



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigg.

Jun 21, 2016

## Query Understanding: A Manifesto

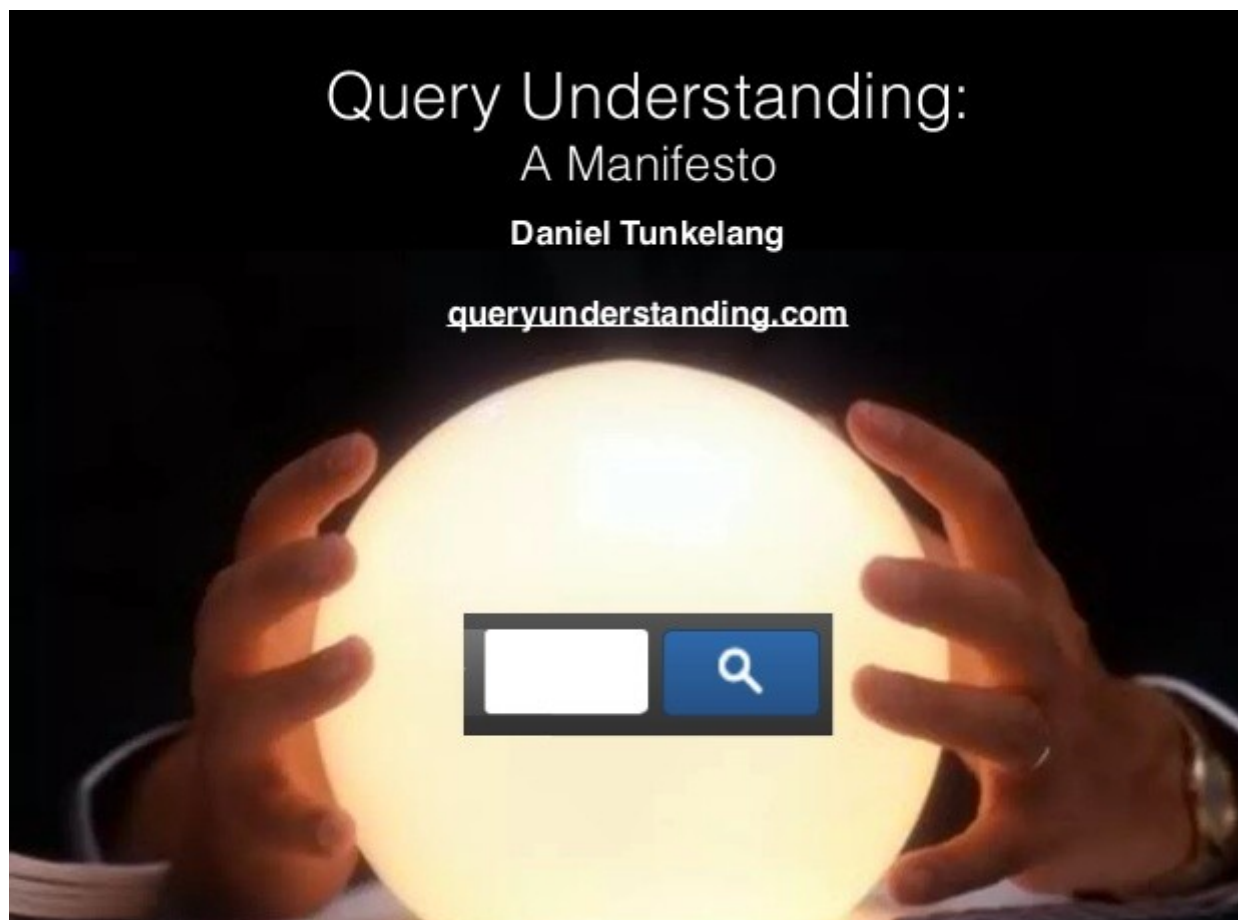
Query understanding has been my obsession for years. But my recent work at Etsy and Pinterest has sharpened my thinking on the topic.

So I put together a presentation to share that thinking with anyone interested in query understanding — which hopefully includes anyone working on search.

If you're impatient, here's the tl;dr: **Query understanding is about focusing less on the results and more on the query.**

Query understanding is about figuring out what the searcher wants, rather than scoring and ranking results. Once you've established this mindset, your approach to search changes: you focus on query performance rather than ranking. You also pay far more attention to query suggestions, particularly those generated through autocomplete.

You can find the full presentation on SlideShare or view the embedded version below. I also encourage you to read the series of posts I'm writing about query understanding, starting with this introduction.





Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigg.

Oct 28, 2016 · 3 min read

## Query Understanding: An Introduction

Search engines are so core to our digital experience that we take them for granted. Most of us cannot remember the web without Google to search its contents. Or online shopping without Amazon's search engine to find what we want to buy. Search engines are how we navigate the digital world.

**But what is a search engine?** To most people — including most software engineers — a search engine is a system that accepts a text query as input and returns an list of results, ranked by their purported relevance to the query. Most work on search engines focuses on improving that ranking, with the help of increasingly sophisticated machine learning systems.

**Query understanding focuses on the beginning of the search process: the query.** Query understanding is about what happens before the search engine scores and ranks results — namely, the searcher's process of expressing an intent as a query, and the search engine's process of determining that intent. In other words, query understanding is about the communication channel between the searcher and the search engine.

**Query understanding treats the query as first-class.** It makes queries the focal point of the search process. Much of query understanding takes place before retrieving a single result, although it's possible to perform post-retrieval analysis for validation. But query understanding isn't about determining the relevance of each result is to the query. Rather, it establishes the interpretation of the query, against which results are judged.

**Query understanding plays a key role in the search user interface.**

Because query understanding is the first step in the search process, it is the part of the process the user interacts with most intensely. Query understanding is core to basic search interface features like autocomplete, spellcheck, and query refinement. More broadly, query understanding pervades every interaction the searcher has with the search engine.

**In particular, query understanding is at the heart of search suggestions.**

The most important kinds of suggestions are autocomplete suggestions, especially as autocomplete is becoming the primary surface for the search experience. More broadly, query formulation is itself a search problem — a

search through the space of possible queries rather than through the space of results. And in a mobile-first world, it's especially important to support query formulation and refinement, saving searchers from the delay and frustration of submitting ineffective search queries.

**Finally, focusing on query understanding creates a different mindset for search engine developers.** This focus shifts the emphasis from scoring and ranking to a goal of determining the searcher's intent. Instead of striving to create the optimal ranking algorithm, search engine developers aspire to create the optimal query interpretation mechanism. The driving success metric is query performance — an end-to-end measure of the quality of the communication channel between the searcher and the search engine.

**Over the next months, this publication will take us through the journey from characters to words to phrases, and ultimately to meaning.**

We'll start at the bottom of the query understanding stack with character-level techniques like normalization and tokenization. We'll look at stemming, lemmatization, and dictionary-based canonicalization. We'll continue on to higher-order operations like query relaxation, query segmentation, and entity recognition. Along the way, we'll explore semantic resources like synonyms, hypernyms, taxonomies, ontologies, and knowledge graphs.

We'll then look at ways that we can rewrite queries through automatic phrasing, field restriction, and query expansion. We'll pay particular attention to autocomplete, from indexing prefix completions to providing structured autocomplete suggestions to weighing query probabilities against query performance.

Moving outside the search box, we'll explore context — particularly session, geographical, and temporal contexts. We'll then look at personalization, both explicit and implicit, and we'll explore the differences between personalization and relevance.

We'll then see how query understanding helps us establish search as a conversation between the searcher and the search engine. We'll look at interaction patterns like clarification dialogs, faceted search, and relevance feedback. We'll also explore results presentation, particularly snippets and clustering. Finally, we'll look at natural-language search interfaces: question answering, voice interfaces, and chatbots.

**As you join me along this journey, I hope you'll come to appreciate the key role that query understanding plays in the search process.** If you're a software engineer, data scientist, or product manager working on a search

engine, I encourage you to use what you learn to improve your searchers' experience.

**Next: Language Identification**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twiggie.

Nov 5, 2016 · 3 min read

# Language Identification

Queries are the primary way that searchers communicate with search engines. But it's difficult to get started with query understanding until the search engine identifies the language in which the query is written.

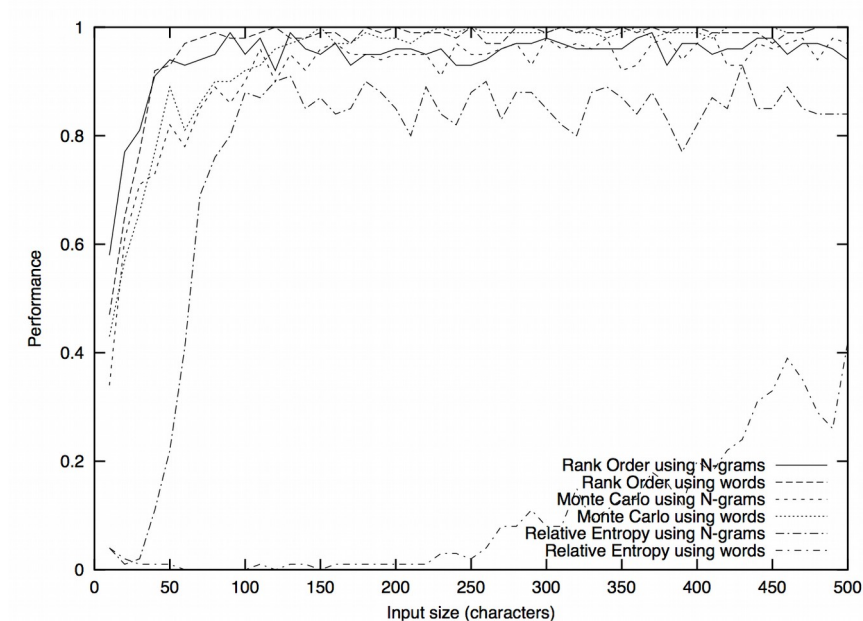
## History

Automatic language identification has seen several decades of work, going back to research in the 1970s to automatically identify the language of written or spoken text based on the frequencies of letter combinations or sound segments. Work on statistical analysis to determine language and authorship goes back even further to Udney Yule, George Zipf, and Jean-Baptiste Estoup.

Given the importance that language plays in society, it's not surprising that this topic has been the object of sustained study, particularly among linguists and statisticians. Most of that study predates modern search engines.

## Search Queries are Short

In general, statistical approaches to language identification take advantage of distributional characteristics of the text they analyze. As a result, the accuracy of these approaches is a function of the amount of text they analyze. As Arjen Poutsma showed in his 2001 paper on "Applying Monte Carlo Techniques to Language Identification", a wide variety of language identification algorithms perform poorly for inputs of fewer than 50 characters.



This result is unsurprising: it's difficult to obtain a meaningful distribution from a short text sample. Unfortunately, search queries tend to be short.

### Using Signals Beyond the Query

In 2009, Hakan Ceylan and Yookyung Kim wrote a paper on “Language Identification of Search Engine Queries”. In it, they described how they used clickthrough logs from Yahoo’s search engine to identify the language of queries based on the language of results searchers clicked for those queries.

Their approach took advantage of the relative ease of identifying the language of longer documents: in their data, the average search result contained almost 500 words, far more than the average search query. Even a single result could provide a robust signal, and most search queries were associated with clicks on multiple results.

They also noted that the language of the query terms might not be the same as the desired language of the results. For example, someone who searched for *homo sapiens* was probably not looking for documents written in Latin. To address these kinds of queries, they introduced a non-linguistic signal: the country from which the searcher was accessing the search engine.

But geography, while useful, cannot be trusted as the only signal. For example, Ceylan and Kim found that a quarter of the English queries in their data came from countries whose primary language was not English.

Hence, Ceylan and Kim approached language identification as a machine learning problem, using a combination of query features, document features,

and the searcher's country. Their best classifier achieved an average of 94.5% accuracy across 10 European languages.

### **When in Doubt, Ask!**

No language identification approach is perfect. Even an accuracy in the mid 90s means that there will be frequent errors. Besides, it's never a good idea to rely entirely on algorithmic approaches when it's easy to ask the searcher for clarification.

If your search engine serves a multilingual population, make it easy for the searcher to specify his or her language. Don't hide this functionality behind an undiscoverable advanced search page. And if your algorithmic language identification exceeds some uncertainty threshold, offer the searcher a clarification dialogue to specify the desired language.

### **Further Reading**

If you'd like to learn more about this topic, I recommend the 2014 "Survey of Language Identification Techniques and Applications" by Archana Garg, Vishal Gupta, and Manish Jindal. It discusses current work in this area, as well as a summary of the accuracy of various methods. It's a well-written overview and serves a useful annotated bibliography.

### **Previous: Introduction**

### **Next: Character Filtering**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigggle.

Nov 19, 2016 · 4 min read

# Character Filtering

Search queries are made up of characters. It's easy to take characters for granted; indeed, people who have never implemented a search engine might not think of working with characters as particularly challenging. But there are subtleties at the character level that we must tackle before working our way up the rest of the query understanding stack.

This document describes a series of character filters, which are steps to transform text at the character level. It's important that you execute these filters in the order presented, and that you apply exactly the same sequence of filters to indexed content as to search queries.

## Unicode Normalization

Modern software systems support Unicode, a living, international standard for consistently encoding text across the world's written languages. Because there can be many ways to represent the same character, Unicode introduces normalization as a way to transform characters into standard canonical forms, making it possible to recognize and match equivalent representations.

Unicode supports two kinds of normalization:

- Normalization Form Canonical Decomposition (NFD). Characters are decomposed by canonical equivalence, and multiple combining characters are arranged in a specific order.
- Normalization Form Canonical Composition (NFC). Characters are decomposed and then recomposed by canonical equivalence.

Unicode also supports variants of the above, NFKD and NFKC, in which the decomposition step uses compatibility forms that more aggressively standardize characters.

For search engines, it's simpler and more efficient to use a normalization based on decomposition than one based on composition. That's because the first thing you'll want to do to a normalized string is remove accents — which is straightforward after decomposition. You can use either NFD or NFKD — given the subsequent transformations, it's unlikely you'll notice a difference between the two.



Fortunately, Unicode normalization is natively supported by most of the tools you're likely to use for developing search applications. You'll find support for Unicode normalization in Java and Python, as well as in open-source search engines Apache Lucene and Elastic.

## Removing Accents

Unicode normalization transforms strings into a standard character encoding, but it leaves accents (more technically, diacritics) in place. For example, *café* with an accent is different than *cafe* without an accent.

Accents can change the meanings of words. In Spanish, for example, *papa* means potato, while *papá* means dad.

But not all keyboards support accents. And even when keyboards do support accents, they require searchers to perform additional work to enter accented characters. Searchers are people, and people are lazy. So you shouldn't assume searchers — or even content creators — will use accents consistently.

Hence, it's a good idea to remove accents as part of indexing content and processing search queries. This step takes place after Unicode normalization, by removing all the diacritic characters.

Again, there are standard tools to remove accents. In Java, there's `StringUtils.stripAccents`; in Python, there's the `Unidecode` module. In Apache Lucene, use the `ASCIIFoldingFilter`; in Elastic, use `asciifolding`. These tools also simplify ligatures, e.g., they transform *æ* into *ae*.

Removing accents doesn't mean you should forget about them entirely. It's important to preserve original strings, accents and all, for displaying both the search results and the query. That's not only respectful to the original text; it also mitigates the risk of confusing searchers when removing accents changes a word's meaning.

## Ignoring Capitalization

Many alphabets, notably the Roman alphabet, are bicameral — that is, they support two cases. In formal writing, such as this document, we expect to see proper use of uppercase and lowercase letters.

But, just as it's unreasonable to expect searchers to make proper use of accents in their search queries, it's unreasonable to expect proper capitalization. Hence, we recommend converting all strings to lowercase — a process also known as case folding.

In Java, you can convert a string to lowercase using `String.toLowerCase`. In Python, you can use `lower`. Apache Lucene has a `LowerCaseFilter`; Elastic has a lowercase token filter.

Case folding requires you to specify language of the text, called the locale (see previous post on language identification). If you don't know the language of the text, then you should use a default locale (e.g., United States English) for consistency.

### **Every Filter is a Trade-Off**

Each of the filters we've discussed represents a trade-off of precision for recall. It's possible that filtering will cause two words with different meanings to be canonicalized to the same representation.

If you believe this is a significant risk, then you should evaluate the precision of your filters through human judgments. Take a sample of queries, and evaluate how often the filtered queries preserve the original intent. If you see systematic errors, adjust your filters accordingly.

But be careful not to be so finicky about precision that you give up too much recall. The character filters described here are conservative, and you'll probably want all of them in order to achieve effective string matching.

**Previous: Language Identification**

**Next: Tokenization**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigggle.

Dec 17, 2016 · 3 min read

# Tokenization

Now that we can handle characters, let's move on to words.

A critical task for query and document understanding is breaking up the text into a sequence of words. We call these words tokens — but as we'll see in a moment, tokens include strings that aren't necessarily words that appear in a conventional dictionary. Tokenization converts a string of characters into a sequence of tokens.

Tokenization is deceptively subtle. You might think it's as simple as splitting text on spaces, something you could accomplish using the `split` method in Java or Python. At the very least, you'll at want to split all white space characters — after all, some people prefer tabs to spaces.

## Punctuation

It may seem innocuous to treat punctuation marks like white space — for example, we expect to the same search results for a comma-separated string *apple, oranges, pears* as for the corresponding space-separated string *apples oranges pears*. We're unlikely to encounter end-of-sentence punctuation in search queries unless users ask natural-language questions ending in question marks, but even then we can probably get away with treating those marks as white space.

## Hyphens and Apostrophes

Hyphens are tricky. Words with true hyphens, like *twenty-five*, should be considered single tokens. In contrast, words connected by grammatically required hyphens, such as in the compound modifier *California-based*, should be considered multiple tokens.

Apostrophes are even trickier. We encounter them in contractions (e.g., *we're here*) and possessives (e.g., *women's rights*). For contractions, we'd ideally tokenize the query as if it were uncontracted (*we are here*) both in queries and in documents. For possessives, logic would have separate out the *'s* as a separate token. But many people are careless about using apostrophes — just think of how often people confuse *its* and *it's*. Our job in query understanding is not to be grammarians, but rather to optimize for capturing the searcher's intent.

## Multiple Tokenizations

Hence, a reasonable strategy for apostrophes is to compute multiple tokenizations, e.g. tokenizing *women's* as both *womens* and *women's*. We improve recall by allowing for multiple tokenization, but we also maintain precision by avoiding tokenizations like *women s* that would retrieve documents containing the letter *s* as a token. This strategy doesn't always work, but it is a decent compromise. We can also use it for hyphens.

## Beyond English

All of the above discussion has focused on tokenization in English. Other languages are just as important, and some raise challenges we don't usually encounter in English.

Some languages, most notably German, make extensive use of compound words. In these languages, it's important to decompound words into their components as part of tokenization, e.g., tokenizing the German *Fruchtsalat* as *Frucht* (fruit) *Salat* (salad). Again we can use a multiple tokenization strategy (*Fruchtsalat*, *Frucht Salat*) to optimize for recall.

The situation is even harder in Chinese, Japanese, and Korean text, where there are generally no spaces between words. In these cases, it's necessary to employ a word segmentation algorithm specific to that language.

Multilingual search engines generally include tools for decompounding and word segmentation. But you should be aware that not all languages are as easy to tokenize as English. Be prepared to handle the inevitable bugs.

## Tokens that Aren't Words

Numbers and other symbols require special care.

Standalone numbers like *42* can be treated as tokens. But numbers are often part of larger tokens, e.g., *EC2*, *2x4*. In these cases, we want to be resilient to spacing variations, e.g., we shouldn't distinguish between *EC2* and *EC 2*. And some number sequences, like telephone numbers and part numbers, require specialized algorithms to recognize and tokenize them.

Other symbols can also create challenges. For example, we sometimes treat the *@* symbol as a separator, but we don't want to do so when it occurs in the middle of an email address. Similarly, we want to tokenize *C++* as a single token. You may find that you need to customize your tokenizer to handle these cases.

## **No Tokenization Approach is Perfect**

As with every aspect of query understanding, tokenization represents a set of tradeoffs. A highly literal tokenization of the query is likely to be good for precision, but bad for recall; while a more aggressive approach using multiple tokenizations will improve recall at the expense of precision. Handling special cases like strings that combine letters and numbers introduces complexity, but that complexity may be important for your use cases. No approach is perfect, so you'll need to decide how to manage the trade-offs.

**Previous: Character Filtering**

**Next: Spelling Correction**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigggle.

Jan 16, 2017 · 7 min read

# Spelling Correction

Spelling correction is a must-have for any modern search engine. Estimates of the fraction of misspelled search queries vary, but a variety of studies place it between 10% and 15%. For today's searchers, a search engine without robust spelling correction simply doesn't work.

That doesn't mean, however, that you should build a spelling correction system from scratch. Off-the-shelf spelling correction systems, such as Aspell or Hunspell, are highly customizable and should suit most people's needs.

It's still important, however, to understand how spelling correction works. A great starting point is Peter Norvig's classic post on "How to Write a Spelling Corrector", which walks through fundamental concepts like edit distance, language models, and error models.

Let's look at the key aspects of a spelling-correction system.

## Overview

Offline, before any queries take place:

- **Indexing Tokens.** Building the index used at query time for candidate generation.
- **Building a language model.** Computing the model to estimate the a priori probability of an intended query.
- **Building an error model.** Computing the model to estimate the probability of a particular misspelling, given an intended query.

At query time:

- **Candidate generation.** Identifying the spelling correction candidates for the query.
- **Scoring.** Computing the score or probability for each candidate.
- **Presenting suggestions.** Determining whether and how to present a spelling correction.

Let's drill down into each of these aspects.

## **Indexing Tokens**

Indexing for spelling correction is a bit different than for document retrieval. The fundamental data structure for document retrieval is an inverted index (also known as a posting list) that maps tokens to documents. In contrast, indexing for spelling correction typically maps substrings of tokens (character n-grams) to tokens.

Most misspelled tokens differ from the intended tokens by at most a few characters (i.e, a small edit distance). An index for approximate string matching enables discovery of these near-misses through retrieval of strings — in our case, tokens — by their substrings. We generate this index by identifying the unique tokens in the corpus, iterating through them, and inserting the appropriate substring-to-string mappings into a substring index, such as an n-gram index or a suffix tree.

Although this description assumes a batch, offline indexing process, it is straightforward to update the index incrementally as we insert, remove, or update documents.

This index grows with the size of the corpus vocabulary, so it can become quite large. Moreover, as vocabulary grows, the index becomes more dense: in particular, short substrings are mapped to large numbers of tokens. Hence, a spelling correction system must make tradeoffs among storage, efficiency, and quality in indexing and candidate generation.

It is also possible to index tokens based on how they sound, such as canonicalizing them into Metaphone codes. This approach is most useful for words with unintuitive spellings, such as proper names or words adopted from other languages.

## **Building Models**

The goal of spelling correction is to find the correction, out of all possible candidate corrections (including the original query), that has the highest probability of being correct. In order to do that, we need two models: a language model that tells us a priori probability of a query, and an error model that tells us the probability of a query string given the intended query. With these two models and Bayes' theorem, we can score candidate spelling correction candidates and rank them based on their probability.

## **Building a Language Model**

The language model estimates the probability of an intended query — that is, the probability, given no further information, that a searcher would set out to type a particular query.

Let's consider the simplifying assumption that all of our queries are single tokens. In that case, we can normalize the historical frequencies of unique queries to establish a probability distribution. Since we have to allow for the possibility of seeing a query for the first time, we need smoothing (e.g., Good-Turing) to avoid assigning it a probability of zero. We can also combine token frequency in the query logs with token frequency in the corpus to determine our prior probabilities, but it's important not to let the corpus frequencies drown out the query logs that demonstrate actual searcher intent.

This approach breaks down when we try to model the probability of multiple-token queries. We run into two problems. The first is scale: the number of token sequences for which we need to compute and store probabilities grows exponentially in the query length. The second is sparsity: as queries get longer, we are less able to accurately estimate their low probabilities from historical data. Eventually we find that most long token sequences have never been seen before.

The solution to both problems is to rely on the frequencies of n-gram frequencies for small values of n (e.g. unigrams, bigrams, and trigrams). For larger values of n, we can use backoff or interpolation. For more details, read this book chapter on n-grams by Daniel Jurafsky and James Martin.

## **Building an Error Model**

The error model estimates the probability of a particular misspelling, given an intended query. You may be wondering why we're computing the probability in this direction, when our ultimate goal is the reverse — namely, to score candidate queries given a possible misspelling. But as we'll see, Bayes' theorem allows us to combine the language model and the error model to achieve this goal.

Spelling mistakes represent a set of one or more errors. These are the most common types of spelling errors, also called edits:

- **Insertion.** Adding an extra letter, e.g., *truely*. An important special case is a repeated letter, e.g., *pavillion*.
- **Deletion.** Missing a letter, e.g., *chauffer*. An important special case is missing a repeated letter, e.g., *begining*.



- **Substitution.** Substituting one letter for another, e.g., *appearence*. The most common substitutions are incorrect vowels.
- **Transposition.** Swapping consecutive letters, e.g. *acheive*.

We generally model spelling mistakes using a noisy channel model that estimates the probability of a sequence of errors, given a particular query. We can tune such a model heuristically, or we can train a machine-learned model from a collection of example spelling mistakes.

Note that the error model depends on the characteristics of searchers, and particularly on how they access the search engine. In particular, people make different (and more frequent) mistakes on smaller mobile keyboards than on larger laptop keyboards.

### Candidate Generation

Candidate generation is analogous to document retrieval: it's how we obtain a tractable set of spelling correction candidates that we then score and rank.

Just as document retrieval leverages the inverted index, candidate generation leverages the spelling correction index to obtain a set of candidates that is hopefully includes the best correction, if there is one.

To get an idea of how candidate generation works, consider how you might retrieve all tokens within an edit distance of 1 of the misspelled query *retrieval*. A token within edit distance must end with *etreival*, or start with *r* and end with *treival*, or start with *re* and end with *reival*, or start with *ret* and end with *eival*, etc. Using a substring index, you could retrieve all of these candidates with an OR of ANDs.

That's not the most efficient or general algorithm for candidate generation, but hopefully it gives you an idea of how such algorithms work.

The cost of retrieval depends on the aggressivity of correction, which is roughly the maximum edit distance — that is, the number of edits that can be compounded in a single mistake. The set of candidates grows exponentially with edit distance. The probability of a candidate decreases with its edit distance, so the cost of more aggressive candidate generation yields diminishing returns.

### Scoring

Hopefully the right spelling correction is among the generated candidates. But how do we pick the best candidate? And how do we determine whether

the query was misspelled in the first place?

For each candidate, we have its prior probability from the language model. We can use the error model to compute the probability of the query, given the candidate. So now we apply Bayes' theorem to obtain the conditional probability of the candidate, given the query:

$$\text{Prob}(\text{candidate} \mid \text{query}) \propto \text{Prob}(\text{query} \mid \text{candidate}) * \text{Prob}(\text{candidate})$$

We use the proportionality symbol ( $\propto$ ) here instead of equality because we've left out the denominator, which is the a priori probability of the query. This denominator doesn't change the relative scores of candidates.

In general, we go with the top-scoring candidate, which may be the query itself. We use the probability to establish our confidence in the candidate.

### **Presenting Suggestions**

Deciding how to present suggested spelling corrections may not be an algorithmic problem, but it's a critical part of the search experience.

The key factors are whether the top-ranked candidate has a high probability, and whether that candidate is the original query:

- If the top candidate is the original query and has a high probability, then do nothing — your system believes the query is already spelled correctly.
- If the top candidate is not the original query and has a high probability, then automatically rewrite the query and notify the searcher with a prominent message above the search results. Include a link that allows the searcher to see results for the original query.
- If the top candidate is not the original query but does not have a high probability, then show results for the original query but propose the alternative as a “did you mean”. Consider the same approach when the top candidate is the original query but the next candidate has almost as high a probability.

Finally, you shouldn't rewrite a query into one that returns no results, regardless of the probability. Converse, there isn't as much risk in correcting a query that would otherwise return no results.

### **Conclusion**

Hopefully you now understand how spelling correction systems work — congratulations!

Again, that doesn't mean you should try to write your own. As you hopefully appreciate by now, spelling correction systems are complex beasts, dependent on robust language and error models, as well as trade-offs to optimize their effectiveness and efficiency. Knowing the internals will help you use and customize best-of-breed, off-the-shelf tools.

Finally, remember that the quality of spelling correction is one of the most important attributes of the search experience. Invest accordingly, and make your searchers happy.

**Previous: Tokenization**

**Next: Stemming and Lemmatization**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twiggie.

Feb 6, 2017 · 4 min read

## Stemming and Lemmatization

Different forms of a word often communicate essentially the same meaning. For example, there's probably no difference in intent between a search for *shoe* and a search for *shoes*.

These syntactic differences between word forms are called inflections, and they create challenges for query understanding. In the example of *shoe* and *shoes*, we probably want to treat the two forms identically. But we wouldn't want to do the same for the words *logistic* and *logistics*, which mean very different things despite their apparent similarity. Nor would we want to equate the words *universe* and *university*, even though both words derive from the same Latin root.

Stemming and lemmatization are two approaches to handle inflections in search queries. We will discuss each of them and then consider a more general approach, which I call canonicalization.

### Stemming

When we stem a mushroom, we chop off its stem and keep the cap that most people think of as the edible portion. Similarly, when we stem a word, we chop off its inflections and keep what hopefully represents the main essence of the word. Technically, it depends on the type of mushroom, and we're throwing away the mushroom stems while keeping the word stems. Nonetheless, I hope the metaphor is useful.

The best-known and most popular stemming approach for English is the Porter stemming algorithm, also known as the Porter stemmer. It is a collection of rules (or, if you prefer, heuristics) designed to reflect how English handles inflections. For example, the Porter stemmer chops both *apple* and *apples* down to *appl*, and it stems *berry* and *berries* to *berri*.

If we apply a stemmer to queries and indexed documents, we can increase recall by matching words against their other inflected forms. It is critical that we apply the same stemmer to both queries and documents.

You can find an implementation of the Porter stemmer in any major natural language processing library, such as NLTK and the Stanford NLP suite. You can find stemmers for other languages (or create your own) in Snowball.

Just as using a knife to chop a mushroom stem may leave a bit of the stem or cut into the cap, stemming algorithms sometimes remove too little or too much. For example, Porter stems both *meanness* and *meaning* to *mean*, creating a false equivalence. On the other hand, Porter stems *goose* to *goos* and *geese* to *gees*, when those two words should be equivalent.

In general, stemming algorithms err on the side of being too aggressive, sacrificing precision in order to increase recall.

### **Lemmatization**

Going back to our mushrooms, even an amateur chef knows that you shouldn't just chop off the stems with a knife. Instead, you should carefully remove the stems, cutting around them with a paring knife or gently twisting them off.

Lemmatization tries to take a similarly careful approach to removing inflections. Lemmatization does not simply chop off inflections, but instead relies on a lexical knowledge base like WordNet to obtain the correct base forms of words.

For example, WordNet lemmatizes *geese* to *goose* and lemmatizes *meanness* and *meaning* to themselves. In these examples, it outperforms than the Porter stemmer.

But lemmatization has limits. For example, Porter stems both *happiness* and *happy* to *happi*, while WordNet lemmatizes the two words to themselves. The WordNet lemmatizer also requires specifying the word's part of speech — otherwise, it assumes the word is a noun. Finally, lemmatization cannot handle unknown words: for example, Porter stems both *iphone* and *iphones* to *iphon*, while WordNet lemmatizes both words to themselves.

In general, lemmatization offers better precision than stemming, but at the expense of recall.

### **Canonicalization**

As we've seen, stemming and lemmatization are effective techniques to expand recall, with lemmatization giving up some of that recall to increase precision. But both techniques can feel like crude instruments.

If we generalize from stemming and lemmatization, what we have are ways to group together the related forms of a word, assigning them all a canonical form. While it's easy to rely on heuristics like Porter stemming and WordNet

lemmatization, there's nothing to stop us from building our own knowledge base for canonicalization.

For example, we can use the equivalence classes from Porter stemming and Wordnet lemmatization as candidates, and then use further processing — either automatic or editorial — to improve precision by estimating word similarity.

The automatic detection of word similarity could be based on corpus analysis (e.g., word embeddings like word2vec) or searcher behavior (queries, clicks from those queries, etc). Ideally, we use all available signals to train a machine-learned model.

Editorial detection requires human judgements. These could be at the level of token-token pairs, query-query pairs (to assess the tokens in context), or query-document pairs (to assess the end-to-end effect of query expansion).

Building your own knowledge base for canonicalization may sound like a lot of work, and it can be. I recommend that you start by leveraging off-the-shelf tools like Porter and WordNet, and then determine how much you want to invest in improving on them.

## **Summary**

Inflections pose an important challenge for query understanding. Stemming and lemmatization are out-of-the-box tools for managing inflections, and you should always consider them as ways to improve recall. But you need to be aware of their weaknesses, and you should consider investing in a canonicalization approach that establishes the right balance of precision and recall for your application.

**Previous: Spelling Correction**

**Next: Query Rewriting: An Overview**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigggle.

Feb 16, 2017 · 3 min read

## Query Rewriting: An Overview

Thus far, we've focused on query understanding at the character and token level. It's time move up the stack, on to entities and the query itself.

At this level, the most powerful techniques for query understanding fall into a broad class of strategies that we call query rewriting. Query rewriting automatically transforms search queries in order to better represent the searcher's intent. Query rewriting strategies generally serve two purposes: increasing recall and increasing precision.

This post provides an overview of query rewriting. We'll dive into the details of specific techniques in future posts.

### Increasing Recall

A key motivation for query rewriting search queries is to increase recall — that is, to retrieve a larger set of relevant results. In extreme cases, increasing recall is the difference between returning some (hopefully relevant) results and returning no results.

The two main query rewriting strategies to increase recall are query expansion and query relaxation.

### Query Expansion

Query expansion broadens the query by adding additional tokens or phrases. These additional tokens may be related to the original query terms as synonyms or abbreviations (we'll discuss how to obtain these in future posts); or they may be obtained using the stemming and spelling correction methods we covered in previous posts.

If the original query is an AND of tokens, query expansion replaces it with an AND of ORs. For example, if the query *ip lawyer* originally retrieves documents containing *ip* AND *lawyer*, an expanded query might retrieval documents containing (*ip* OR "*intellectual property*") AND (*lawyer* OR *attorney*).

Although query expansion is mostly valuable for increasing recall, it can also increase precision. Matches using expanded tokens may be more

relevant than matches restricted to the original query tokens. In addition, the presence of expansion terms serves as a relevance signal to improve ranking.

### **Query Relaxation**

Query relaxation feels like the opposite of query expansion: instead of adding tokens to the query, it removes them. Specifically, query relaxation increases recall by removing — or optionalizing — tokens that may not be necessary to ensure relevance. For example, a search for *cute fluffy kittens* might return results that only match *fluffy kittens*.

A query relaxation strategy can be naïve, e.g., retrieve documents that match all but one of the query tokens. But a naïve strategy risks optionalizing a token that is critical to the query’s meaning, e.g., replacing *cute fluffy kittens* with *cute fluffy*. More sophisticated query relaxation strategies use query parsing or analysis to identify the main concept in a query and then optionalize words that serve as modifiers.

Both query expansion and query relaxation aim to increase recall without sacrificing too much precision. They are most useful for queries that return no results, since there we have the least to lose and the most to gain. In general, we should be increasingly conservative about query expansion — and especially about query relaxation — as the result set for the original query grows.

### **Increasing Precision**

Query rewriting can also be used to increase precision — that is, to reduce the number of irrelevant results. While increasing recall is most valuable for avoiding small or empty result sets, increasing precision is most valuable for queries that would otherwise return large, heterogeneous result sets.

### **Query Segmentation**

Sometimes multiple tokens represent a single semantic unit, e.g., *dress shirt* in the query *white dress shirt*. Treating this segment as a quoted phrase, i.e., rewriting the query as *white “dress shirt”* can significantly improve precision, avoiding matches for white shirt dresses.

Query segmentation is related to tokenization: we can think of these segments as larger tokens. But we generally think of tokenization at the character level and query segmentation at the token level. We will discuss query segmentation algorithms in a future post.

### **Query Scoping**



Documents often have structure. Articles have titles and authors; products have categories and brands; etc. Query rewriting can improve precision by scoping, or restricting, how different parts of the query match different parts of documents.

Query scoping often relies on query segmentation. We determine an entity type for each query segment, and then restrict matches based on an association between entity types and document fields.

Query rewriting can also perform scoping at the query level, e.g., restricting the entire set of results to a single category. This kind of scoping is typically framed as a classification problem.

### **The Power of Query Rewriting**

Query rewriting is a powerful tool for taking what we understand about a query and using it to increase both recall and precision. Many of the problems that search engines attempt to solve through ranking can and should be addressed through query rewriting.

We'll spend the next posts diving into the details of the techniques introduced above.

**Previous: Stemming and Lemmatization**

**Next: Query Expansion**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigggle.

Mar 13, 2017 · 6 min read

# Query Expansion

A key application of query rewriting is increasing recall — that is, matching a larger set of relevant results. In extreme cases, increasing recall means the difference between returning some results and returning no results. More typically, query expansion casts a wider net for results that are relevant but don't match the query terms exactly.

## Overview

Query expansion broadens the query by introducing additional tokens or phrases. The search engine automatically rewrites the query to include them. For example, the query *vp marketing* becomes *(vp OR "vice president") marketing*.

The main challenge for query expansion is obtaining those additional tokens and phrases. We also need to integrate query expansion into scoring and address the interface considerations that arise from query expansion.

## Sources for Query Expansion Terms

We've covered spelling correction and stemming in previous posts, so we won't revisit them here. In any case, they aren't really query expansion. Spelling correction generally replaces the query rather than expanding it. Stemming is usually implemented by replacing tokens in queries and documents with their canonical forms, although it can also be implemented using query expansion.

Query expansion terms are typically abbreviations or synonyms.

## Abbreviations

Abbreviations represent exactly the same meaning as the words they abbreviate, e.g., *inc* means *incorporated*. Our challenge is recognizing abbreviations in queries and documents.

## Using a Dictionary

The simplest approach is to use a dictionary of abbreviations. There are many commercial and noncommercial dictionaries available, some intended

for general-purpose language and others for specialized domains. We can also create our own. Dictionaries work well for abbreviations that are unambiguous, e.g., *CEO* meaning *chief executive officer*. We simply match strings, possibly in combination with stemming or lemmatization.

But abbreviations are often ambiguous. Does *st* mean *street* or *saint*? Without knowing the query or document context, we can't be sure. Hence our dictionary has to be conservative to minimize the risk of matching abbreviations to the wrong words. We face a harsh trade-off between precision and recall.

## Using Machine Learning

A more sophisticated approach is to model abbreviation recognition as a supervised machine learning problem. Instead of simply recognizing abbreviations as strings, we train a model using examples of abbreviations in context (e.g., the sequence of surrounding words), and we represent that context as components in a feature vector. This approach works better for identifying abbreviations in documents than in queries, since the former supply richer context.

How do we collect these examples? We could use an entirely manual process, annotating documents to find abbreviations and extracting them along with their contexts. But such an approach would be expensive and tedious.

A more practical alternative is to automatically identify potential abbreviations using patterns. For example, it's common to introduce an abbreviation by parenthesizing it, e.g., *gross domestic product (GDP)*. We can detect this and other patterns automatically. Pattern matching won't catch all abbreviations, and it will also encounter false positives. But it's certainly more scalable than a manual process.

Another way to identify abbreviations is to use unsupervised machine learning. We look for pairs of word or phrases that exhibit both surface similarity (e.g., matching first letters or one word being a prefix of the other) and semantic similarity. A popular tool for the latter is Word2vec: it maps tokens and phrases to a vector space such that the cosine of the angle between vectors reflects the semantic similarity inferred from the corpus. As with a supervised approach, this approach will both miss some abbreviations and introduce false positives.

## Synonyms

Most of the techniques we've discussed for abbreviations apply to synonyms in general. We can identify synonyms using dictionaries, supervised learning, or unsupervised learning. As with abbreviations, we have to deal with the inherent ambiguity of language.

An important difference from abbreviations is that, since we can no longer rely on the surface similarity of abbreviations, we depend entirely on inferring semantic similarity. That makes the problem significantly harder, particularly for unsupervised approaches. We're likely to encounter false positives from antonyms and other related words that aren't synonyms.

Also, unlike abbreviations, synonyms rarely match the original term exactly. They may be more specific (e.g., *computer* -> *laptop*), more general (*ipad* -> *tablet*), or similar but not quite identical (e.g., *web* -> *internet*).

Hence, we not only need to discover and disambiguate synonyms; we also need to establish a similarity threshold. If we're using Word2vec, we can require minimum cosine similarity.

Also, if we know the semantic relationship between the synonym and the original term, we can take it into account. For example, we can favor a synonym that is more specific than the original term, as opposed to one that is more general.

## Scoring Results

Query expansion uses query rewriting to increase the number of search results. How do we design the scoring function to rank results that match because of query expansion?

The simplest approach is to treat matches from query expansion just like matches to the original query. This approach works for abbreviation matches that completely preserve the meaning of the original query terms — assuming that we don't make any mistakes because of ambiguity. Synonym matches, however, may introduce subtle changes in meaning.

A more sophisticated approach is to apply a discount to matches from query expansion. This discount may be a constant, or it can reflect the expected change in meaning (e.g., a function of the cosine similarity). This approach, while heuristic, integrates well with hand-tuned scoring functions, such as those used in many Lucene-based search engines.

The best — or at least most principled — approach is to integrate query expansion into a machine learned ranking model using features (in the machine learning sense) that indicate whether a document matched the

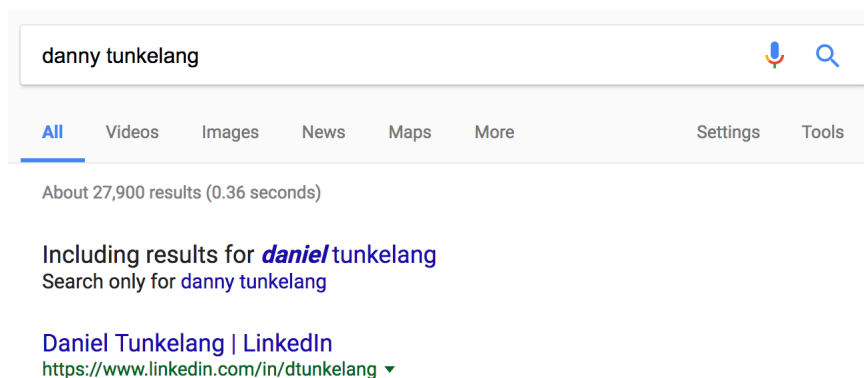
original query terms or terms introduced through query expansion. These features should also indicate whether the expansion was through an abbreviation or a synonym, the similarity of the synonym, etc.

Integrating query expansion into a machine-learned ranking model is a bit tricky. We can't take full advantage of pre-existing training data from a system that hasn't performed query expansion. Instead, we start with a heuristic model to collect training data (e.g., one of the previously discussed approaches) and then use that data to learn weights for query expansion features.

## Interface Considerations

Query rewriting is always a gamble: it's an attempt to do what the searcher meant, rather than what the searcher said. It's important to mitigate this gamble through communication.



We can communicate that the search engine has performed query expansion by including an explicit message in the response. Here's an example of a search for my name on Google:



If the query expansion is aggressive, then it's a good idea to inform the search and provide a one-click opportunity to undo it.

It's even more valuable to communicate at the result level, by highlighting the expanded terms used to match each result. Here's another example from Google: note how the second result bolds the term *human-computer interaction* that is an expansion of *hci*.

hci for search engines



All

News

Shopping

Images

Videos

More

Settings

Tools

About 1,010,000 results (0.70 seconds)

**HCI and Information Search: Capturing Task and ... - SpringerLink**

[link.springer.com/chapter/10.1007/978-3-540-73345-4\\_43](https://link.springer.com/chapter/10.1007/978-3-540-73345-4_43)

by NK Agarwal - 2007 - [Cited by 1](#) - [Related articles](#)

HCI and Information Search: Capturing Task and Searcher Characteristics ... Search engines provide a 'one-size-fits-all' model, which do not adequately cater to ...

**Usability Guidelines for Desktop Search Engines - Springer**

[link.springer.com/chapter/10.1007/978-3-642-39232-0\\_20](https://link.springer.com/chapter/10.1007/978-3-642-39232-0_20)

by M Burghardt - 2013 - [Cited by 3](#) - [Related articles](#)

Human-Computer Interaction. ... Usability Guidelines for Desktop Search Engines ... usability testing heuristic evaluation desktop search engines usability ...

## Summary

Query expansion is a valuable tool for increasing recall. Matching abbreviations and synonyms helps searchers find what they're looking for, even when their language doesn't match the results exactly. We can use off-the-shelf dictionaries or create our own, either manually or using machine learning. Beyond the challenge of identifying abbreviations and synonyms, we have to make changes to how we score and present results.

Query expansion is a lot of work, and it introduces complexity and risk. But the increased recall usually justifies the investment.

**Previous: Query Rewriting: An Overview**

**Next: Query Relaxation**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigg.

Mar 28, 2017 · 5 min read

## Query Relaxation

In the previous post, we discussed query expansion as a way to increase recall. In this post we'll discuss the other major technique for increasing recall: query relaxation.

Query relaxation feels like the opposite of query expansion. Instead of adding tokens to the query, we remove them. Ignoring tokens makes the query less restrictive and thus increases recall. An effective query relaxation strategy removes only tokens that aren't necessary to communicate the searcher's intent.

Let's consider four approaches to query relaxation, in increasing order of complexity: stop words, specificity, syntactic analysis, and semantic analysis.

### Stop Words

The simplest form of query relaxation is ignoring stop words. Stop words are words like *the* and *of*: they generally don't contribute meaning to the query; hence, removing them preserves the intent while increasing recall.

But sometimes stop words matter. There's a difference between *king hill* and *king of the hill*. And there's even a post-punk band named *The The*. These edge cases notwithstanding, stop words are usually safe to ignore.

Most open-source and commercial search engines come with a list of default stop words and offer the option of ignoring them during query processing (e.g. Lucene's StopFilter). In addition, Google has published lists of stop words in 29 languages.

### Specificity

Query tokens vary in their specificity. For example, in the search query *black hdmi cable*, the token *hdmi* is more specific than *cable*, which is in turn more specific than *black*. Specificity generally indicates how essential each query token is to communicating the searcher's intent. Using specificity, we can determine that it's more more reasonable to relax the query *black hdmi cable* to *hdmi cable* than to *black cable*.

## Inverse Document Frequency

We can measure token specificity using inverse document frequency (idf). The inverse (idf is actually the logarithm of the inverse) means that rare tokens — that is, tokens that occur in fewer documents — have a higher idf than those that occur more frequently. Using idf to measure token specificity is a generalization of stop words, since stop words are very common words and thus have low idf.

Information retrieval has used idf for decades, ever since Karen Spärck Jones’s seminal 1972 paper on “A statistical interpretation of term specificity and its application in retrieval”. It’s often combined with term frequency (tf) to obtain tf-idf, a function that assigns weights to query tokens for scoring document relevance. For example, Lucene implements a `TFIDFSimilarity` class for scoring.

But be careful about edge cases. Unique tokens, such as proper names or misspelled words, have very high idf but don’t necessarily represent a corresponding share of query intent. Tokens that aren’t in the corpus have undefined idf — though that can be fixed with smoothing, e.g., adding 1 before taking the logarithm. Nonetheless, idf is a useful signal of token specificity.

## Lexical Databases

A completely different approach to measuring specificity is to use a lexical database (also called a “knowledge graph”) like WordNet that arranges concepts into semantic hierarchies. This approach is useful for comparing tokens with a hierarchical relationship, e.g., *dog* is more specific than *animal*. It’s less useful for tokens without a hierarchical relationship, e.g., *black* and *hdmi*.

A lexical database also enables a more nuanced form of query relaxation. Instead of ignoring a token, we can replace it with a more general term, also known as a hypernym. We can also use a lexical database for query expansion.

## Syntactic Analysis

Another approach to query relaxation to use a query’s syntactic structure to determine which tokens are optional.

A large fraction of search queries are noun phrases. A noun phrase serves the place of a noun — that is, it represents a thing or set of things. A noun phrase



can be a solitary noun, e.g., *cat*, or it can be a complex phrase like *the best cat in the whole wide world*.

We can analyze search queries using a part-of-speech tagger, such as NLTK, which in turn allows us to parse the overall syntactic structure of the query. If the query is a noun phrase, parsing allows us to identify its head noun, as well as any adjectives and phrases modifying it.

A reasonable query relaxation strategy preserves the head noun and removes one or more of its modifiers. For example, the most important word in *the best cat in the whole wide world* is the head noun, *cat*.

But this strategy can break down. For example, if the query is *free shipping*, the adjective *free* is at least as important to the meaning as the head noun *shipping*. Syntax does not always dictate semantics. Still, emphasizing the head noun and the modifiers closest to it usually works in practice.

### **Semantic Analysis**

The most sophisticated approach to query relaxation goes beyond token frequency and syntax and considers semantics — that is, what the tokens mean, particularly in relation to one another.

For example, we can relax the query *polo shirt* to *polo*, since *shirt* is implied. In contrast, relaxing *dress shirt* to *dress* completely changes the query's meaning. Syntax isn't helpful: in both cases, we're replacing a noun phrase with a token that isn't even the head noun. And *shirt* is no more specific than *polo* or *dress*. So we need to understand the semantics to relax this query successfully.

We can use the Word2vec model to embed words and phrases into a vector space that captures their semantics. This embedding allows us to recognize how much the query tokens overlap with one another in meaning, which in turn helps us estimate the consequence of ignoring a token. Word2vec also allows us to compute the similarity between a token and a phrase containing that token. If they're similar, then it's probably safe to relax the query by replacing the phrase with the token.

### **Relax but be Careful**

Like query expansion, query relaxation aims to increase recall while minimizing the loss of precision. If we already have a reasonable quantity and quality of results, then query relaxation probably isn't worth the risk.

Query relaxation is most useful for queries that return few or no results, since those are the queries for which we have the least to lose and the most to gain. But remember that more results doesn't necessarily mean better results. Use query relaxation, but use it with care.

**Previous: Query Expansion**

**Next: Query Segmentation**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twiggie.

Apr 28, 2017 · 4 min read

## Query Segmentation

The previous two posts focused on using query rewriting to increase recall. We can also use query rewriting to increase precision — that is, to reduce the number of irrelevant results. While increasing recall helps us avoid small or empty result sets, increasing precision helps us avoid large result sets that are full of noise.

In this post, we'll discuss the first of two query rewriting strategies used to increase precision: query segmentation. We'll first talk about how to perform query segmentation, and then about how to use query segmentation to increase precision through query rewriting.

### Semantic Units

Query segmentation attempts to divide the search query into a sequence of semantic units, each of which consists of one or more tokens. For a single-token query like *machine*, there's only one possible segmentation. Multiple-token queries, like *machine learning framework*, admit multiple possible segmentations, such as "*machine learning*" *framework* and *machine* "*learning framework*". In theory, the number of possible segmentations for a query grows exponentially with the number of tokens; in practice, there are only a handful of plausible segmentations.

The goal of query segmentation is to identify the query's semantic units. In our *machine learning framework* example it's clear that the correct segmentation is "*machine learning*" *framework* — that is, the searcher is interested in frameworks for machine learning, rather than something to do with machines and learning frameworks. Identifying this segmentation is critical to understanding the query and thus returning precise matches.

So how do we automatically identify the correct segmentation?

### Dictionary Approach

The simplest approach is to obtain a dictionary of phrases appropriate to the application domain, and then to automatically treat the phrases as segments when they occur in queries. For example, a search engine for computer science articles could use a dictionary of common terms from that domain. Hopefully such a dictionary would include the term *machine learning*.

A dictionary-based approach requires a mechanism to resolve overlapping phrases, e.g., to segment the query *machine learning framework* if the dictionary includes both *machine learning* and *learning framework*. To keep things simple, we can associate each phrase with a score — such as its observed frequency in a subject-specific corpus — and favor the phrase with the highest score when there's overlap.

Such a strategy tends to work reasonably well, but it can't take advantage of query context. We'll return to this point in a moment.

### **Statistical Approach**

If we can't buy or borrow a dictionary, we can create one from a document corpus. We analyze the corpus to find collocations — that is, sequences of words that co-occur more often than would be expected by chance.

We can make this determination using statistical measures like mutual information, t-test scores, or log-likelihood. It's also possible to obtain collocations using Word2vec.

Once we have a list of collocations and associated scores, we can create a dictionary by keeping the collocations with scores above a threshold. Choosing the threshold is a trade-off between precision and coverage. We can also review the list manually to remove false positives.

### **Supervised Machine Learning**

A more sophisticated approach to query segmentation is to model it as a supervised machine learning problem.

Query segmentation is a binary classification problem at the token level. For each token, we need to decide whether it continues the current segment or begins a new one. Given a collection of correctly segmented queries, we can train a binary classifier to perform query segmentation.

As with all machine learning, our success depends on how we represent the examples as feature vectors. Features could include token frequencies, mutual information for bigrams, part-of-speech tags, etc. Bear in mind that some features are more expensive to compute than others. Slow computation may be an acceptable expense or inconvenience for training, but it can be a show-stopper if it noticeably slows down query processing.

A supervised machine learning approach is more robust than one based on a dictionary, since it can represent context in the feature vector. But it's a more

complex and expensive to implement. It's probably better to start with a simple approach and only invest further if the results aren't good enough.

### **Using Query Segmentation for Query Rewriting**

Now that we've obtained a query segmentation, what do we do with it? We rewrite the query to improve precision!

A straightforward approach is to auto-phrase segments — that is, to treat them as if they were quoted phrases. Returning to our example, we'd rewrite *machine learning framework* as “*machine learning*” *framework*, filtering out results that contain the tokens *machine* and *learning* but not as a phrase. The approach certainly increases precision, but it can be so drastic as to lead to no results. For this reason, many search engines boost phrase matches but don't require them.

A more nuanced approach is to couple auto-phrasing with query expansion approaches like stemming, lemmatization, and synonym expansion. Continuing with our example, *machine learning framework* could include results that match “*machine learned*” *framework* or “*machine learning*” *infrastructure*. This approach can achieve strong precision and recall.

Finally, if we're using query relaxation, it's important for relaxation to respect query segmentation by not breaking up segments.

### **Summary**

Searchers use multiple-word phrases intuitively and are unpleasantly surprised when search engines fail to recognize those phrases. If a significant number of searchers are quoting phrases in their queries, you're doing it wrong. If you can only invest in two areas of query understanding, I recommend that you prioritize spelling correction and query segmentation.

### **Previous: Query Relaxation**

### **Next: Query Scoping**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigg.

May 16, 2017 · 4 min read

## Query Scoping

In the previous post, we discussed how query segmentation improves precision by grouping query words into semantic units. In this post, we'll discuss query scoping, another query rewriting technique that improves precision by matching each query segment to the right attribute.

### Leveraging Structure

Documents or products often have an explicit structure that mirrors how people search for them.

For example, if you search for *black michael kors dress* on a site that sells clothing, you intend black as a color, Michael Kors as a brand, and dress as a category.

Similarly, if you search for *microsoft ceo* on LinkedIn, you're looking for Microsoft as an employer and CEO as a job title.

Query scoping rewrites queries to take advantage of this structure.

### Query Tagging

Query scoping begins with query tagging. Query tagging maps each query segment to the intended corpus attribute. It's a machine learning problem — specifically, a non-binary classification problem. In the information retrieval literature, query tagging is a special case of named-entity recognition (NER), which can be applied to arbitrary text documents.

In our clothing example, the attributes include category, brand, and color. LinkedIn's attributes include person name, company name, and job title. In general, attributes are specific to the domain and corpus.

Ideally, document structure is explicitly represented in the search index. If not, we may be able to extract document structure through algorithms for document understanding, also known as information extraction.

Regardless of how we obtain document structure, we need to establish the corpus attributes in advance. Query tagging is a supervised machine learning problem, which means that we can't map query segments to attributes we

didn't anticipate. At best, we can label those segments as "unknown" because the classifier isn't able to pick an attribute for them.

## Training Data

We generate training data for query tagging from a representative set of segmented, labeled queries. For example, our training data could include examples like (*black, michael kors, dress*) -> (*Color, Brand, Category*).

Obtaining a representative set of queries from search logs is straightforward for a search application that's already in production. To train a query tagger for a new search application, we'll need to be more creative, either synthesizing a set of queries or borrowing a search log from a similar application. It may be easier to launch a search application without tagging, and then train a tagger after we've collected a critical mass of search traffic.

Let's assume that we already have a query segmentation algorithm, as discussed in the previous post, and that we've applied it to our set of queries. Query tagging assumes — and thus requires — query segmentation.

We then need to label the query segments. We can use human labelers, but this process is slow and expensive. Human labelers also need to be familiar with the domain in order to provide robust judgements.

A more efficient approach is to obtain labels from our search logs — specifically from clicks on search results. For each click, we look at the query-result pair and label each query segment with the attribute it matches in the results. For example, if we look at a click for the search *black michael kors dress*, we'll probably find that *black* matches the color attribute, Michael Kors matches the brand attribute, and *dress* matches the category attribute.

This approach works when each segment matches precisely one attribute in the clicked result. If a segment matches multiple attributes, we can exclude the click from our training data, or we can randomly select from the attributes. If a segment doesn't match an attribute, we label it as "unknown".

Finally, we have to represent the training examples as feature vectors. We can create features from tokens, stemmed or lemmatized tokens, or we can look below the token levels at the characters. If we have additional context about the searcher, we can also incorporate it into the feature vector.

## Building the Model

Once we have training data, we can use various approaches to build a machine-learned model. A common choice for query tagging is a conditional random field (CRF). The Stanford NLP package provides a CRF named-entity recognizer that is suitable for query tagging.

It may be tempting to use a cutting-edge approach like deep learning. But bear in mind that more data — and better data — usually beats better algorithms. Robust query tagging relies mostly on robust generation of training data.

### **From Tagging to Scoping**

Having tagged each query segment, we rewrite the query to scope the query.

A straightforward approach is to only match each segment against its associated attribute, as if the searcher had done so explicitly using an advanced search interface. This approach tends to optimize for precision, but it can be a bit unforgiving when it comes to recall.

A problem with this literal approach is that some attributes are difficult to distinguish from one another. For example, it's often difficult to determine whether a query segment in a LinkedIn search matches a skill or part of a job title (e.g., a Developer who knows Java vs. Java Developer). Rather than count our ability to make this distinction, we can ignore it, effectively combining skill and job title into a single attribute.

If we have a similarity matrix for our attributes, or we implement the query tagger as a multiclass classifier, then we can use the query tagger to restrict matches to the best attributes for each segment. Determining the number of attributes to match is a precision / recall tradeoff.

Finally, since query scoping is an aggressive way to increase precision, we can combine it with techniques like query expansion and query relaxation to increase recall. By requiring query expansion and relaxation to respect query scoping, we can achieve a good balance of precision and recall.

### **Summary**

Query scoping is a powerful technique to increase precision by leveraging the explicit structure of the corpus and the implicit structure of queries. It requires more work than most of the other techniques we've discussed, but it's worth it: it's one of the most effective ways to capture query intent.

### **Previous: Query Segmentation**





Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigg.

May 30, 2017 · 3 min read

# Entity Recognition

In the previous post on query scoping, we discussed query tagging as a special case of named-entity recognition (NER). In this post, we'll dive a little bit deeper into the topic of entity recognition.

## Entities

Entity recognition presumes that we know what kind of entities we're trying to recognize. But defining what constitutes an entity is tricky. At a high level, we can distinguish between named entities and terms.

## Named Entities

Named entities have two distinguishing features. First, they have names — that is, they are proper nouns. In English, that means they're usually capitalized. Second, they have types, such as person, place, or organization. A named entity is a specific, named instance of a particular entity type.

The specificity of named entities makes recognizing them useful for both query understanding and document understanding. In addition, named entities often have relationships with one another, comprising a semantic network or knowledge graph. For example, Sundar Pichai is the CEO of Google, which is located in Mountain View. Query rewriting can use these relationships among entities to improve both precision through disambiguation and recall through expansion.

Named-entity recognizers can be rule-based (e.g., using a collection of regular expressions) or machine-learned. Rule-based recognizers are simple and understandable, but it's hard to achieve good accuracy without machine learning. As discussed in the previous post, training a machine-learned named-entity recognizer requires a collection of segmented, labeled queries.

One subtlety we did not address in that post is that recognizing an entity type is not the same as identifying a specific entity. For example, recognizing that *sundar pichai* is a person is not the same as recognizing that he's the specific person who is the CEO of Google. Many named-entity recognizers, such as do not perform this further task of entity identification.

Entity identification requires that the training data labels be those specific entities, rather than their generic entity types. Entity identification plays a critical role for domains like social networking and media, but building it requires significant data assets, as well as investment to maintain them.

## **Terms**

In contrast, terms generally have neither names nor types. Terms are simply words or phrases — usually noun phrases — that carry a distinctive amount of meaning, particularly in the context of an application domain. If the corpus were a book, then terms are what you'd expect to find in its glossary. A rule of thumb is that terms have Wikipedia entries. Conversely, Wikipedia strives to be a universal glossary.

What terms lack in structure and specificity, they make up for in generality. The terms in a document carry most of the document's meaning. We can think of the list of terms as a document summary, particularly for the purpose of search indexing. Similarly, terms represent the most important query segments.

Terminology extraction algorithms typically identify noun phrases in a corpus using part-of-speech tagging, and then select terms from those noun phrases based on frequency and other statistical measures that reflect specificity and distinctiveness. While it's challenging to build a knowledge graph on top of unstructured terms, it's possible to relate terms to one another through text mining algorithms, or using the techniques described in the earlier post on query expansion.

Because terms lack structure, it's hard to do much more than string matching for term recognition. In particular, it's difficult to disambiguate words or phrases. We may be able to do so in the context of a document, but it's hard to do so for short queries.

## **Leveraging Entity Recognition**

Entity recognition focuses both query understanding and document understanding on the most salient words and phrases. As such, it can be the lynchpin of a query rewriting strategy — from precision-increasing query scoping to recall-increasing expansion and relaxation.

Entity recognition is what elevates search from strings to things. It takes a lot of work to implement robust entity recognition, but doing so dramatically simplifies the rest of the information retrieval problem.

## **Previous: Query Scoping**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigg.

Jul 27, 2017 · 3 min read

# Taxonomies and Ontologies

In order to understand queries, it's important to ground that understanding in a knowledge base. Two common ways to represent a knowledge base are taxonomies and ontologies.

## Taxonomies

Taxonomies date back to Aristotle as a way to organize knowledge. A taxonomy is a hierarchical classification system: a tree that starts from a universal root concept and progressively divides it into more specific child concepts. The nodes of a taxonomy correspond to concepts, connected by branches or edges directed from the root node towards the leaf nodes.

Perhaps the most famous taxonomy in use today is the Dewey Decimal System, used by libraries to organize their materials by subject. For example, *butterflies* are classified as *Natural Science (500)* -> *Zoological Sciences (590)* > *Other Invertebrates (595)* -> *Insects (595.7)* -> *Lepidoptera (595.78)* -> *Butterflies (595.789)*.

A proper taxonomy is a strict hierarchy — that is, every node other than the root has precisely one parent. In practice, it's common to relax this constraint by allowing nodes to have multiple parents. For example, in a consumer product taxonomy, *USB Cables* might be a descendant of both *Computer Accessories* and *Cell Phone Accessories*.

The rigid constraints of a single hierarchy have led many people to use faceted classification systems instead. In a faceted classification, a topic is composed from nodes in multiple, independent hierarchies. We'll discuss faceted classification — and its use in faceted search — in a future post.

Regardless, it's a good idea to minimize the amount of shared parentage in a taxonomy, as shared parentage introduces complexity. A little bit can go a long way to model the messiness of the real world. But it's strictly forbidden to have cyclical relationships, e.g., a concept can't be its own grandparent. The chain of parents from a node must always lead to the root.

## Ontologies

Ontologies are somewhat more complex than taxonomies. While taxonomies represent a collection of topics with “is-a” relationships, ontologies make it possible to express a much richer collection of objects and relationships, such as “has-a” and “use-a”. There can be classes, instances of those classes, class attributes, and general relationships among classes instances. Moreover, the relationships aren’t necessarily binary — for example, a *co-worker* relationship might connect three instances: two people and the place where they worked together.

The complexity of ontologies makes them powerful but dangerous. In order to represent a complex domain like medicine and reason over it, it may be necessary to use an ontology like SNOMED CT. But to organize products or documents on a site aimed at non-expert consumers, it’s probably best to stick to a simple taxonomy or faceted classification system.

### **Query Understanding**

We can implement query understanding by mapping queries to taxonomies or ontologies. The simplest approach is to treat taxonomy nodes as categories and train a classifier to map queries to categories, e.g., mapping the query *knapsack* to the node *Bags and Purses* -> *Backpacks*. A far more complex approach is to parse queries and represent them as networks of relationships in an ontology.

Either approach requires a significant amount of effort and data — whether from explicit human judgments, implicit behavioral judgments, or curated knowledge — to establish a mapping. Just having a taxonomy or ontology as a knowledge base doesn’t tell us how to apply it to queries.

But a knowledge base establishes the platform on which to build query understanding systems. A knowledge base — even it’s just a simple taxonomy — is essential for query understanding.

### **Previous: Entity Recognition**

### **Next: Autocomplete**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twiggie.

Aug 28, 2017 · 4 min read

# Autocomplete

In the past decade, autocomplete has become a required feature for search engines. Today, searchers who type into a search box expect to see autocomplete suggestions; otherwise, they'll conclude that search is broken.

Autocomplete isn't just a way to help searchers type less. Autocomplete helps searchers express their intent. When autocomplete works as intended, query understanding guides the process of typing a search query.

## Query Probability

Autocomplete predicts complete search queries from partial ones. We can model this process using conditional probability: given a partial query, we want to offer searchers the most probable query that completes it.

We can use historical query logs to compute the probability of a query based on how frequently it appears in the log. For example, if our log contains a 1,000,000 queries and 1,000 of them are for *pants*, then the probability of *pants* is  $1,000/1,000,000 = 0.001$ .

That's the a priori probability. As the searcher types in more characters, the conditional probability of a query either increases (because the denominator decreases) or goes to zero because the query doesn't start with the entered prefix. Continuing with our example, if 50,000 of the 1,000,000 queries start with *pa*, then  $\Pr(\text{pants}|\text{pa}) = 1,000/50,000 = 0.02$ . But  $\Pr(\text{pants}|\text{pat}) = 0$ . Or does it? We'll get back to that in a moment.

## Not as Simple as it Looks

Computing probabilities this way is simple, but reality is a bit more complex. Here's a taste of that complexity:

- Query frequency in the logs doesn't take into account recency, seasonality, or other time-dependent factors. There are algorithmic and editorial strategies to emphasize recency or other time-dependent factors, but they all have trade-offs. There's no one right approach.
- There are also non-temporal factors: we may want to build a regression model using features like location, gender, session context, etc. These

approaches generally lead to more accurate probability estimates, but they also increase the risk of overfitting.

- The partial query may not be a prefix, for example, we shouldn't assume  $\Pr(\text{mens pants}|pa)=0$  just because *pa* isn't a prefix of *mens pants*. We should strongly prefer prefixes, but we should allow for exceptions.
- The partial query may be misspelled, e.g. something typing in *pat* may intend to type *pants*. In that case, it's necessary to combine autocomplete with spelling correction — something we'll cover as an advanced topic in a later post.
- You may want to exclude autocomplete suggestions from the logs when computing query probabilities, in order to avoid a positive feedback loop.

## Query Performance

Query popularity tells us how often people express their intent through a particular search query. But it doesn't tell us how often we manage to understand that intent and deliver a successful search experience.

We can use clicks or actions (e.g., purchases in the context of a shopping site) as indicators of search success. We'll save a deeper discussion of query performance for another post, but for now let's assume that clicks indicate successful searches.

Then, instead of counting all of the times a query appears in the log, we can count only the queries that lead to clicks. In other words, we choose the autocomplete suggestions that maximize the conditional probability of a click, given the entered prefix.

## A Trade-Off

We can think of the conditional probability of a click given a prefix as the product of two factors:

- The conditional probability of the query, given the prefix.
- The conditional probability of a click, given the query.

Continuing our earlier example, if the query *pants* has a click-through rate (CTR) of 0.1, then the probability of a click for *pants* given the prefix *pa* is  $0.02 * 0.1 = 0.002$ .

Let's add another query to our example: the query *pant cuffs* with only 100 occurrences in the log (a tenth of *pants*), but a CTR of 0.5. The probability of a click for *pant cuffs* given the prefix *pa* is  $0.002 * 0.5 = 0.001$ .

The probability of a click for *pants* given the prefix *pa* is double the probability of a click for *pant cuffs*. Yet there's something disconcerting about favoring a query with a CTR of 0.1 over a query with a CTR of 0.5, just because the lower-performing query is much more popular.

The risk of favoring low-performing popular queries is that we'll lead the searcher to an unsuccessful experience. If we instead favor queries that perform better but are less popular, we may require searchers to type more, but we're less likely to lead them astray.

There's no one answer on how to manage the trade-off between query popularity and performance. But it's good to at least set a minimum threshold of query performance for autocomplete suggestions.

### **Summary**

Autocomplete isn't just about helping searchers type less. It's a way to guide searchers to successful search experiences. It's easy to build a basic autocomplete system, but we've seen there's a lot more to computing query probabilities and taking into account query performance. Building a robust autocomplete system can be a major endeavor.

But remember that, for a large fraction of your searchers, autocomplete is the entire search experience. Invest accordingly.

**Previous: Taxonomies and Ontologies**

**Next: Autocomplete and User Experience**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigggle.

Sep 1, 2017 · 3 min read

## Autocomplete and User Experience

The previous post focused on how to determine the best autocomplete suggestions based on query probability and query performance. In this post, we'll dive into some user experience concerns specific to autocomplete.

### Less is More

How many autocomplete suggestions should we show to searchers? As a general principle, we should show them as few suggestions as possible. The set of suggestions shouldn't be an exhaustive list that requires effort to scan — after all, it's not that hard for the searcher to type one more character.

At the very least, all the suggestions should be visible on a single screen, without requiring the searcher to scroll down the page or use a scrollbar. This requirement poses a particularly tight constraint for mobile devices.

While every application is different, a good rule of thumb is to show at most five suggestions — and certainly no more than ten.

### Order

There are two standard ways to order autocomplete suggestions: by score (i.e., query probability or some combination of query probability and query performance), or in alphabetical order.

On one hand, ordering suggestions by score minimizes how many results the searcher has to scan — assuming that the scoring function is working as intended. On the other hand, ordering suggestions alphabetically makes them easier for the searcher to scan.

Both approaches have their merits. Ordering suggestions by score works better when the top suggestion is dominant, i.e., the next-best suggestion has a significantly lower score. Ordering suggestions alphabetically works better when the differences in scores are insignificant.

A compromise is to order suggestions by score, but to use rounding to eliminate insignificant differences in scores, and then to break ties by sorting alphabetically.



## Structured Suggestions

Autocomplete suggestions don't have to be raw strings: they can take advantage of structured data in the index. For example *jea* could suggest *jeans in Men's Clothing* and *jeans in Women's Clothing*, both of which scope the query *jeans* to a particular department. Autocomplete can serve as a painless entry point for faceted search, especially on devices with limited screen real estate.

This approach works especially well when there is a dominant suggestion that leads to a large, diverse result set that the searcher will want to refine. Since structured suggestions replace a single suggestion with multiple suggestions, they work best when there's a single dominant suggestion worth expanding to displace lower-scoring suggestions.

## Instant Search

Instant search goes a step beyond autocomplete: instead of suggesting search queries, it shows searchers actual search results as they type.

Instant search works best for what information scientists call "known-item search": the searcher has a single result in mind and can specify it precisely through a search query, such as a product name or document title.

Instant search offers the prospect of nearly frictionless search. Bear in mind, however, that not every search are known-item search. Instant search isn't useful for exploratory search. Moreover, it introduces complexity and distraction, and it competes with autocomplete for the searcher's attention. The search engine should only show instant search results when it is confident that the intended search query is a known-item search.

## Summary

We've discussed several user experience considerations for autocomplete: the number of suggestions to present, the order of suggestions, the use of structured suggestions, and instant search. There are other user experience considerations, such as formatting and layout, which are beyond the scope of this post. There's no single best approach for designing an autocomplete user experience, but hopefully this post provides useful guidelines.

**Previous: Autocomplete**

**Next: Contextual Query Understanding**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twiggie.

Sep 11, 2017 · 2 min read

## Contextual Query Understanding: An Overview

So far, we've focused on understanding searchers based entirely on the words they type into the search box. But search doesn't occur in a vacuum. In the next posts, we'll look at how context informs query understanding. Here's an overview of the different kinds of context we'll consider.

### Session Context

The most immediate context for a search query is a search session, a sequence of activities that the searcher performs in order to pursue an information-seeking task. Knowing that two or more searches take place in the same session can help us understand them better. For example, when a searcher performs two searches in a row, it's often the case that the second search is an attempt to clarify or refine the first.

### Location as Context

Location, location, location. Sometimes location serves as an implicit part of the search query, such as when someone is searching for local businesses. Other times, location can play a more subtle role, influencing query understanding through the use of aggregate data about searchers from a particular location. We often know the searcher's location, and we can use the information to improve query understanding.

### Seasonality

Search intents on shopping sites show highly seasonal patterns — just accessing a site around certain holidays serves as a strong intent signal. Even time of day can tell us about intent, e.g. a search for restaurants at noon on a weekday may have different intent (e.g., a good place for a business lunch) than the same search on a Saturday evening (e.g., a good place for a dinner date). Whether it's time of day or time of year, when someone performs a search can help us determine what the search means.

In the next posts, we'll dive more deeply into each of these kinds of context.

**Previous: Autocomplete and User Experience**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigggle.

Oct 9, 2017 · 3 min read

## Session Context

The most immediate context for a search query is a search session, a sequence of activities the searcher performs in order to pursue an information-seeking task. Relating a search query to the searcher's previous session activities helps us understand the searcher's intent.

### Recognizing Query Refinement

When a searcher performs two searches in a row, it's often the case that the second search is an attempt to refine the first. How do we recognize query refinement, and what should we do once we recognize it?

There are three broad classes of query refinement:

- Narrowing: e.g., *iphone games* -> *free iphone games*.
- Broadening: e.g., *iphone kids games* -> *iphone games*.
- Lateral refinement, e.g., *iphone games* -> *android games*.

As we can see, these refinement classes correspond to adding, removing, and replacing tokens. This correspondence isn't exact — for example, the query *iphone games* narrows *mobile games*. But it works well in practice.

### Why Searchers Refine

When searchers refine queries, the type of refinement provides insight into their intent. For example, a searcher who refine from *iphone games* to *free iphone games* probably looked at the results for the first query, saw iPhone games that were not free, and then supplied additional information to narrow the results.

In general, it's safe to assume that anyone refining a query is dissatisfied with the results. Refinement requires effort, and searchers, like all human beings, tend to be lazy. Hence, searchers expect their refined queries to return results that are substantially different than their original queries — at least on the first page.

Specifically, narrowing implies a desire to improve precision by filtering out irrelevant results, while broadening implies a desire to improve recall by

retrieving more relevant results. The search engine should recognize what the searcher is trying to achieve, act accordingly, and clearly communicate its understanding back to the searcher. In particular, narrowing should return fewer results, while broadening should return more results.

### **Personalization using Session Context**

Consider the following scenario. You search on an ecommerce site for shirts, and then narrow down to men's shirts through faceted navigation.

Continuing your online shopping, you search for pants and find that you again need to narrow down to men's pants. After perform a similar sequence to find underwear, you wonder how the search engine can be so oblivious.

A little bit of session context goes a long way. If you are searching for shirts, pants, and underwear in a single session, then you are probably buying them for the same person (e.g., yourself) — which means the search engine should preserve attributes like gender and size.

As with all personalization, it's important to be cautious. Overly aggressive personalization leads to false positives that cause more harm than good. But conservative inferences allow the search engine to understand queries more intelligently.

### **Summary**

The use of session context for query understanding can be arbitrarily sophisticated. Any query understanding algorithm based on machine learning can incorporate session features into its model. In particular, search engines should recognize query refinement and respond to it intelligently.

But it's important not to get so excited about session context that we ignore the query itself. Session context is useful, but the query will always be the strongest signal of the searcher's intent.

### **Previous: Contextual Query Understanding**

### **Next: Location as Context**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twiggie.

Oct 23, 2017 · 3 min read

## Location as Context

Location often provides a strong signal of searcher intent. Sometimes location acts as an implicit part of the search query, such as when people are searching for local businesses. Other times, location can subtly influence query understanding: aggregate data about searchers from a particular location helps establish a more precise query interpretation.

### Location as an Implicit Part of the Query

For some searches, the searcher's location acts as an implicit part of the search query. The canonical case is a search for a local business, e.g., a search for *restaurants* indicates a desire to find restaurants near the searcher's current location, which hopefully the search engine can determine.

If the search engine is embedded in a mapping application like Google or Apple Maps, or business directory like Yelp, then nearly all queries carry local intent. In other cases, the search engine has to determine which queries have local intent. In the context of web search, for example, someone searching for *starbucks* probably wants to find the closest one, while someone searching for *starbucks suppliers* is less likely to care about the suppliers' locations.

Determining whether a query has local intent is a classification problem, and the implementation of the classifier is highly dependent on the search application. But, as we'll discuss in a moment, we can often learn from searcher behavior — specifically, which documents attract searchers from particular locations, and which queries exhibit engagement with results that depends on the searcher's location.

### Location as an Intent Signal

Sometimes location isn't part of the query intent, but it's still useful as a signal. For example, *football* means something different in the United States than it does in the rest of the English-speaking world. People searching for *jackets* are likely to have preferences that reflect local climate.

Again, determining whether location is a useful intent signal for a particular query is a classification problem.

## **How to Locate a Document**

While our focus is on query understanding, it helps to shift our focus to the documents. Specifically, we want to know when a document is associated with a particular location, and what that location is.

Looking at documents in general (not as part of a search application), we could choose to analyze the document text, using entity recognition to find locations in the form of addresses or location names. This approach can work, but it's vulnerable to two kinds of errors. On one hand, we might fail to recognize locations for documents that don't contain salient location entities, e.g., a document about a mural or sculpture that doesn't include its address. On the other hand, a location entity in the text may be spurious, e.g., it may be part of an author's contact information or some other boilerplate text.

But we do have a search application, and we can take advantage of it. Specifically, we can use the locations of searchers engaging with a document to locate that document. When a document is strongly associated with a location, searchers engaging with the document tend to be tightly clustered around its associated location.

There are various ways to geolocate searchers, the simplest being to use their IP addresses. While geolocation can be noisy, it's generally simpler and more reliable than applying entity recognition to document text — especially if all we want to know is the centroid of the searchers' locations and an estimate of the variance to determine how tightly they're clustered.

Once we've associated documents with locations, we can use those associations to better classify which queries have local intent and to improve results for those queries.

## **Summary**

Location often serves as a valuable contextual information to help us better understand search queries. Sometimes, location serves an implicit part of the search query; other times, its influence on query understanding is more subtle. We often know the searcher's location, and we can associate documents with locations based on the locations of searchers that engage with those documents. Unlike in real estate, location isn't everything. But it provides valuable context to improve query understanding.

## **Previous: Session Context**

## **Next: Seasonality**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigg.

Oct 30, 2017 · 3 min read

## Seasonality

Whether it's the time of year or the time of day, the time when someone performs a search sometimes help us determine the searcher's intent. As with location, it's important to determine which queries are time-sensitive and then determine how temporal context can help us understanding them.

### Query Seasonality

Search behavior often exhibits seasonality. Certain queries are more frequent at particular times of year, i.e., their probability is time-sensitive. For example, many people search for flowers in the week leading up to Valentine's Day.

Sometimes, it's not the query probability but the intent that is time-sensitive. For example, someone searching for jackets on a clothing site in the summer probably has different intent than someone making the same search in winter. In other cases, the time of day matters: for example, a search for local restaurants at noon on a weekday often reflects a different intent than the same search on a Saturday evening.

When queries exhibit seasonality, temporal context helps us better estimate their probability — which is particularly useful for autocomplete. On a site that sells clothing, for example, a *c* in late October would be likely to complete to *costume*, while in mid-December it more likely completes to *christmas*.

### Determining Seasonality

How do we determine that a query is seasonal? For frequent queries, we can measure the variance of when queries occur over time and compare it to that of queries performed throughout the year. We can apply the same approach for queries that are sensitive to particular times of day, but for simplicity we'll focus on time of year.

The variance of a random variable with uniform distribution between the real values  $a$  and  $b$  is  $1/12 (b-a)^2$ . If we uses years as units, then a query occurring uniformly throughout the year has a variance of  $1/12$ .

In contrast, a query that occurs only on a single day of the year (uniformly throughout that day) has a variance of  $(1/365)^2 * (1/12)$  — over 100,000 times lower than that of a query occurring uniformly throughout the year. In general, the variance correlates negatively to seasonality.

### **Caveats**

A few caveats are in order about computing seasonality this way:

- It's important to normalize based on overall query frequency throughout the year, since overall query traffic may exhibit its own seasonality.
- The year ends where it begins, but our formula for variance treats December 31 as being a year from January 1. A heuristic to mostly avoid this problem is to compute variance two ways, first starting the year on January 1, and then starting the year on July 1, and using whichever yields to lower variance.
- This approach only works for frequent queries. For infrequent queries, the variance is essentially noise. We would need to use a more sophisticated approach to determine that a class of query exhibits seasonality, or a machine learning approach that treats the query probability as a regression problem.

Finally, a query may have seasonal intent even though the query itself isn't seasonal, as in the example of jackets for summer and winter. In those cases, we need to determine which documents exhibit seasonality (using the above variance approach for documents rather than queries), and then determine whether document engagement for a query correlates to time of year or day.

### **Summary**

Context often helps us understand queries. In the last three posts, we've addressed session context, location, and seasonality. Seasonality is especially useful for predicting query probability and thus improving autocomplete. But remember that context is a secondary signal — the searcher's query will always be the primary signal for establishing intent.

**Previous: Location as Context**

**Next: Personalization**





Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigggle.

Nov 6, 2017 · 3 min read

# Personalization

In 1964, information science pioneer William Goffman said:

*Relevance is defined as a measure of information conveyed by a document relative to a query...[but] the relationship between the document and the query, though necessary, is not sufficient to determine relevance.*

In other words, a search query isn't always sufficient to establish the searcher's intent and the relevance of results.

We've already discussed how context can influence query understanding. Now let's look at the ultimate context: the searchers themselves.

## What is Personalization?

Personalization is a broad term that extends beyond search. It refers to any attempt to customize a product or service to accommodate the particular needs of either an individual or a particular segment of the population.

In the context of query understanding, personalization means using information about the searcher to either personalize the estimation of query probabilities or supplement the query as a signal for query understanding.

The main sources for personalization are explicit input from the searcher, inferences from the searcher behavior, and inferences from population segments to which the searcher belongs.

## Personalized Query Probabilities

Knowing who the searcher is helps us predict what he or she will search for. Searchers often repeat queries —sometimes even within the same session — and hence many search engines include a search history feature that provides convenient access to recently performed searches. More generally, what we know about a searcher's interests allows us to personalize query probabilities.

As we saw in our discussion of autocomplete, we can use historical query logs to compute the probability of a query based on how frequently it appears in the log. This approach, however, doesn't take into account the

searcher's location or seasonality, let alone information about the individual searcher.

We can use machine learning to predict personalized query probabilities. Overall query frequency is one feature we would include in a machine learning model, but we would accomplish personalization through features like the following:

- Whether and how recently the searcher has performed the query.
- How the query relates to the searcher's explicitly stated interests.
- Whether and how frequently the searcher has performed similar queries.
- Whether and how frequently searchers in a population segment to which the searcher belongs have performed the query or similar queries.

For searchers with a rich search history or a detailed collection of explicitly stated interests, features based on the individual searcher are likely to have high predictive value. When we don't have much information from a searcher's history or explicitly stated interest, we rely more on the behavior of population segments to which the searcher belongs.

### **Personalization as an Intent Signal**

Sometimes our knowledge about the searcher helps clarify or refine the way we derive intent from the search query. We can use personalization for either query rewriting or result ranking.

For example, a searcher on a site that sells clothing is probably looking for clothes that he or she can wear. If we know— or can infer — the searcher's gender and measurements, then we can rewrite the query to filter or boost results matching those constraints.

A more subtle use of personalization on a clothing site would be to favor styles that the searcher has purchased before, or that are popular with segments the searcher belongs to. We could implement this approach through boosting, or by including searcher-specific features in a machine-learned ranking model. For broad queries that return large result sets, a change in ranking could lead to different searchers seeing completely different results from one another.

Using personalization as an intent signal requires care. Personalization should be a secondary signal that supplements the query: as with all of

context, it's never enough to overcome the query itself as the primary signal.

### **Summary**

As Goffman said, the query is necessary but not always sufficient to determine the searcher's intent and hence which results will be relevant. Personalization can help establish the searcher's intent, both during query formulation and query understanding. The query itself is still the primary input for understanding, but knowledge about the searcher provides valuable context.

**Previous: Seasonality**

**Next: Search as a Conversation**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigggle.

Nov 27, 2017 · 4 min read

## Search as a Conversation

Most search applications assume a query-response paradigm: the searcher submits a search query, and the search engine responds with results. The query-response paradigm works well for simple search needs that the search engine understands.

The query-response paradigm breaks down, however, when searchers have more complex needs, or when the search engine struggles with query understanding. In those cases, it's better to model search as a conversation.

### Communication vs. Relevance

When we evaluate the performance of search engines, we usually think in terms of relevance — measuring it as a combination of precision and recall. In the query-response paradigm, we optimize for relevance: we have only one chance to satisfy the searcher's needs, and we want to make the most of it.

When we model search as a conversation, however, we need to consider a bigger picture. A search session is a series of exchanges between the searcher and the search engine. An effective session is more than a sequence of isolated query-response pairs: it's a learning process for the searcher. The effectiveness of that learning process depends on the quality of communication between the searcher and the search engine.

This bigger picture means good news and bad news for search application developers.

The good news: a conversational approach takes some of the pressure off of the search engine. Query understanding is never perfect, and a conversational model recognizes this imperfection. A conversational model encourages searchers to clarify or refine their queries when the search engine fails to understand them. Moreover, it offers searchers the opportunity to evolve their own intent based on the results the search engine returns to them.

The bad news: an effective conversation requires more from the search engine. It's not enough for the search engine to make a best effort to return

relevant results. The search engine also has to provide transparency, control, and guidance to searchers:

- **Transparency**, to know why they're seeing the results the search engine returns to them — especially if those aren't the results they want.
- **Control**, to take over manually when the search engine fails to automatically understand their queries.
- **Guidance**, to navigate the otherwise overwhelming space of ways in which they could reformulate their queries.

## Transparency

The perfect search engine reads a searcher's mind. But no search engine is perfect, so it's important for a searcher to be able to recover from the search engine's mistakes. In order to do so, the searcher needs to know how the search engine understood — or misunderstood — the query. In other words, the search engine has to provide transparency.

Transparency is difficult to define. A search engine can provide a literal and exhaustive description of how it processed a search query, but the searcher is unlikely to read all of it, let alone understand it. What the searcher wants and needs from the search engine is a clear, concise explanation of how it processed the query.

At the very least, a search engine should expose any spelling correction, query expansion, query relaxation, query segmentation, or query scoping that might not be obvious to the searcher. The searcher won't care about the details when things go right; but when things go wrong, it's important for the searcher to be able to figure out why.

## Control

Transparency is necessary to let the searcher know what went wrong, but it isn't sufficient. The searcher also needs control, in order to fix the problem.

Control means that the searcher can do — or undo — the same things the search engine does automatically. Searchers should be able to use quotation marks to specify exact words (overriding stemming or any other query rewriting) or phrases, as well as Boolean operators like AND, OR, and NOT to specify their query with arbitrary precision.

Providing control is not an excuse for the search engine not to make its own best effort. When the searcher's intent is obvious, the search engine should

be able to automatically derive that intent from the query, without requiring the searcher's assistance.

## **Guidance**

Transparency and control are necessary for a conversational model, but they're not sufficient. The search engine also needs to offer guidance so that searchers can navigate the otherwise overwhelming space of ways they can formulate and reformulate their queries.

There are many ways that search engines can provide guidance to searchers. Autocomplete, when well-implemented, guides searchers to queries that the search engine understands. Spelling correction automatically or interactively (through "did you mean") guides searchers to the queries that the searchers intended. Faceted search provides guidance to searchers by offering them ways to narrow their queries. Search engines can also suggest related searches: variants of the searcher's query that the search engine hopefully understands.

No amount of guidance can address every possible communication breakdown between the searcher and the search engine. But something is far better than nothing, and guidance is often the difference between a successful course correction and a frustrating failure.

## **Summary**

Modeling search as a conversation is a powerful way to address complex search needs. The conversational model recognizes that search engines sometimes struggle with query understanding. It allows searchers to clarify or refine their queries, as well as to evolve their own intent based on results the search engine returns to them.

But a conversational model requires more from the search engine. The search engine has to provide transparency, control, and guidance to searchers. Indeed, building a conversational search engine is harder than just supporting the query-response paradigm. But it's worth the effort to address the wide range of searchers' needs.

## **Previous: Personalization**

## **Next: Clarification Dialogues**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigggle.

Dec 13, 2017 · 2 min read

## Clarification Dialogues

As we saw in the previous post, modeling search as a conversation makes it possible to overcome breakdowns in communication between the searcher and the search engine. An important conversational technique is for the search engine to provide a clarification dialogue when it infers that such a breakdown may have occurred.

### “Did You Mean” and Automatic Rewriting

Perhaps the most familiar clarification dialogue comes from spelling correction. When a search engine determines that a query has a significant probability of being misspelled, it shows results for the original query but proposes an alternative query as a “did you mean” suggestion.

Alternatively, it automatically rewrites the query, notifying the searcher with a prominent message above the search results. The interface typically includes a “search instead for” link that allows the searcher to override to automatic rewriting and see results for the original query.

This approach generalizes beyond spelling correction to other forms of query rewriting. Query rewriting could include query expansion, query relaxation, query segmentation, or query scoping. In all cases, the choice between a conservative “did you mean” and a more aggressive automatic rewriting with an opt-out should reflect the search engine’s confidence in its query interpretation.

### Optimizing for the Searcher’s Success

The goal of a clarification dialogue should be to optimize for the searcher’s success. If the search engine has high confidence in its interpretation of the search query, then a clarification dialogue not only doesn’t help, but can harm the searcher by distracting from relevant results. But when the search engine doesn’t have high confidence in its interpretation, a clarification dialogue risks that distraction in order to hedges the search engine’s bet.

Deciding whether to present a clarification dialogue is essentially an optimization problem, with the goal of maximizing the expected utility for the searcher. Doing so requires three things:

- Making the search engine's confidence in its primary interpretation explicit, preferably modeling it as a probability.
- Doing the same for any alternative interpretation that might be presented as a "did you mean".
- Determining the expected harm caused by the insertion of a clarification dialogue, i.e., the cost of displacing potentially relevant results.

For this last computation, we can use log analysis or experimentation to determine the effect of position on engagement with relevant results. That can help us determine whether and where to insert a clarification dialogue.

### **Summary**

Breakdowns in communication between the searcher and the search engine are unavoidable. Clarification dialogues offer at least a partial remedy: the opportunity for the search engine to hedge its bets. Using them effectively requires estimating the confidence of the search engine's primary and alternative interpretation, as well as determining the harm from displacing relevant results. Clarification dialogues are a baby step towards conversational search, and towards recognizing that query understanding is never perfect.

**Previous: Search as a Conversation**

**Next: Relevance Feedback**





Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigggle.

Jan 3 · 4 min read

# Relevance Feedback

The idea of modeling search as a conversation has been around for decades. One of the oldest ideas in information retrieval is relevance feedback, which dates back to the 1960s. Relevance feedback allows searchers to tell the search engine which results are and aren't relevant, guiding the search engine better understand the query and thus improve the results.

## Feedback Mechanisms

The simplest relevance feedback mechanisms involve direct, explicit feedback applied to the search results themselves. The search engine accompanies each search result with buttons (or other interface elements) that allow the searcher to indicate whether or not that result was relevant. Alternatively, the search engine can provide a single feedback mechanism for the entire page.

It's also possible for the search engine to collect implicit feedback about the results, based on the searcher's behavior. When the searcher engages with a search result (e.g., by clicking on it), the search engine treats the engagement as implicit positive feedback. Conversely, when the searcher doesn't engage with a search result — either by clicking on a lower-ranked result or by not clicking on any results — the search engine treats the lack of engagement as implicit negative feedback. This implicit feedback, while difficult to incorporate directly into the search experience, is useful for training machine-learned ranking models.

Finally, searchers can provide feedback at the level of terms (i.e., words or phrases) rather than documents. Term feedback can improve recall by guiding query expansion, or it can improve precision by disambiguating the query. For example, if someone searches for *tulip bulbs* in order to find information about the historical tulip mania, he or she could improve the search by providing positive feedback for the term *bubble* and negative feedback for the term *horticulture*. Term feedback is more abstract and complicated than result feedback, but it provides a clearer intent signal.

In between terms and documents, searchers can provide feedback at the level of passages, i.e., portions of documents. Passage-level feedback is common in the legal domain.

Finally, there's a technique called pseudo relevance feedback (or blind feedback): it simulates relevance feedback by reinforcing the top-ranked results as if the searcher had provided positive relevance feedback for them.

### **Leveraging Feedback**

After the searcher provides feedback for the search results, the search engine uses this feedback to modify the search query and (hopefully) return more relevant results. If the search engine maps the query to a vector space model, it maps the relevance feedback to the same vector space and rewrites the query as a combination of the two vectors. The goal is to adjust the query towards the results, terms, or passages that the searcher marked as relevant and away from those marked as irrelevant.

The best-known relevance feedback technique is the Rocchio algorithm, which was developed in the 1960s. Lucene's `MoreLikeThis` class is a variant of the Rocchio algorithm. More recently developed approaches tend to be probabilistic and rely on statistical language models.

### **Challenges**

Despite its promise, relevance feedback hasn't seen significant adoption outside of the legal domain and academic research settings — other than the use of implicit feedback from clicks to train ranking models. Its lack of mass adoption reflects three challenges with applying relevance feedback.

The first challenge is that relevance feedback mechanisms struggle when the set of relevant results is not homogenous — particularly if there are two or more distinct clusters of relevant results. For example, the relevant results for a search might include a mix of long-form documents and short-form messages. The more heterogeneous the relevant results, the greater the risk that relevance feedback will favor some segments of the relevant results at the expense of others. This feedback can misdirect the search engine, ultimately degrading the overall relevance of the results.

The second challenge is that explicit relevance feedback requires searchers to expend effort on top of the search itself. Most people avoid efforts they feel are unnecessary. Searchers who find relevant results probably won't be motivated to perform the additional action of explicitly providing positive feedback to the search engine. Conversely, searchers who receive irrelevant results won't necessarily want to actively help the search engine; instead, they are likely to try a different query — or a different search engine.

The third challenge is familiarity bias. Since few search engines outside of academic research settings employ relevance feedback, most developers and

product managers are wary of presenting searchers with unfamiliar interfaces and risking a negative impact on searcher engagement.

### **Summary**

Relevance feedback is one of the oldest ways to model search as a conversation. By telling the search engine which results are and aren't relevant, searchers can help the search engine help them. But its lack of mass adoption suggests that we will need to overcome the challenges of result diversity, searcher motivation, and familiarity bias in order to make effective use of relevance feedback to improve query understanding.

**Previous: Clarification Dialogues**

**Next: Faceted Search**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twiggie.

Jan 24 · 5 min read

# Faceted Search

Faceted search is a topic broad enough to deserve its own book. It has become a standard feature of all modern search engines, including open-source platforms like Solr and Elastic.

In this post, I'll quickly explain how faceted classification and faceted search work. I'll then outline how faceted search interacts with some of the query understanding approaches discussed in previous posts.

## Faceted Classification

Faceted search starts with faceted classification. Faceted classification uses a collection of independent attributes, called facets, to classify each entry in the searchable collection. In contrast with a taxonomy that uses a single hierarchical classification scheme, faceted classification doesn't impose a rigid ordering of attributes on the searcher.

## A Single Taxonomy is Inflexible

The best way to understand faceted classification is to see it in contrast with the limitations of using a single taxonomy when it comes to representing independent attributes. Let's look at an example.

Consider a clothing site with products organized using a single taxonomy. Should the top level of the taxonomy correspond to gender (*Men's*, *Women's*, etc.) or to product type (*Shirts*, *Pants*, etc.)? Or to some other attribute?

If gender is the top level, then *Men's* will have *Men's Shirts* and *Men's Pants* as children nodes in the taxonomy, while *Women's* will have *Women's Shirts* and *Women's Pants* as its children. If product type is the top level, then *Shirts* will have *Men's Shirts* and *Women's Shirts* as its children, etc.

This choice, which is somewhat arbitrary, has important consequences for searchers. Searchers who browse the product taxonomy will be limited to the taxonomy's order. If gender is the top level, then the taxonomy won't have a node corresponding to all shirts. Conversely, if product type is the top level, then it won't have a node corresponding to all men's clothing.

If we look beyond this trivial example with just two attributes, we can see that the number of these choices for a single taxonomy grows exponentially (actually, factorially) with the number of independent attributes. While some orderings are more natural than others, any fixed ordering imposes rigidity.

### **The Flexibility of Facets**

A more flexible approach is to have two distinct facets for *Gender* and *Product Type*. There is not hierarchical relationship between the two facets: each men's shirt is assigned the facet values *Gender: Men's* and *Product Type: Shirts*. Rather than imposing an order on the attributes, faceted classification represents their independence explicitly by modeling each independent attribute as a first-class facet.

There can still be hierarchical relationships within a facet; for example, in the *Product Type* facet, *Shirts* can have child values like *T-Shirts* and *Dress Shirts*. But these are true hierarchical relationships, rather than intersections of independent attributes.

The main benefit of faceted classification is that it allows searchers to traverse the facets in any order they choose. In a faceted classification, it's possible for searchers to retrieve all products with *Product Type: Shirts* or all products with *Gender: Men's*. Faceted classification removes the limitations that a single taxonomy imposes by ordering the attributes.

### **Faceted Search**

Faceted search takes advantage of faceted classification to support query refinement. For example, someone who searches for *polo* can narrow the search results by selecting *Gender: Men's* and *Product Type: Shirts*. Combining an initial free-text search with faceted refinement allows the searcher to express a highly specific intent.

In particular, faceted refinement is most useful when the initial search query returns a large result set, often because it is a short query with low specificity. One use of faceted refinement is to clarify ambiguous search queries, e.g. *matrix* -> *Genre: Science Fiction* vs. *matrix* -> *Product Type: Textbooks*. But the best use of faceted search is for queries that are unambiguous but broad.

For example, a search for *shirts* on a clothing site may return thousands of results, overwhelming the searcher. In response to such a query, the search engine presents a set of facets (gender, style, brand, color, etc.) and associated values that organize the results into multidimensional space that the searcher can navigate.

This brief discussion of faceted search glosses over its complexities. In particular, faceted search creates design challenges when there are a large number of facets, or when a facet has a large number of values. For more discussion of these and other issues, I recommend a book that offers a fuller treatment of the subject.

## Query Scoping

Faceted search interacts naturally with query rewriting — particularly query scoping. Indeed, much of query rewriting is an attempt to automate the faceted refinement process by inferring facet values from search queries.

When query segments obtained from query segmentation match facet values, the search engine can rewrite the query by substituting facet values for the corresponding segments. For example, a search for *stretch leather pants* becomes the single-word search *stretch*, refined by the two facet values *Material: Leather* and *Product Type: Pants*. This rewriting is just query scoping, taking advantage of the faceted classification.

In this example, the matches to facet values are exact. But, as we've seen in previous posts, we can take advantage of stemming, spelling correction, and query expansion to match facet values more aggressively.

## Autocomplete

Another place that facets are useful for query understanding is autocomplete. Facet values tend to be great autocomplete suggestions, since they hopefully represent a curated collection of unambiguous search intents.

Facet values are good candidates for autocomplete when there isn't enough data about query popularity and performance to determine autocomplete suggestions from historical search behavior. They're particularly useful for new search applications, as well as applications that are unlikely to even collect a volume of traffic.

## Related Search Suggestions

Search queries composed entirely of facet value selections tend to be more reliable than queries composed of arbitrary keywords. For example, a search for the facet value *Product Type: Suits* avoids precision problems (e.g., bathing suits and body suits that match the keyword *suit*, as well as recall problems (e.g.,, tuxedos that don't match the keyword *suit*).

As discussed, autocomplete provides an opportunity to guide users to queries composed of facet values. But another opportunity for guidance is to

present search suggestions along with the search results. For example, a search for *two piece swimsuit* can suggest *Product Type: Bikinis* as a related search.

Related search suggestions can be based on query similarity, content similarity, and historical query reformulation behavior. For an example of a related search suggestion system, see this paper.

## **Summary**

Faceted search has become a standard feature of modern search engines. Faceted classification offers more flexibility than a single taxonomy, and faceted refinement allows searchers to clarify and refine queries with large result sets. Faceted search also plays well with query scoping, autocomplete, and related search suggestions. Faceted search plays well with query understanding, and it's a good idea to consider their interaction in designing a search experience.

**Previous: Relevance Feedback**

**Next: Search Results Presentation**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twiggie.

Feb 20 · 2 min read

## Search Results Presentation

Until now, we've mostly focused on query processing — which is to be expected, given that this series is about query understanding. But given that searchers are ultimately interested in the results, it's also important to consider how the search engine presents those results.

How the search engine presents results matters for two reasons.

The first reason is that an effective presentation allows searchers to quickly determine whether a particular search result is relevant to their needs. Search result snippets, also known as search result summaries, will be the subject of the next post. These serve the critical function of signalling relevance at a per-result level. No retrieval strategy can achieve perfect precision, but good search result snippets reduce the need for searchers to click through to a result to determine whether it is relevant to their intent.

The second reason is that the presentation of results should reflect the nature of the query and, when the search engine can infer it, the underlying information-seeking task. For example, a search intended to retrieve a specific result calls for a different presentation style than a query intended to explore a broad collection of results. In addition, the types of objects in the search results — e.g., books vs. shoes — often suggest a particular visual presentation.

The search engine shouldn't just return the right results — it should present them in a way that communicates those results effectively and efficiently. We'll use the next couple of posts to discuss various ways to do so.

**Previous: Faceted Search**

**Next: Search Result Snippets**





Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twiggie.

Mar 12 · 3 min read

## Search Result Snippets

Search result snippets, also known as query-biased summaries, are the additional context included with each result on the search results page. They are an essential tool to help searchers find what they're looking for.

Search result snippets can serve various purposes. Sometimes snippets allow searchers to satisfy their information needs directly from the search results page, such as when a sentence answering the searcher's question appears in the snippet. In general, snippets allow search engines to establish trust and comfort by showing searchers how each result matches the query.

But the most important function that search result snippets serve is helping searchers quickly determine which results are most likely to satisfy their information needs.

### Snippets as Relevance Signals

No retrieval strategy can achieve perfect precision — every strategy represents a trade-off between precision and recall. As a result, searchers often need to scan the set of search results in order to determine which results are relevant to their needs.

An important measure of the effectiveness of snippets — and of search results presentation generally — is how often searchers bounce back from clicks on search results back to the search results page. Unless the searcher is performing an exploratory search with the goal of collecting multiple relevant results, a high bounce rate tells us that the search results page isn't providing a strong enough relevance signal for each result.

### Query-Independent Summaries

In most search applications, the search results page shows a short, query-independent summary for each result entry. A typical summary includes a title, an associated image, and application-appropriate metadata. For example, a document search application might include the document's type, size, and creation date; while an ecommerce search application might include the brand, price, and average review score. Query-independent summaries, which can be computed as part of the indexing process, should

be rich enough to communicate salient information about the results while being concise enough to allow the searcher to scan through them quickly.

It's a good idea to optimize query-independent summaries through experimentation, using the bounce rate as a success metric. For example, searchers on an ecommerce site might change their minds when they discover how much it will cost in time or money to ship a product. In a document search application, the citation count might be a signal that searchers use to determine whether a document is a credible source. Experimentation is the best way to determine which pieces of information matter most to searchers.

### **Query-Dependent Summaries**

Query-independent summaries, while useful, are not always sufficient to communicate how search results relate to the searcher's intent. In particular, the query may match a portion of a search result that is not included in a query-independent summary.

A query-dependent or query-based summary focuses on the part of the search result that most relates to the query. At the very least, it shows which field of the document matches the search results. Typically, it presents a single line of document text that contains all or most of the matched query terms, with the query terms bolded or otherwise highlighted to make them salient. When the matched keywords are the result of stemming, lemmatization, or other query expansion, this highlighting makes the query expansion more transparent.

There are many algorithms for generating query-dependent summaries, and most search engines include them as a product feature. For example, Lucene includes a highlight package that handles the various aspects of generating and presenting query-dependent summaries. Specifically, it includes methods to find a span of text in each result containing the query keywords.

### **Summary**

Much as we try to improve query understanding, result retrieval, and result ranking, a search engine will never achieve perfect precision. By showing searchers how each result matches the query, search result snippets establish trust and help searchers determine which results are most likely to satisfy their information needs—and sometimes the snippets are sufficient to satisfy those needs. It's a reminder the search isn't just about retrieving relevant results; it's about communicating effectively with the searcher and efficiently satisfying the searcher's information needs.

**Previous: Search Results Presentation**

**Next: Search Results Clustering**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twiggie.

Apr 16 · 3 min read

## Search Results Clustering

Search queries that express broad intent often return intractably large result sets. We've already discussed faceted search as a way to help users narrow their intent by refining the initial query. Another way to address queries with broad intent is to cluster the search results.

Search results clustering is an idea that goes back at least to the Scatter/Gather research in the early 1990s. "Scatter/Gather as a Tool for the Navigation of Retrieval Results" describes a technique that clusters search results into semantically coherent groups on-the-fly and presents descriptive summaries of the groups to the searcher. The clustering allows searcher to identify useful subset of the results, when can in turn be clustered to identify narrower subsets.

Since then, there have been thousands of papers on search results clustering, as well as a variety of commercial and open-source efforts to apply clustering to web and enterprise search engines.

### Competing Objectives

A clustering of search results should satisfy three objectives:

- **Coherence.** Search results assigned to the same cluster should be substantially similar to one another, so that the cluster represents a coherent subset of possible search intents.
- **Distinctiveness.** Search results assigned to different clusters should be substantially different from one another, so that each cluster represents a distinct subset of possible search intents.
- **Clarity.** The meaning of each cluster should be clear to the searcher, whether from labels, descriptive summarizations (e.g., a tag clouds), or representative results.

The first two objectives of coherence and distinctiveness compete with one another. Reducing cluster size (e.g., by splitting clusters) generally improves coherence, but at the expense of distinctiveness. Conversely, increasing cluster size (e.g., by merging clusters) improves distinctiveness at the

expense of coherence. Clustering algorithms manage a trade-off between these two competing objectives.

## **Challenges**

Search results clustering is an idea that sounds great in theory, but it's surprisingly difficult to implement clustering well in practice. The main challenges are defining the document similarity function, tuning the clustering algorithm, and descriptively summarizing the clusters.

It's relatively easy to determine when two search results are extremely similar. Indeed, many search engines include a post-ranking step to remove duplicate and near-duplicate results. However, determining that two results are similar is a much thornier problem than determining that they are near-duplicates. It's difficult to choose an appropriate similarity threshold, or even to define a similarity function, that consistently yields coherent, distinct clusters.

Typical clustering approaches involve embedding documents into a vectors and then computing a geometric function on them, such as cosine, to measuring their similarity. While such approaches rely on elegant theoretical models, the results are hit and miss, highly subject to the particulars of the documents. For example, boilerplate text and markup language can affect the similarity function, even though they have no bearing on the substance of the documents. In practice, tuning a similarity function requires a significant amount of data cleaning and can easily become a rathole.

In addition, the results often distribute non-uniformly into distinct search intents. Ideally, the clustering algorithm should adapt to the distribution, producing clusters of suitably varying size. In practice, the most represented intents tend to be split into overlapping clusters, lowering distinctiveness, while the least represented intents tend merged, lowering coherence.

Finally, it's difficult to produce clear descriptive summaries. Algorithmically produced labels often fail to capture and convey meaning, especially if they're produced at query time from salient words and phrases in the results. And even if the clustering algorithm does a good job of selecting representative results, they're only effective as summaries if the searcher can both understand them (cf. search result snippets) and extrapolate from them. In general, producing and communicating intelligible groupings is a triumph of human intelligence that AI still struggles with.

## **Summary**

Search results clustering is an old idea for presenting and organizing result sets, and researchers have continued to work on refining it. It sounds great in theory but has proven to be difficult in practice. Nonetheless, clustering can be a valuable tool to address broad-intent queries.

**Previous: Search Result Snippets**

**Next: Question Answering**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigggle.

May 1 · 4 min read

# Question Answering

When we talk about search, we have traditionally thought in terms of search results being documents or products. Searchers, however, increasingly think of search engines as question answering engines.

Many searchers expect to be able to express questions in natural language (e.g., *What is the answer to life, the universe, and everything?*) and obtain concise, relevant answers (in this case, 42). Moreover, the emergence of intelligent assistants like Siri, Alexa, and Google Assistant is training the next generation of searchers to ask direct questions and expect direct answers.

## Early QA Systems

Question answering (QA) systems date back to the 1960s. Early QA systems focused on narrow, closed domains. Two notable examples are BASEBALL, which answered questions about a single season of American League baseball games, and LUNAR, which answered questions about the analysis of rock samples from the Apollo moon missions. These systems parsed natural-language queries and translated them into database queries, which they executed against custom-built knowledge bases. They worked reasonably well, as long as the queries conformed to their narrow scope of knowledge.

In the 1980s and 1990s, researchers shifted their attention to more general-purpose, open-domain QA systems. Moving away from knowledge bases, they embraced an information retrieval approach (such as this one) that was less domain-dependent. They treated each question as a search query, retrieved a set of relevant documents, extracted candidate answers from the results, and then presented the best candidate answer to the searcher. The emergence of open-domain QA systems inspired the Text Retrieval Conference (TREC) to establish a question-answering track, which has been running since 1999.

## Modern QA Systems

The emergence of the web led to a large-scale digitization of knowledge, swinging the pendulum back to QA systems built on knowledge bases. Resources like Wikipedia became critical building blocks for these systems,

the most famous being the Watson system that IBM researchers built to defeat top *Jeopardy!* champions in 2011. That system mined 200 million pages to create a knowledge base, including a full crawl of Wikipedia.

At the same time, we started to see open-domain QA systems available to the general public. In 2009, Wolfram Alpha launched an “answer engine” based on a collection of curated content, and Siri integrated with it when it launched in 2011. Finally, in 2012, Google embraced QA by launching its Knowledge Graph, leveraging the Freebase knowledge base from its acquisition of Metaweb.

Unlike previous open-domain QA systems that relied on information retrieval to extract answers from unstructured content, modern QA systems build knowledge bases by extracting a rich ontology of entities and relationships from a combination of structured and unstructured content. They take advantage of the latest developments in machine learning, representing text with word embeddings and character embeddings, and using deep learning — specifically sequence learning methods like LSTM.

And most recently, “smart speakers” like the Amazon Echo and Google Home, are bringing voice-based QA systems into millions of homes.

## **Challenges**

The foundation of a QA system is its knowledge base. Given the current state of the art, a knowledge base can be broad or deep but not both. Google’s Knowledge Graph optimizes for breadth, while domain-specific knowledge bases like Twgggle’s consumer product ontology optimize for depth. Decisions about where to emphasize breadth vs depth are critical trade-offs in the design of a QA system.

But the biggest challenges in designing QA systems come from their interface constraints.

Questions are natural-language queries, whether they are submitted through a keyboard or a microphone. As a result, the input interface can’t effectively use techniques like autocomplete to guide the searcher. Any feedback to the searcher has to wait until after the searcher has submitted the query.

Unfortunately, there are many opportunities for the system to misunderstand the searcher: spelling mistakes, voice recognition errors, and a variety of natural language processing errors. No system is perfect, but the error rate has to be low enough that searchers don’t simply give up in frustration.

The output interface for a QA system is even more constrained. Returning a single answer to the searcher — especially if it is presented as voice output —



leaves almost no room for error. Such an interface is much less forgiving than a ranked list of search results that the searcher can scan. It forces a trade-off between accuracy and coverage: a QA system has to decide at what confidence threshold to present an answer, versus admitting that it doesn't know. Rejecting too many questions frustrates searchers, but wrong answers quickly erodes trust. Ideally the interface would be conversational, but none of today's QA systems support meaningful conversation.

### **Hybrid Approach**

Given all these challenges, it's not surprising that we see hybrid approaches that combine traditional search engines with QA systems. When the QA system has high confidence, it returns an answer; otherwise, the system falls back to performing a search and returning a ranked list of results. This approach works reasonably well when the output is presented on a screen. For voice output, however, there isn't a good analog to a result set.

A hybrid approach also makes it easier to develop a QA system incrementally. The initial coverage of the QA system might be a narrow set of frequent queries, or queries that conform to easily recognized patterns. Given how much more challenging it is to develop a QA system than a search engine, such an incremental approach makes it possible to prioritize QA efforts to focus on where they will deliver the highest return.

### **Summary**

QA systems represent a logical evolution of search engines, catering to a new generation of searchers who expect to be able to express questions in natural language and obtain answers rather than search results. QA systems are becoming mainstream, but they still face many challenges. If the knowledge base is broad, it's unlikely to be deep — and vice versa. Moreover, the interface constraints make it difficult to manage searcher expectations, and no one has developed QA systems that meaningfully support conversational interaction. Where possible — specifically, when there's a display for the results — the best approach is generally a hybrid that combines QA with traditional search.

### **Previous: Search Results Clustering**

### **Next: Query Understanding and Voice Interfaces**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twigg.

May 29 · 2 min read

# Query Understanding and Voice Interfaces

On the surface, using a voice interface for search doesn't seem that different from typing text into a search box.

**Converting speech to text and vice versa is a mostly solved problem.**

Speech recognition, while still not perfect, has matured to the point that it is ubiquitous. Application developers can leverage the major cloud providers — namely, Amazon, Microsoft, and Google —which provide reasonably priced APIs to use their speech-to-text services.

Meanwhile, computers have been able to talk to us for decades, and the quality of synthesized speech today is far more natural than when the Software Automatic Mouth (SAM) speech synthesizer was released for personal computers in 1982. Indeed recent developments like Google Duplex have raised concerns that people will be unable to distinguish human speakers from AIs.

So, while there's still ample opportunity to improve on both speech recognition and speech synthesis, both are good enough for everyday use.

**But there's a big gap between recognizing speech and understanding it.**

At best, speech recognition reduces the problem of query understanding with voice to the problem of query understanding. But the state of query understanding is far less mature than that of speech recognition. Indeed the ability of computers to recognize speech but not understand it can be particularly frustrating searchers who don't distinguish the two problems.

**And the biggest challenges come from interface constraints.**

On one hand, voice input interfaces lack autocomplete, spelling correction, or any other mechanisms to guide searchers as they construct queries. Hence, there's a greater chance that the searcher make queries that the search engine is unable to understand.

On the other hand, today's voice output interfaces don't allow the searcher to easily scan a set of search results, let alone conversational mechanisms to

refine queries, such as clarification dialogues and faceted search. That not only makes voice interfaces less resilient to misunderstanding, but also limits the scope of queries that they can handle. In particular, voice interfaces are not well suited to exploratory search.

**We'll get there — someday.**

We're just entering an age of voice interfaces for mainstream consumer applications. It will take some time to work out the kinks, improve query understanding, and figure out the design of conversational interfaces. But we'll get there — at least once we start to recognize these challenges and face them head-on.

**Previous: Question Answering**

**Next: Query Understanding and Chatbots**



Daniel Tunkelang

[Follow](#)

High-Class Consultant. Chief Search Evangelist at Twiggie.

Jun 14 · 3 min read

# Query Understanding and Chatbots

In this series, I've focused on the context of traditional search applications, which are the main beneficiaries of better query understanding. But given the rapid emergence of chatbots as alternatives to traditional interfaces, I'd like to briefly discuss how chatbots depend on and incorporate query understanding.

## Searching with Chatbots

A chatbot is a program designed to engage people in a natural-language dialogue, either through typed text or voice. In general, chatbots provide both more and less than a natural-language search interface: more in that chatbots are expected to engage users in extended conversations (not just search), but less in that they usually support only a small set of tasks.

Using a chatbot for search typically starts with a natural-language search query, e.g., "Shop for medium men's t-shirts" or "Find sushi near me." After parsing the query to extract the search query from the request, the chatbot responds by showing — or speaking descriptions of — the top results. If the query is highly specific and if the chatbot succeeds in understanding it, then this short result list should be sufficient to address the searcher's needs.

## Search as a Conversation

In other cases, it's important for the chatbot to engage searchers in further conversation to establish and address their needs.

If the chatbot is uncertain as to the meaning of the request — whether because of challenges in natural-language understanding or speech recognition — it can offer the user a clarification dialogue, e.g., responding to "Find me a hotel in Dublin" with "Did you mean Dublin, California or Dublin, Ireland?" Such a dialogue is particularly useful for disambiguating named entities, such as people, places, and titles of media like songs and movies.

If the request is clear but overly broad, e.g., "Shop for jeans," then the chatbot can suggest useful follow-up questions, such as "Shop for men's jeans." These follow-up questions are often a subset of those that would

offered by faceted search, but they are subject to the constraints of a minimal interface that can at most accommodate a handful of suggestions.

### **Compartmentalizing Complexity**

While a chatbot interface may seem simple on the surface, it takes a lot of hard work — and luck — to manage the complexity of an open-ended conversation. A request for a canned response like “Tell me a joke” shouldn’t be treated as a search query. Meanwhile, a request for a song like “Close to Me by The Cure” requires a different parsing strategy than “What’s the cure for hiccups?”

In order for a chatbot to fully parse and respond to a request, it helps to first route the request to the right domain (e.g., songs, medical advice, canned responses). Doing so is a classification problem that lends itself well to supervised machine learning, given a reasonably sized collection of requests labeled with the correct domains. This classification task gets harder as the number of domains increase, or as the individual domains increase in scope.

Another challenge is for a chatbot to determine whether the searcher is following up on a previous request or initiating a new one. A chatbot needs to maintain sufficient session context to make this determination, and even so it’s likely to get it wrong. The best approach is not to have to guess, but instead to provide follow-up suggestions that serve the most frequent user needs.

Compartmentalization helps reduce the overall problem of conversation to a more manageable set of subproblems. But this simplification comes at a cost. It requires an exhaustive enumeration of the set of domains, as well as care to keep the domains coherent and distinct from one another. It also requires building a parser for each of domain, which can be a lot of work. Still, it’s a more viable approach than building a universal chatbot that can pass the Turing Test without first reducing the problem space.

### **Summary**

Chatbots are emerging as human-computer interfaces, and they’re increasingly being used to support search tasks. Query understanding is a critical subproblem that search-oriented chatbots must address, relying on conversational techniques like clarification dialogues and faceted search to address misunderstandings and broad queries. But a chatbot’s interface constraints require that these techniques be used sparingly and effectively. Finally, chatbot designers can manage complexity by compartmentalizing their scope into a discrete collection of categories, using a classifier to route each request to an appropriate parser.