected and top results merged together into a final list.

The dynamic ranker compiles the query, searches the index for matching pages and returns a list of the ***n*** best pages, each with a score representing its estimated quality as a result. It may also be instrumented to return various levels of debug information, e.g., to allow the scoring calculation be re-run off-line, including redoing it with different scoring parameters.

The query string is a canonicalized form of whatever the user has typed. In Japanese and other Asian languages, the queries are commonly word-broken, e.g., 知多半島道路 becomes 知多+ 半島+ 道路, literally, Chita peninsula road. In Western languages, a naïve word-breaker simply breaks on and discards most punctuation and folds the character set by discarding accents and lower-casing the characters to match the way words are typically indexed.

Typically, the front end will also have prepended some market-specific augmentation string onto the canonicalized query, e.g., to specify that Japanese language pages are preferred in the Japanese market.

Scoring is intended to obey the property that if you ever find a page a human judge would rate as a better result, you will give a higher score, and if you find a lesser page, you will give a lower score. This has to be done without knowing how many other pages match, what they look like or even if they exist. It must also be reasonably consistent across individual nodes so as to allow the results from different nodes to be compared and merged.

Dynamic rankers typically only attempt to guarantee ordering for rank scores; they make no attempt to guarantee linearity. If one page scores .6 and another .3, that means the first is probably better than the second; it does not mean it's twice as good.

The Index

The index of the web works pretty much like the index at the back of a book:

cat 5, 7, 9, 83, 107
dog 6, 12, 83, 107

If you were asked to find all the pages with both the words cat and dog and had to read the whole book, that could take a while. But with an index, you can simply scan the lists to find the intersections.

Conceptually, all the pages on the web (or at least, those in the particular slice within a given index) are laid out end-to-end, numbering the word locations. The lists of locations for each word, also called posting lists, are generated, and a dictionary interface is provided so that, given a string, you can get the list. This is called an inverted word index. The location numbers may be kept as (document, offset) pairs or as offsets from the beginning of the index. Keeping them as offsets from the beginning of the web is more efficient. Internally, the offsets are further compressed by encoding them as deltas from the previous locations, using a variable-length encoding, e.g., UTF-8, and by introducing a higher-level index of synchronization points.

A constraint solver, first used by Mike Burrows at Altavista, allows the creation of objects called Index Stream Readers (ISRs) that can fast-forward through these lists to the next occurrence that matches its constraints, e.g., to the next location of a word, or the next occurrence of a phrase of sub-ISRs, or seek to the first posting after a particular location. Except when seeking, the ISRs only move forward through the index. There is no "previous" function.

The information stored in an individual posting entry depends on the type of token. Individual word occurrences can have attributes, e.g., bold or normal or possibly a count. Occurrences in anchor, URL, title and body are distinguished into four metastreams by decorating the search word in some way that can't collide with anything the user is allowed to type. This same trick is also used to insert special metawords marking the end of a document or the end of an anchor text phrase or to indicate membership in some category, e.g., adult content. The decision of whether to encode as an attribute or a decorated word usually turns on whether we need to search based on the distinction.

The end-of-document token contains the document length plus other static information about it, e.g., LinkRank, total amount of anchor text, etc. Putting the token at the end of a document rather than the start has the advantage that if a match is found, the document information can be found using only a forward search. And given the document length stored in that token, it's possible to seek back to the beginning.

The major parts

Dynamic ranking is typically broken into three parts: Compiling, matching and scoring.

The query language has specific semantics, e.g., that a 2-word query means you need both words, i.e., that it's an AND, which allows us to say whether a page either does or does not match. Pages that do not match are not usually scored, both because scoring pages is expensive and as an important simplification: It avoids the question of what a score would even mean for a page that does not match.

Other features of the query language may include the ability to change the scoring parameters (e.g., to make an exact phrase in the title much more important) and to specify conditions not required for the match, but which, if they occur, tell if the page is better or worse, e.g., preferring Japanese language and Japanese location pages in the Japanese market or that a page not be porn, allowing market-specific augmentation. Scoring parameters are typically read from a file containing values calculated offline using a separate tuner.

The compiler uses a top-down recursive descent parser to build an ISR structure to find matching pages efficiently, simply by calling the Next( ) function. But as might be expected, the order in which you search and your optimism about finding something better if you keep on searching affects the number of pages you both match and score. The idea is to raise the hurdle as high as possible as fast as possible, aggressively prune away whole chunks of search space that can't be productive and make quick estimates of the best score a page could realistically achieve if you did score it, without actually having to do it.

Scoring

Matching pages are found via the constraint solver, but to score a page, it's the Dynamic Ranker's job to walk each word ISR from one location to the next in an extraction phase.

During extraction, the individual ISRs are reset to the beginning of the document (if some of them were part of a phrase required to make the match, they may not be pointing at their own individual first occurrences) and then advanced in a single forward pass over the document, incrementing a flurry of low-level counters, e.g., the number of times each word occurred.

Following extraction, the results in the counters are scored as features according to a set of parameters that amount to saying, if you got this feature, e.g., an exact phrase in the title, the page gets so many points, which could be positive or negative.

The strategy behind features is to create a large number of cheap heuristics which are expected to provide evidence of whether a given page is or is not a good match. For any given page and query, any particular heuristic may be misleading, but that's why there are a lot of them. That said, it's a pretty interesting question what to measure. For most queries, the garbage outnumbers the good results, so the best heuristics tend to be those that can cherry-pick a small number of really good pages out of a much larger set of also-rans.

Extraction

For extraction, the query is flattened to a simple list of the search words, in-order walking the parse tree. Though each of these queries matches a different set of pages, all of them flatten to the same list of search words for purposes of scoring:

apollo moon landing
(apollo | moon) landing
"apollo moon landing"

A central notion of the extraction is something called a ***span***, defined as a complete set of all the words in the query found within a single metastream.

If it's a multiword query and each of the words occurs multiple times within the document, it's not possible to visit all possible combinations in a single pass, defined as each ISR only advancing. (Consider the query "a b" and the document "a b a b"; it's not possible to visit both "**a** b a **b**" and "a **b a** b" if both ISRs are only allowed to advance.)

But an Occam's razor of ranking, just as in Poker, is that the less likely or more exact the match, the more significant. If you have a choice of how to define a set of things you propose to count, it's generally better to define a set that's small and most likely to represent especially good matches. This also tends to be faster.

One solution is to choose the rarest word as an anchor and advance the ISR corresponding to that word through the document. At each occurrence of the rarest word, the other ISRs are advanced to position them as close as possible to the desired location in this flattened list, using one stage of lookahead. If there is at least one occurrence of each of the words somewhere in the metastream, that's called a span, which might be further distinguished as ordered, short, an exact phrase, etc. Only the rarest word is guaranteed to be a unique occurrence in each span; the other words may be reused.

Doubles and triples, meaning various combinations of just two or three of the words in the query, one of which must be the rarest, are also counted. (The reasons for requiring that doubles and triples must include the rarest word are the same as above for spans, namely, to count what's most important and to do it a single pass.) Finally, individual word occurrences are counted.

Parameterized thresholds for short vs. not short, frequent vs. not frequent, etc., allow the spans to be classified.

Score Calculation

The actual score is calculated as a linear combination of features, which may be thought of as:

$$R = \Sigma\, Ci(Q) * Ai(P, Fi) * Si(Fi)$$

Where:

- R = Overall Rank
- Q = The Query and its characteristics, e.g., the number of rare vs. common words in the query.
- P = The Page and its characteristics, e.g., the number of words in the URL or title.
- Fi = An arbitrary feature observation, e.g., an exact phrase in the title, or a raw value, e.g., raw LinkRank.
- T = A set of thresholds for defining terms like rare, common, short or near.
- Ei = The extraction function which determines if feature Fi was observed.
- Si = Scaling for feature Fi from the raw number space of the feature into a nominal 0.0 .. 1.0 range.
- Ai = Attribute scaling for feature Fi, possibly dependent on the characteristics of the page P.
- Ci = Coefficient for feature Fi, depending on the characteristics of the query Q.
- Fi = Ei(Q, P, T)

Most features are binary; they either occur or do not occur in a very unambiguous way; the Si scalings for these features are identity functions. The features themselves can be thought of as probability buckets; we may not know the probability of a given query + page combination falling into that bucket to put a number on it, but we can reasonably assert that it does have some specific probability and that, if it occurs, might have some binary significance.

The exceptions are LinkRank, which is scaled linearly, and numbers of occurrences in anchor text, which we scale as a log. (The actual scaling is specified via a configurable list of Quanta, representing a monotonic sequence of n raw values corresponding to ***n*** evenly-spaced scaled values.)

The various Ai values represent an attempt to reduce the number of unique features we need to parameterize by grouping them based on some cross-sectional attribute. For example, a phrase match in the URL might be in the domain, somewhere along the path or in the leaf, or made up of some combination; each combination having some unique statistical significance. Similarly, the significance of a phrase match in the title quite commonsensically depends on the length of the title. If every possible combination were exposed as a separate feature, the numbers would be unmanageable. No human could possibly find a pattern and since many combinations could be quite rare, even an automated tuner might have difficulty finding examples in the training data.

The various Ci values reflect the relative point-value assignments for each feature, e.g., an exact phrase match in the URL versus the title, plus some statistics about the query, namely, the

number of words and the number of rare vs. common words it contains: As the number of words, and particularly, the number of rare words in the query increases, we shift the mix of Ci to less LinkRank and more quality of match, reflecting the commonsensical notion that the more specific the query, the more likely the user is searching for a specific, less generic result.

The objective is that the values for Ci, Ai and even for the Si quantizations and the thresholds will be chosen by an automated tuner, trained using queries taken from the search engine log together with human judgments of the page quality of the results shown by several competing engines. The basic algorithm of the tuner is to push as many perfect and other really good pages onto the results as it can, maximizing the expected value for our overall relevance. The quality of any unjudged pages that appear in the results as a parameter is varied up or down can be either estimated or the unjudged pages can be ignored.

Lessons

Here were some lessons learned from the investigation.

1. There are only two ways to get better at ranking:

    a. Have more or better information.
    b. Make better use of it.

2. Matches in anchor text are most significant, followed by matches in the URL, title and body. The exception is Japanese, where matches in the URL are much less useful due to the mismatch between Kanji or Katakana queries and any possible ASCII translations in the URL.

3. The quality and structure of the match, e.g., that the words be in order and close together, is more important than simply having lots of occurrences of the individual words. One surprising confirmation of this point was an experiment with tf/idf (term frequency/inverse document frequency.) Tf/idf proposes to calculate similarity between a query and a document as an elegant dot-product of vectors representing the frequencies of all the words in the document or the query. It does not, however, consider whether multi-word matches are close together or in order. Tf/idf is considered an important work-horse metric and we expected it to be significant contributor to overall rank.

    But adding tf/idf offered minimal improvement to the results, even on one-word queries. Given that the added calculations to calculate various static "vectors" used by tf/idf for each page were slowing the crawl by 30%, tf/idf was disabled entirely. Scoring isn't so much about estimating interesting statistics as it is about identifying interesting distinctions in quality.

4. LinkRank did not turn out to be nearly as important as originally expected. Depending on how it's measured, static rank turned out to be only about 5% to 10% of the overall rank. But it's possible the implementation was naïve.

5. Trying to choose features that can't be gamed by the spammers is an exercise in futility: Virtually anything can be gamed by the spammers, who know all about how to generate arbitrary amounts of static rank and blizzards of anchor text.

6. It's dramatically easier to tune a binary feature representing an observable event that either does or does not occur than it is to tune one that ostensibly encodes a range of "goodness" values. Tuning the various Ci to set the basic number of points at stake for a feature is an easier task than setting the Ai to reflect more subtle attributes of the feature, which, in turn is an easier task than setting the Si scalings and the thresholds.

7. It's difficult, if not impossible to estimate the possible improvement in relevance that might be gained by a possible new feature without actually building it or at least simulating it. Since we don't yet have the infrastructure for simulation, features had to be chosen either by intuition or because we had a specific situation (e.g., spurious anchor) we hoped to correct.

8. Lacking numbers on a proposed new feature, it's helpful to think about the problem in terms of the "blind spot" created by not having it: Without this information, when would the ranker not be able to tell the difference in result quality between two pages that it should be able to distinguish? How egregious would that be and how often would it happen?

9. If the ranker has to calculate some number for any possible reason, it's important never to throw it away. Almost every intermediate result or low-level counter value is of interest downstream for debug or off-line recalculation of the overall rank by the tuner as it adjusts the scoring parameters.

10. The most common user complaint is, "I ran this query and the results look wrong." But with 700 or so individual features in the ranker (depending on how you count), it's nearly impossible to say what, if anything, is actually wrong or to explain the result as correct (if unexpected) without digging through the calculation.

Analysis

Thinking about how to improve based on this experience, these seemed to be most interesting areas for investigation:

1. Complaints about proximity, e.g., that it doesn't recognize the importance of finding exact phrases in the title or proper names as exact phrases.
2. Correctly ordering results when the best result is only of middling quality.
3. Japanese relevance.
4. Word-breaking, including incompatibilities between queries and URLs.
5. Static rank may be a disappointing contributor to overall rank.
6. Difficulty analyzing the reasons for a poor result.
7. Difficulty estimating the effects of any proposed changes.

Proximity

The biggest category of complaints is typically that the ranker didn't find results with the words close together the way people expected, often labeled proximity. Variations on this are that other engines did a better job of finding results with an exact phrase in the title and that, in searches for people's names, it didn't get first and last names together very well.

Complaints about not finding exact phrases in the title may be a training problem: By boosting the points awarded for an exact phrase in the title high enough, it's possible to get as many you want. This plus other evidence, e.g., negative points awarded by the tuner to hits in the body, suggest we have a methodological error. If no really strong page with lots of anchor text and a good hit in the URL is found, the remaining pages having hits only in title or body weren't scored very well. I don't think this was about proximity; I think it's about ranking middling-quality results, discussed later.

Poor results with names usually involved silly splits. A page where "bob smith" matched "john **smith** … **bob** jones" would get ranked over other pages that actually said "bob smith" as an exact phrase. Often, rerunning the query with quotes around part of it produced better results (by virtue of changing the selection set, even though scoring was unchanged.) This was also observed on queries that joined two concepts, e.g., "NJ state flower"; putting quotes around "state flower" gave much better results.

But unfortunately, proximity also became something of a catch-all. If our results were poor in some regard, proximity was always a handy suspect if only because it was a word everyone knew: Many complaints cited proximity, but in a hyperlinked map of user complaints, virtually every complaint categorized as proximity was also categorized as matching one or more other complaint categories. Proximity was cited repeatedly as ***the*** problem in Japanese, causing us to focus almost exclusively on possible problems in word-breaking; it may not be fully clear why we're doing so poorly in Japanese or what we should do to fix it (more on that later) but it does look like proximity and word-breaking were not the most likely problems.

In those cases where proximity is genuinely a problem, here are some hypotheses:

1. The extraction centers on just a single rarest word as a proxy for what the query is all about. But it may not perform well if there's more than one important word or if the rarest word isn't what the query is really about.

2. We have a bug in either the ranker or the tuner. It is software, so we know there are bugs, we just haven't discovered them yet. But the likelihood of finding something new in the ranker that could suddenly give us a big boost is shrinking. The tuner has had less peer review, however, and probably merits closer inspection.

3. Stop word elimination is eliminating words needed for exact phrase matches.

4. We need some new features. Undoubtedly. But answering whether this is the real problem asks that we first identify the characteristics of the failure and can invent a feature that detects the condition.

5. We are overtraining. Here is how that could happen: The judged query/page pairs used to train the tuner are scraped from various search engines. All good search engines should produce good results, e.g., with lots of exact phrases, especially in the title and with lots of doubles and triples. So it's possible these are overrepresented in our judged results, even on pages judged poor. This would cause the tuner to underestimate the value of these features in distinguishing result quality.

6. We are using the wrong goal function in tuning.

7. The static rank is to blame.

Anchoring on the single rarest word

Given that queries average only about 2.4 words, the rarest word is usually a pretty good proxy for what the query is all about. But it may not work well in some cases.

Consider a query for person's name and location: "nicole hamilton redmond." If "redmond" happens to be the rarest word, we'll end up scoring only meaningless doubles "nicole redmond" and "hamilton redmond," ignoring the "nicole hamilton" doubles a human would care about.

Another example would be a query for "excruciating pain." "Excruciating" is rarer than "pain," skewing the results; we get much better results searching just for "pain."

This might be attacked through a dictionary approach and by taking a linear combination of scores anchored on each of the suspected important words.

Linear combinations of scores

If we were willing to make multiple passes over a document, the additional passes could be used to extract spans anchored on the second or even the third rarest word in the query. Since scoring amounts to discovering that a page has fallen into some "probability bucket," a linear combination of scores where the coefficients represent the estimated likelihood that a given word is what the query is "really" about should be safe.

Dictionary approaches

A dictionary might tell us what goes together or that frequency in our index notwithstanding, some words are simply more important than others. For example, it might tell us that "pain" is a noun and (we might postulate) more important than "excruciating," which is just an adjective.

We might also use a dictionary to decide what parts go together, telling us, e.g., that "nicole" and "hamilton" are common first and last names and "redmond" is likely a city name, something altogether different.

The scoring algorithm becomes more difficult; it's no longer a matter of simply flattening the parse tree to score a result.

Multiple levels of binding between words

Currently, all queries are flattened for scoring. I think it should be possible to do a better scoring by considering more of the structure of the query. But I don't think it's possible with a naïve algorithm; it takes more work.

Consider these queries:

> hawaii (hotel | resort)
> mortgage "interest rates"

To find matching pages, the compiler builds a tree-structure of ISRs representing the criteria. ISRs that expect sub-ISRs as operands (e.g., an OR) don't generally care whether the operand is a word ISR or something more complex. So it's an easy experiment to run the algorithms for extracting spans and scoring by treating the OR term or the phrase as if it was a word. And that's what I did when I first wrote the compiler. I expected it to be a great heuristic. It was terrible. Any queries that actually used quotes or OR'ing instantly got visibly worse. (Admittedly, this was before the automated tuner existed, so my experiments were done with manual tuning.)

I think this naïve strategy failed for several reasons:

1. With an OR, it's better if you get both terms, but this scheme couldn't distinguish that.
2. With a phrase, it's better if you get the individual words also scattered through the document, but this scheme also could not distinguish that. Experiments in Japanese seemed to show the same behavior for the results of word-breaking, that also finding the individual words was better than only finding the exact phrase.
3. It's very difficult to estimate probabilities except in a naïve manner. How do you estimate the likely number of occurrences of a phrase, given occurrence data only on the individual words? How do you decide what's "rarest" when the terms are abstract?

Ranking middling-quality pages

The approach worked well on queries where there actually was a perfect or excellent result but was less effective if the best available page was only good or fair.

Here are some hypotheses:

1. We don't have enough labeled pages to adequately train the tuner.
2. The tuner needs a better estimation methodology for unlabeled pages.
3. We are using the wrong goal function in tuning.
4. Where you end up in tuning depends on where you start.
5. We should score these pages differently.
6. We need some new features.
7. Our static rank is to blame.

Labeling the Web

As the tuner adjusts a given scoring parameter up or down, some pages come onto the result sets and some leave. A given result set is scored using the relevance methodology, which, for each query, measures the overall quality of a result set on a 0 .. 1 scale as a fraction of the estimated best possible score, with the number of points given to each individual result based on the quality of the page and its position in the set.

But only some of the pages have judgments and the tuner has to estimate the quality of the pages for which it lacks judgments. The estimate is adjusted somewhat by the characteristics of the page, so that the estimate is higher for pages in the desired language and which have other match characteristics thought desirable, e.g., an exact phrase. But the bottom line is that the tuner has to make an estimate.

If the best result available for a given query is a known "perfect" or "excellent" page, the chance we might incorrectly estimate an unjudged page relative to that known page is low. But if the best result is only of middling quality, the chance of an error – especially, one that affects the tuner – is much higher.

Here's how that could happen: For any given query, matching pages likely fall into a pyramidal distribution: Very few really good ones, more that of middling quality, and an enormous number that are poor. Because we're scraping and judging results from three search engines whose results are presumably biased toward finding really good results if they exist at all, we can reasonably expect that if a "perfect" or "excellent" page exists at all, it will have been found and judged.

We certainly won't estimate an unjudged page as a "perfect," so if we have a "perfect," almost any reasonable estimate of an unjudged page will indicate that it's less desirable than the perfect we already know about. That estimated relationship (i.e., which page is better) will

almost always be correct, meaning we'll have correctly instructed the tuner as to which page should go into the results and which page should be dropped.

But if the best available result is merely "good" or "fair," there may well be a lot of other pages out there, only some of which have judgments, that are just as good or even a little better than the one we know about. In this case, even small errors in the estimated quality of an unjudged page may result in the tuner choosing the wrong page.

The problem could be solved if only we had enough pages labeled, but since this currently is being done by human judges, it's unrealistic to think we could ever get judgments for a sizeable portion of the web (unless we could get it by other means, e.g., click data.)

Better, more workable alternatives may be to improve the estimates or change the tuner's goal function.

Improving the Estimate

Here are a couple ways we could try to improve the estimate:

1. We could look for trends as a given parameter is adjusted up or down among the judged pages coming and going from the results to see what it tells us about the unjudged page.
2. We could do a more sophisticated evaluation of the observed features of the unjudged page.

Using trends to make the estimate

I hypothesize that if changing a scoring parameter by a small delta causes some number of pages to enter a given result set, some labeled and some unlabeled, that the unlabeled ones are likely to be similar in quality to labeled ones that came in at the same time. A similar hypothesis might be made about the pages which leave the result sets. If changing a given parameter causes 5 new pages to enter the result set for a given query and 4 of them are known to be "good," it seems more likely the 5th one is also a "good" than if the 4 known pages are "poor."

If the behavior of pages entering and leaving the result sets is tracked as the parameter is varied, it may be possible to further refine the estimate from the observed trend.

The problem is that, unless the result set size is increased well beyond the current top 10 or 15 for a given query, this approach may not be feasible. Here's why:

1. Any given small delta may only affect a small number of queries and only by changing one or two pages in the result sets for those queries. Since this approach depends on the assumption that an unlabeled page that enters will be of similar quality to the labeled pages that entered at the same time, the approach suffers if there aren't any labeled pages entering at the same time. Identifying the trend is most important at the precisely the time when you have the least evidence to do it.

2. At each iteration, the tuner chooses to adjust the parameter which gives what it thinks is the best improvement, without regard to which parameter was adjusted on the preceding round. But this could make it difficult to estimate any trend since the result of changing a parameter (e.g., whether it brings in good pages or bad ones) probably depends on the existing settings of all the other parameters. (Considering this as a hyperspace of feature dimensions, the slope of the relationship between an individual feature and likely result quality almost certainly varies over the space.)

A better estimate using page characteristics

The earliest design iteration of the tuner estimated that all unjudged pages were uniformly of the same quality regardless of anything known about the page. In what was called "promising proximity," the estimate was improved based on a few basic characteristics of the page itself. In effect, the estimate is now being done by a "mini-ranker."

This is a pretty reasonable approach. The underlying assumption, that you can consider a bunch of features and, from that, identify good results, is the whole idea behind ranking.

But promising proximity only considers a small number of features and the scoring values are hard-coded, not parameterized, and appear to be initial guesses, neither tested nor tuned, so it's a pretty naïve ranker. Fortunately, we already have a much, much smarter ranker available that's able to consider many hundreds of features and for which we have a lot of tuning data already.

One way to improve the estimate would be to replace the toy ranker in the tuner with our real ranker. To make this work, the algorithm should use the tuning values from the previous or, even better, a known good iteration to calculate an overall rank for the unjudged page. Assuming we can meaningfully estimate a relationship between overall rank scores and likely human judgments, knowing the overall rank would tell us the likely human judgment for the unjudged page. (I prefer the known good set primarily because calculating the relationship every time between rank scores and human judgments would be costly. A known good set also tends to introduce a bit of inertia into the system, which is both good and bad: It's not as easy to walk off into the weeds; but it's also more difficult to make progress.)

The Goal Function

A different solution to the problem of improving the estimate would be to choose a goal function that explicitly avoids requiring any estimates at all about unjudged pages. One possibility would be to train the tuner using only queries for which we assert we've identified a *definitive* result, scoring how well we did at getting those definitives to the top of the result sets. This works because, by definition, we are asserting that sight-unseen, we believe no better page can possibly exist for a given query. The problem with this approach is:

1. The characteristics which discriminate between really outstanding pages and everything else (e.g., lots of hits in anchor text) may not be the characteristics which discriminate between two mediocre pages, one of which is only slightly better.

2. It's harder to find examples of definitives than it is to find examples of two matching pages of middling quality, meaning the training set size is sure to shrink.

A better alternative goal function would be to consider only those queries and pages for which we do have judgments and maximize the number of correctly ordered pairs minus the number of incorrectly ordered pairs (or some variation thereof.)

I hypothesize that this goal function would likely have the effect of both (a) making results for queries where the best result is of only middling quality more significant to the tuner and (b) decreasing the likelihood of training set errors on these queries.

But if our ultimate objective is the highest possible relevance and we choose any goal function other than relevance, doesn't that inevitably mean we will sacrifice relevance?

I hypothesize that it will not. Here's why:

1. The current goal function isn't the relevance score; it's the ***estimated*** relevance score. We don't find out the actual relevance results until any unjudged pages are submitted to human judges. I expect the estimates are likely optimistic, mostly as a result of the large "error bars" associated with the estimation methodology.

2. If we always order result pages correctly, it's not possible to get a better relevance score with a different order.

The whole point of the dynamic rank calculation is to produce an overall rank score that's transitive. If the ranker ever encounters a better page, it should calculate a higher score and if it encounters a lesser page, it should calculate a lower score. And since we intend to merge results from multiple index nodes, we require the overall rank be calculated without knowing what other matching pages exist or even if there are any such pages.

Goaling the tuner on ordering relationships rather than estimated relevance scores should, in effect, reduce the noise in the feedback loop between overall rank values calculated from a given set of scoring parameters and changes to those parameters.

Japanese Relevance

I consider these the most likely hypotheses:

1. We need a dictionary to translate or transliterate from Kanji and other Japanese language queries into ASCII so we can match on URLs.
2. The Japanese web is not yet well-developed for queries.
3. Training data lacks perfect and excellent examples. (There aren't as many really good pages in Japanese as in other languages for many queries.)
4. Our word-breaking algorithm is poor.

I consider these less likely:

1. We need to consider in-domain anchor text for Japanese pages.
2. Links on Japanese pages point to the root of a domain, not to the actual page of interest.
3. Japanese pages don't cross-link as much.
4. Tail queries are more prevalent in Japanese.

Dictionary needed for matching Japanese URLs

There's strong evidence for this. To better understand the reasons for our poor performance in Japanese, we invited native speakers from our Japanese lab to Redmond and spent several days examining individual Japanese queries.

What we frequently observed was that a good result would have a phoneticized spelling or trivial English translation of one of the search words. But without a dictionary to translate or transliterate, we cannot make the match. In other languages, the URL is the second most important metastream; not making these matches in Japanese is almost certainly hurting us badly.

Here are some examples:

Japanese Query	Desired URL
岡山大学(Okayama University)	www.okayama-u.ac.jp
岡山県 (Okayama prefecture)	www.pref.okayama.jp
東芝(Toshiba)	www.toshiba.co.jp
知多半島(Chita peninsula)	www.japan-net.ne.jp/~chita/
関西スーパー (Kansai Supermarket)	www.kansaisuper.co.jp/

The Japanese Web

With guests from our Japanese lab here to help, a repeated scenario was that we'd run a query that'd been identified as poorly performing against three popular engines, and ask which was

the best result. In English, this is almost always a fairly straight-forward exercise: Quickly scanning the lists, one or two URLs quickly pop out as distinctly better than the rest because, quite simply, they're obviously pretty decent results.

But our Japanese guests would often pore over the results, clicking links and reading everything, carefully trying to decide, e.g., between a couple of blog pages. It was difficult to explain that if the only way to find a good page was by poring over them, then likely enough, there weren't any that were very good.

An example was a query for "天体望遠鏡," which translates as "astronomic telescope." The results are absolutely terrible; the top results are retailers selling inexpensive hobbyist telescopes. By contrast, corresponding queries in English easily produce serious results at .edu and other sites discussing the Hubble telescope, etc.

I hypothesize that the development of the Japanese web may have been retarded, in a chicken-or-the-egg fashion, by the fact that none of the 3 top search engines seems able to consistently produce good results in Japanese (even though clearly, the others do better than we do.) The western web took off commercially when early engines like AltaVista made that possible. Without good engines able to do a good job on Japanese pages, there may not be nearly so much incentive to develop a great website in Japanese.

If this hypothesis is true, then quite possibly part of the key to improving Japanese results may lie in solving the problem (already discussed above) of doing better on queries where the best possible result is only of middling quality.

Things to investigate

This is not exhaustive:

1. Build dictionaries. Here are some examples:
    a. A dictionary to translate/transliterate Japanese queries so we can make matches in URLs.
    b. A thesaurus of synonyms, e.g., for stemming, alternate spellings and close matches, e.g., "cat" and "kitten."
    c. Lists of words that cluster together so we can discern pages that have a lot of occurrences of related terms. We already know from a public talk given at UW that a competitor is attempting to do this on a research platform they call Map/Reduce.
    d. A dictionary to translate between phrases and acronyms. We already know a competitor is doing this. (For example, the query "nicole hamilton rkba" turns up matches to "right to keep and bear arms" that can only be made with a dictionary.) One way to create such a dictionary might be by searching for one-word acronyms that exactly match whole phrases in the anchor text associated

with an individual page; if you find lots of matches in lots of pages, you conclude it's a valid acronym.

e. A dictionary to tell us parts of speech, e.g., to allow us to bias the choice of the rarest word toward the rarest noun.

2. Run separate extraction passes for each of the significant words and on more tightly-bound subgroups of words (e.g., a quoted section) in the query.

3. Enhance the scoring to add multiple levels of binding between the words to give more weight in scoring to matches against quoted strings, the results of word-breaking or to other groups of words we identify as more "connected."

4. Improve the encoding for anchor and URL. These are our most important sources of information. Here are ways we could improve:

    a. Distinguish hits in the leaf of a URL from hits along the path.
    b. Distinguish phrase hits within a single directory level along the path.
    c. Index in-domain anchor, but with a different attribute, so it can be distinguished by the ranker.
    d. Attempt to distinguish more reliable anchor from less reliable and negative citations from positive ones.

5. Try these strategies for improving the tuner results:

    a. Use the real ranker to estimate judgments for unjudged pages rather than the toy "promising proximity" feature.
    b. Goal the tuner by "right minus wrong" ordering relationships rather than estimated relevance. Binary ordering data may be developed from either direct HRT judgments or from observations of deviations from expected click profiles. (For example, we know that users tend to click the #1 result more often than the #2 result just because it's #1; when the #2 result is consistently more frequently clicked, we can infer the #2 result really is better.)

9. Explore neural net techniques. I expect it may be possible to improve our relevance by using a neural net to score the counters and features we either currently or could expose. I judge this to be the most promising alternative to our current method of scoring as a linear combination of individual features.

10. Build some new static ranks. Each static rank characteristic should be stored separately in the index. Here are some characteristics we could bleed across the web by appropriate settings of the jump vector and the decay rate:

    a. Traffic, taken from click data.
    b. Negative information can be bled backward. You shouldn't be able to get better by linking to something, but you could go down if you point to junk.

c. Buckets for types of links. For example, we might try to distinguish positive versus negative citations by some means of classification, then bleed that.

References

(Added November 2026 for archival context; the original 2004 internal memorandum contained no formal bibliography.)

[1] G. Salton and C. Buckley, “Term-weighting approaches in automatic text retrieval,” *Information Processing & Management*, vol. 24, no. 5, pp. 513–523, 1988.

[2] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, and M. Gatford, “Okapi at TREC-3,” in *Proceedings of the Third Text REtrieval Conference (TREC-3)*, NIST Special Publication 500-225, pp. 109–126, 1995.

[3] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank citation ranking: Bringing order to the web,” Stanford InfoLab, Technical Report 1999-66, 1999.

[4] M. Burrows, “Method for parsing, indexing and searching world-wide web pages,” U.S. Patent 5,991,756, Nov. 23, 1999.

[5] N. Hamilton, “System and method for dynamic ranking,” U.S. Patent 7,254,576, Aug. 7, 2007.

[6] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, “Learning to rank using gradient descent,” in *Proceedings of the 22nd International Conference on Machine Learning (ICML ’05)*, pp. 89–96, 2005.

Appendix A. The query language.

```
// BNF of the query grammar:

//  Notation:
//
//      <x> ::= <y> <z>             <x> is composed of a <y> followed
//                                  by a <z>
//
//      <x> ::= <y> | <z>           <x> is composed of either a <y> or
//                                  a <z>
//
//      <x> ::= <y> [ <z> ]         <x> is composed of a <y> followed
//                                  by an optional <z>
//
//      <x> ::= <y> { <z> }         <x> is composed of a <y> followed
//                                  by zero or more <z> elements
//
//      <x> ::= 'c'                 <x> is a literal string, as specified
//                                  between the quotes
//

//
//      All keywords and search words are always case-folded.  URLs are
//      kept case-preserved.


//
//  The actual grammar:
//

//  A Query is the basic search statement and includes a means for
//  the front end (or trusted users) to inject optional information
//  overriding the ranking parameters or to provide additional
//  information about the user that may help in ranking.
//
//  For security reasons, the front end will transform any [ ... ]
//  option lists into spaces and any colons in the restricted #: or
//  category: operators into hyphens unless the query is from a
//  trusted (i.e., CorpNet) IP.
//
//  The back end will implement the functionality of the entire
//  Query grammar excepting word-breaking.
//
//  Names will be case-insensitive and may not be word-broken.
//  I anticipate a full C-like expression grammar.  But presently,
//  this is stubbed to simple assignments to the parameters
//  exposed by the Dynamic Ranker, FCS, CDG and the Aggregator.
```

```
//  <Query>             ::= { <Options> | <Preference> | <Constraint> }

//  <Options>           ::= '[' <Expression> { [ <Separator> ]
//                              <Expression> } ']'


//  <Expression>        ::= <ParameterName> [ <AssignOp> <ValueList> ]
//
//  <AssignOp>          ::= '=' | ':'
//
//  <ParameterName>     ::= <Leaf> { '.' <Leaf> }


//  Any number of preferences can be listed.  Each expressed as a
//  simple query which may/may not be satisfied by a page matching
//  the specified constraints.
//
//  <Preference>        ::= 'prefer:' <SimpleConstraint>


//  Constraints can either be on the pages that should be considered,
//  e.g., only pages in English or only on a specific site, or on the
//  match, e.g., only if there's a match in the title.
//
//  <Constraint>        ::= <BaseConstraint>
//                              { <OrOp> <BaseConstraint> }
//
//  <OrOp>              ::= 'OR' | '|' | '||'
//
//  <BaseConstraint>    ::= <SimpleConstraint>
//                              { [ <AndOp> ] <SimpleConstraint> }
//
//  <AndOp>             ::= 'AND' | '&' | '&&'
//
//  <SimpleConstraint>  ::= <Phrase> | <NestedConstraint> |
//                          <UnaryOp> <SimpleConstraint> |
//                          <Literal> | <PageConstraint> |
//                          <SearchWord>
//
//  <UnaryOp>           ::= '+' | '-' | 'NOT'
//
//  <Phrase>            ::= '"' { <SearchWord> } '"'
//
//  <NestedConstraint>  ::= '(' <Constraint> ')'


//  Literals are intended mostly for use by internal developers as
//  a way of creating searches for special words that can't be
//  typed normally.
```

```
//  <Literal>           ::= 'literal:' <LiteralList> |
//                          '#:' <LiteralList>
//
//  A literal is a concatenation of quoted strings, hex strings
//  and any ordinary tokens.
//
//  <LiteralList>       ::= <LiteralItem> |
//                          '(' <LiteralItem> { <LiteralItem> } ')'
//
//  <LiteralItem>       ::= '"' <literal characters> '"' |
//                          <HexString> |  <NonBrokenToken>


//  <PageConstraint>    ::= <Language> | <Location> |
//                          <Category> | <UrlConstraint>
//
//  Semantics are that the languages and categories listed
//  are OR'ed together.
//
//  <Language>          ::= 'language:' <SimpleList>
//
//  <Location>          ::= 'location:' <SimpleList>
//                          'loc:' <SimpleList>
//
//  <Category>          ::= 'category:' <SimpleList>
//
//  <SimpleList>        ::= <Leaf> | '(' { <Separator> <Leaf> } ')'
//
//  <UrlConstraint>     ::= <UrlOperator> <UrlList>
//
//  <UrlOperator>       ::= 'domain:' | 'site:' | 'inindex:' |
//                          'link:' | 'url:'
//
//  <UrlList>           ::= <Url> |
//                          '(' { [ <Separator> ] <UrlList> } ')'


//  <SearchWord>        ::= Any simple token other than
//                          one of the operators or a token
//                          composed only of separators.


//  <Separator>         ::= Any token or series of tokens composed
//                          purely of unintelligible punctuation.
```

Appendix B. Sample scoring parameters.

```
; This is a machine-generated file containing scoring parameters
; used by the Dynamic Ranker.

; There are four metastreams associated with each document,
; in expected order of significance in scoring:
;
;   Anchor text
;   Url ( site, domain, path, leaf )
;   Title
;   Body text ( normal, bold, headings 3-6, headings 1-2 )
;
;   Each metastream is scored independently, then the scores
;   are added to form an overall QualityOfMatch measure.


[Thresholds]

;  Boolean to determine if phrases should be considered as search
;  words or flattened out to individual search words.

PhrasesAsSearchWords=0

;  We estimate how often a word occurs in our index as
;
;                              Number of word locations
;                              in the entire index
;   Estimated frequency   =  -------------------------
;                              Number of occurrences of
;                              this particular word
;
;  A word is considered rare if its estimated frequency is
;  less than or equal to MinimumOneInNWordsToBeRare.

MinimumOneInNWordsToBeRare=100000

;  A word is considered common if its estimated frequency
;  is greater than or equal to MinimumOneInNWordsToBeCommon.

MaximumOneInNWordsToBeCommon=200

; Some pages have so much anchor text as to generate spurious
; matches.  If we get a match on a word in the anchor text,
; but that word represents less than some small percentage of
; all the anchor text pointing to that page, it's not really
```

```
; much of a match.

MaximumOneInNWordsToBeSpuriousAnchor=1000


; Given that we've found a document that contains a given
; search word, we estimate:
;
;                               Number of occurrences of
;       Expected number         that word in the index
;       of occurrences    =   ----------------------------
;       in this document        Number of documents in the
;                               index in which it appears
;
; A word is considered frequent within a metastream
; associated with a matching document if there are more
; than
;
;      expected * TimesExpectedToBeFrequent
;
; occurrences of that word in that metastream.

; Order of metastreams is ( Anchor, Url, Title, Body )

TimesExpectedToBeFrequent=( 2, 2, 2, 2 )
TimesExpectedToBeTooFrequent=( 5, 3, 3, 10 )
TimesRarestExpectedToBeFrequentSpan=( 2, 2, 2, 3 )


; MinPercentToBeMost specifies the percentage of all the
; words in the query that must be found to be considered
; "most".

; Order of metastreams is ( Anchor, Url, Title, Body )

MinPercentToBeMost=( 60, 60, 60, 60 )


; Each stream can be broken into segments, dividing
; it up by word location numbers.  Since actual
; length of each metastream isn't known, we choose
; a fixed number and define that the last segment
; includes everything past that.

; Order of metastreams is ( Anchor, Url, Title, Body )
```

```
SegmentSize=( 1000, 3, 4, 100 )
NumberOfSegments=( 10, 3, 3, 4 )
SegmentsInMost=( 9, 2, 2, 3 )


NearTop=( 25, 37, 37, 60 )

; MaxSkippedLocations tells how many unmatched locations
; can be allowed between the first and last words of
; a NearDouble or NearTriple.

MaxSkippedLocations=( 2, 2, 2, 2 )


; ShortSpanLimitMultiplier tells how to decide if
; a span, meaning a complete set of all search words
; found together, is close enough to consider a
; "short span."  A span is short if:
;
; length <= number of words * ShortSpanLimitMultiplier

; Order of metastreams is ( Anchor, Url, Title, Body )

ShortSpanLimitMultiplier=( 1, 2, 2, 3 )


; As we iterate from one index tier to the next, we use the
; reported highest static rank within that tier to tell us
; whether it's possible that even a 'perfect' page with that
; highest rank could make it into our current N best pages
; found so far.  If even a perfect page from that tier couldn't
; make it, we can skip searching the entire tier.
;
; But realistically, no page is ever going to get a perfect
; score and if we set the hurdle at that level, we'd search
; tiers that don't really have a chance of producing good
; matches.  MaximumPracticalTierQuality and
; MaximumPracticalPageQuality are estimates of the
; practical limit on highest scaled QualityOfMatch and are
; required to be in the range 0.0 to 1.0, inclusive.

MaximumPracticalTierQuality=0.05
MaximumPracticalPageQuality=0.30


[Mix]
```

```
; LinkRankCoefficient and QualityOfMatchCoefficient together
; specify the basic mix between static and dynamic rank
; when computing overall rank.

LinkRankCoefficient=34
QualityOfMatchCoefficient=133

; We expect that as the number of words in a query is increased,
; the user likely knows more specifically what he/she is looking
; for and that quality of match should become more important in
; the mix between static and dynamic rank.  Most words are of
; ordinary frequency, but the rare words are especially interesting
; and the most common ones add very little.

QualityIncrForEachWord=187
QualityIncrForEachRareWord=91
QualityIncrForEachCommonWord=-112


; Static rank, junk page score and numbers of occurrences of
; hits in anchor text are scaled using a bucketing arrangement.
;
; The buckets are monotonically increasing, and a raw score is
; converted to a scaled score by finding the highest bucket it's
; greater than.  The bucket number, linear interpolating between
; buckets to a float, is the scaled value.  To try out a new
; scaling relationship (e.g., a new combination of log and linear
; components), simply choose new buckets.

LinkRankQuanta=( 2 )
AnchorQuanta=( 2, 3, 5, 8, 14, 24, 40, 68, 116, 197, 333, 565, 959, 1625,
2756, 4674, 7926, 13440, 22791, 38648 )
JunkPageQuanta=( 55, 100, 130, 255 )


; Order of Url attributes is ( Service, SubDomain, Domain, TopDomain, Port,
Path, Query, Anywhere )

UrlAttributeRelativeValues=( 1, 1, 69, 1, 1, 30, 0, 1 )


; Order of Text attributes is ( Normal, Bold, H3-6, H1-2, Anywhere )

TextAttributeRelativeValues=( 0, 0, 2, 70, 294 )
```

```
[Words]

; The XxxWordsFound values for Anchor text are scaled
; according to the count, using the AnchorQuanta class
; and by attribute for url and body.
;
; In Url and Body text, XxxWordsFound values will be
; scaled by the attribute values.

; Order of metastreams is ( Anchor, Url, Title, Body )

AllWordsFound=( 418, 88, -5, -34 )
RarestWordFound=( 3, 0, 19, -100 )
MostWordsFound=( 125, 96, 49, -11 )
SomeWordsFound=( -69, 140, 47, -44 )


; The XxxWordsFrequent and XxxWordsTooFrequent are
; absolute numbers of points for being more frequent
; on this page in the given stream than expected.
;
; It's great to find the search words are really frequent in
; anchor text, and less so in body text, but not in a title
; or a URL.

; These points either are/are not earned.  They
; are not scaled.

; Order of metastreams is ( Anchor, Url, Title, Body )

AllWordsFrequent=( 23, 46, 0, 4 )
RarestWordFrequent=( 2, -77, 0, 13 )
MostWordsFrequent=( 15, -54, -23, 32 )
SomeWordsFrequent=( 7, 6, 0, 0 )

AllWordsTooFrequent=( 0, 8, -59, 1 )
RarestWordTooFrequent=( 60, 69, 42, 10 )
MostWordsTooFrequent=( -33, 54, -78, -12 )
SomeWordsTooFrequent=( -3, -26, -50, -2 )


; The XxxWordsInXxxSegments metrics test, in effect
; by sampling, whether the 'whole' of this metastream
; seems to be a good match.
```

```
; Order of metastreams is ( Anchor, Url, Title, Body )

AllWordsInAllSegments=( 88, -2, -58, -3 )
RarestWordInAllSegments=( 13, 32, -5, 0 )
AllWordsInMostSegments=( 186, 64, 78, 53 )
RarestWordInMostSegments=( 0, -14, -2, 0 )
MostWordsInAllSegments=( 68, 0, 4, 0 )
MostWordsInMostSegments=( 76, -101, -35, -9 )
SomeWordsInAllSegments=( -246, 27, 56, -1 )

; Order of metastreams is ( Anchor, Url, Title, Body )

AllWordsNearTop=( -22, -37, -6, 0 )
RarestWordNearTop=( -7, 0, 0, 7 )
MostWordsNearTop=( 138, 0, 27, 26 )
SomeWordsNearTop=( 33, 34, 30, 6 )


[Spans]

; ExactPhrase, ShortSpan and InOrder points for
; anchor text are scaled by the quanta.  They are
; not scaled in the other metastreams.

; Order of metastreams is ( Anchor, Url, Title, Body )

ExactPhrase=( 218, 8, 3, 21 )
ShortSpan=( 84, 30, 23, 51 )
InOrder=( 111, -1, 16, 0 )
ShortInOrder=( -4, -6, 4, 0 )

; Double and Triple are scored as a two- or three-
; word match in any order with no intervening space
; where one of the words is the rarest.

Double=( -15, 0, 0, 59 )
Triple=( 84, -1, 8, 30 )

; NearDouble and NearTriple can have at most
; MaxSkippedLocations that weren't matched somewhere
; between the begin and end of the double or triple.
; Obviously, every Double is also a NearDouble.

NearDouble=( -116, 0, -7, 17 )
NearTriple=( 197, 0, 26, 11 )
```

```
; InOrderDouble and InOrderTriple require that
; the words appear in the same order as in the
; query.  Every InOrderDouble is also a Double
; and also a NearDouble.

InOrderDouble=( -29, 0, 2, 164 )
InOrderTriple=( 1, 0, 0, 29 )


; NearInOrderDouble and NearInOrderTriple require
; that the words appear in the same order as in the
; query and with at most MaxSkippedLocations that
; weren't matched between the begin and end of the
; double or triple.

NearInOrderDouble=( 85, 0, 0, 65 )
NearInOrderTriple=( 124, 0, -4, 11 )


; ManyXxxXxx measures test only for the condition
; being met/not met; they are not scaled.

; Order of metastreams is ( Anchor, Url, Title, Body )

ManyExactPhrases=( 8, -16, -41, -9 )
ManyShortSpans=( 0, -40, -1, -29 )
ManyInOrder=( 2, -5, -1, 0 )
ManyShortInOrder=( -3, 14, 0, 0 )

ManyDoubles=( -1, 2, 0, -9 )
ManyTriples=( 0, 0, 0, -7 )
ManyNearDoubles=( 0, 0, 0, -4 )
ManyNearTriples=( 13, -2, 1, 0 )
ManyInOrderDoubles=( -1, 0, -1, -3 )
ManyInOrderTriples=( -1, -60, -21, -10 )
ManyNearInOrderDoubles=( 0, -2, 0, 0 )
ManyNearInOrderTriples=( 10, 3, 21, -1 )

ExactPhraseNearTop=( -7, 0, 3, 0 )
ShortSpanNearTop=( -31, 0, 33, 0 )
InOrderNearTop=( -114, 0, 13, -1 )
ShortInOrderNearTop=( -7, 0, 17, -3 )
DoubleNearTop=( -1, 0, 0, -2 )
TripleNearTop=( 9, -1, 9, 0 )
NearDoubleNearTop=( -40, 0, -5, -10 )
NearTripleNearTop=( 6, 0, 28, 0 )
```

```
InOrderDoubleNearTop=( -1, 0, 1, 0 )
InOrderTripleNearTop=( 0, 0, 0, 0 )
NearInOrderDoubleNearTop=( 0, 0, 0, 0 )
NearInOrderTripleNearTop=( 4, 0, -4, 0 )

FirstSpanIsExactPhrase=( 59, 0, -9, -7 )
FirstSpanIsShort=( 38, 0, 32, 0 )
FirstSpanInOrder=( 124, 2, -3, 0 )
FirstSpanShortInOrder=( 0, 0, 0, 0 )
FirstSpanNearTop=( 109, 0, 21, 18 )



[Streams]

; TF/IDF is measured per metastream in the range 0 .. 1,
; then multiplied by the value for that metastream.

TfIdf=( -119, -122, -122, -67 )


; CompleteMatch means that you have all the words in that
; metastream (or portion we'll consider) even if the query
; contained additional words and even if the query and the
; metastream are just one word long.
;
; An example might be a query for
;
;   discount catnip mice
;
; that matches http://discountmice.com.
;
; For anchor text, we compare against an anchor phrase,
; scaling by frequency using the AnchorQuanta.  For URL,
; we consider only domain.  For title and body, we consider
; the entire metastream.

CompleteMatch=( 923, 29, 0, 239 )


; PerfectMatch means that you have an ExactPhrase and that
; the number of words in the query matches the number of
; words in the metastream (or portion we'll consider.)
; Having a PerfectMatch implies CompleteMatch.

PerfectMatch=( 318, 11, 0, -235 )
```

```
; Each document earns additional points (or penalties)
; depending on the length of the url, path and title.
;
; XxxLengthAdjustments are indexed by the length of that
; component.  If the length is greater than the
; number of elements in the array, the last value is
; used.


; Domain Length is known to 3 bits:  0 .. 7+  (>7 encoded as 0; 0 encoded as
1)
;
;                         >7  0/1   2   3   4   5   6   7
DomainLengthAdjustment=( 54, 87, 40, 13, 23, 0, 0, 4 )


; Path Length is known to 4 bits:  0 .. 14+  (>14 encoded as 0; 0 encoded as
1, 1 encoded as 2, etc.)
;
;                       >14   0   1   2   3  4  5  6   7   8   9  10  11  12
13   14
PathLengthAdjustment=( 0, 135, 51, 2, 0, 1, 0, 0, 6, 6, 0, 3, 0, 0, 0, 0 )


; Title Length is known to 4 bits:  0 .. 15+  (>15 encoded as 0; 0 encoded as
1)
;
;                        >15  0/1  2  3  4  5  6  7   8   9  10   11   12   13
14   15
TitleLengthAdjustment=( 0, 47, 0, 0, 1, 2, 1, 0, 2, 0, 0, 0, 14, 3, 2, 1 )

; TopLevelDomain, e.g., .com vs. .co.uk, is stored for each page as a 12-bit
; enum uniquely identifying the roughly 1300 TLDs we currently know about.
; The table used to generate that enum is defined in
;
;   TopLevelDomainList.h
;
; The DynamicRanker uses other fields of that table to make a basic
classification
; for the TLD of each page being scored into one of 9 buckets:
;
;     Gov, Edu, Com, Net, Org, Other, Foreign, Junk, IPAddress
;
; The IPAddress bucket is for urls containing an IP address, not an actual
```

```
; domain name.  Currently, the only Junk TLD we've identified is .biz.
;
; Our intent is that if the page is domestic, that we will use the
classification
; it has locally to score the page.  All foreign pages are bucketed
separately,
; thus avoiding mixing, e.g., foreign Gov pages in with domestic.  The
; exceptions are junk and IP TLDs:  If a TLD is junk or IP in one market,
; it's the same in all markets.
;
; We consider a page to be domestic if a location preference is stated in
; the query and we matched on that preference.  If a preference was stated
; and we did not match, we consider the page foreign.  If no preference was
; stated, we consider that to be the same as a preference for the generic
; variants.
;
; TLD Bucketing order is ( Gov, Edu, Com, Net, Org, Other, Foreign, Junk,
IPAddress )
TopLevelDomain=( 1, 1, 1, 1, 1, 1, 1, 1, 1 )



; HyphensInDomain is known to 2 bits:  0 .. 3+.  For efficiency in tuning,
; these values are presented to the tuner as "or more" buckets.
;
;                  0  1+ 2+ 3+
HyphensInDomain=( 1, 1, 1, 1 )



; The ODP is the Open Directory Project at dmoz.org, a categorized listing
; of pages on the web.  A page being listed more once in the ODP is expected
; to be associated with higher quality.  EntriesInODP is known to 3 bits,
; 0 .. 7+.  For efficiency in tuning, these are presented to the tuner
; as "or more" buckets.
;
;               0  1+ 2+ 3+ 4+ 5+ 6+ 7+
EntriesInODP=( 1, 1, 1, 1, 1, 1, 1, 1 )



; The value of a match in the domain, path or title depends
; on the length of that component.  A match in a title that's
; over 15 words long is a lot less interesting than one that's
; 3 words long.
;
```

```
; XxxMatchRelativeValuesByLength describe the relative value
; of a match in that component based on the length.


; Domain Length is known to 3 bits:  0 .. 7+  (>7 encoded as 0; 0 encoded as
1)
;
;                                    >7   0/1   2   3   4   5   6   7
DomainMatchRelativeValuesByLength=( 42, 73, 49, 24, 7, 12, 21, 34 )


; Path Length is known to 4 bits:  0 .. 14+  (>14 encoded as 0; 0 encoded as
1, 1 encoded as 2, etc.)
;
;                                  >14   0   1   2   3   4   5   6   7   8   9
10  11  12  13  14
PathMatchRelativeValuesByLength=( 84, 453, 469, 497, 270, 154, 90, 126, 119,
117, 92, 121, 179, 253, 247, 247 )


; Title Length is known to 4 bits:  0 .. 15+  (>7 encoded as 0; 0 encoded as
1)
;
;                                   >15   0/1   2   3   4   5   6   7   8   9
10  11  12  13  14  15
TitleMatchRelativeValuesByLength=( 0, 25, 31, 25, 22, 17, 15, 11, 6, 5, 2, 0,
0, 0, 0, 0 )


; If there are no matches of a word except in anchor text and if
; those matches appear to be spurious (i.e., the frequency in anchor
; text is no more than MaximumOneInNWordsToBeSpuriousAnchor), adjust
; the page score.

AllWordsSpuriousAnchor=-138
RarestWordSpuriousAnchor=-155
MostWordsSpuriousAnchor=-194
SomeWordsSpuriousAnchor=-214


; QualityOfPage is a 8-bit aggregate measure of the overall quality
; of the page, independent of the query.  That metric is normalized
; to a range of 0 .. 1.0 and multiplied by the value specified.

QualityOfPage=0
```

```
; JunkPage is a 8-bit aggregate measure of whether the page is
; junk, independent of the query.  That metric is normalized to
; a range of 0 .. 1.0 and multiplied by the value specified.

JunkPage=239


; YoungerThanXxx awards points to the page if it is newer than
; that number of days/week/months/years.

; (This feature is not yet implemented; this is a placeholder.)

YoungerThan1Day=0
YoungerThan4Days=0
YoungerThan1Week=0
YoungerThan1Month=0
YoungerThan1Year=0
YoungerThan5Years=0


[Preferences]

WordPreference=462
LiteralPreference=467
LanguagePreference=358
LocationPreference=416
CategoryPreference=464
UrlPreference=500
PhrasePreference=1
ComplexPreference=756
```