Semantic Search for the Preaching and Worship Portal

Harry Plantinga

April 24, 2014

Introduction

This document describes the architecture for a new search engine that will be used by the Preaching and Worship Portal (PWP) project. It will also serve as a replacement for the search engine used by the Christian Classics Ethereal Library (CCEL), and it may be used by other websites that are a part of the CCIW's preaching initiative.

The goal is that the PWP search engine should be significantly better than the basic search offerings of most competing websites, but still relatively straightforward to implement. We want to be able to go live with an initial version after the first summer.

The main distinctives of the search engine beyond a basic ranked full-text search will be that it supports

- Identifying possible matches to entities in an ontology of concepts, people, places, events, books, etc.
- Auto-completion showing disambiguated entity matches and normalized search string matches
- Identification and intelligent handling of scripture references
- Spelling correction and stemming
- Prioritizing result lists according to likelihood that they will be clicked, with global, cohort, and per-user learning based on search history
- Showing immediate results in a sidebar if we are confident we know what the user is searching for

The open-source Sphinx full-text search engine has the capabilities we need, uses a familiar SQL-like query language, and is already being used effectively for the Hymnary.org search engine. We will build a new search server using Sphinx that can respond to search queries from PWP, CCEL, and partner preaching sites.

Types of Queries

In preparation for developing the Preaching and Worship Portal, we performed a survey of hundreds of preachers to learn about the kinds of resources they use and how they locate them. One question on the survey asked for respondents to give sample queries they had recently used:

Give some examples of recent search terms you have used in your sermon preparation process.

A study of the queries suggests several common approaches to search:

1. Topical searches. Topics included prayer, justice, persistence, gratitude, trust, suffering, anger, grief, justification, infant baptism, ethnicity, busyness, peace, justice, righteous, depression and faith, money, sexuality, power/politics, technology, war and violence response, labor, stewardship, employment, spiritual discernment, Pope Francis, loneliness, obedience, sacrifice, church, archangels, thanksgiving, seven churches, jacob's ladder, mt moriah, and noah's ark.
2. Bible passages. Many of the queries centered on entering a bible passage in order to find study resources—commentaries, exegesis, study notes, Strong's, Greek word studies, theological word studies, etc. These were searched by entering the passage, and possibly a resource name. A topic, desired result type, or source website may be added to the query, e.g.
    a. "Illustrations for [text]"
    b. "Sermons on [text]"
    c. "John 3 atonement"
3. Holidays, seasons, and occasions, lectionary weeks, e.g. reformation Sunday, childbirth, thanksgiving, sanctity of human life Sunday.
4. Liturgical elements for a worship service, "call to worship", "prayers of confession", "prayer of approach", a prayer for humility

Thus, the subject of the search was most frequently identified by a theological topic, person, place, or other entity. We will create an ontology of such entities, their types and relationships, and the search terms used to find them.

Bible passages were another key way by which the subject of the search was identified. Other methods of identifying the subject of the search included holidays, seasons, occasions, and lectionary weeks.

The searches often also specified the desired type of resource. Users may be looking for

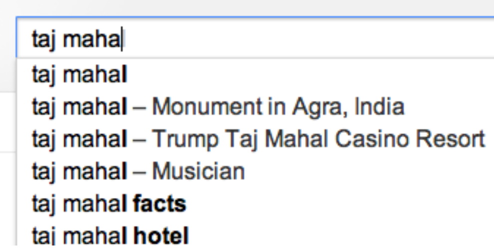
- Sermon preparation resources such as illustrations, sermons, blog posts
- Photos, paintings, movie clips (e.g. "image of hyssop")
- Quotes, books
- Bible passages relevant to a topic
- Background information such as maps, definitions, cultural references
- Liturgical elements

CCEL searches fit into the same framework, although the entities searched for might be slightly different. CCEL users also frequently type in book or author names, which could be handled as part of the ontology as well. CCEL users also sometimes typed in navigation terms like "forums," "free downloads," "mp3 files," etc. At the CCEL in 2012, about 44% of searches were for a book title or author name, 26% for an entity of the type described above, 22% for a bible passage, and the remainder navigation terms, full text searches, and the like.

Some queries were ambiguous. Users searching for "John" could intend a book of the bible, the apostle, or the first name of an author. We will store search terms that are commonly used for each entity; we'll derive probabilities of the entity intended through analysis of search result data. By analysis of search terms that lead to entities and the pages commonly opened as a result, we should be able to offer likely search results for many queries. Together with handling of scripture passages and navigation terms, we should be able to offer likely results for about half of queries that can be sorted in with the full-text search results according to the probability that each result will be clicked.

Semantic Search

Semantic search attempts to understand user queries and answer them directly, as opposed to full-text search, which finds instances of the search string in documents. So, for example, a search for "commentaries on Matthew 5:8" should go straight to commentaries on that verse, not to a page of full-text hits of documents containing the string "commentaries on Matthew 5:8".



Semantic search engines also need a way to disambiguate queries. For an ambiguous query like "John," the search engine could return results for the book of the bible, along with a link saying "or did you mean John the Apostle". However, it would save a page load if users were able to disambiguate while typing the query. Google does this in the autocomplete box, as illustrated in the image to the right. In that image, there are two types of links: disambiguated strings (e.g. "taj majal – Monument in Agra, India") and common queries starting with the letters typed (e.g. "taj mahal facts").

When it becomes likely that a user is searching for a particular entity, we can immediately show information on that entity in a sidebar. So, for example, when a CCEL user types "au" it is likely that the user is typing "augustine" and wants to see a list of books by that author. When the probability that the string typed so far refers to a particular entity reaches a certain threshold, e.g. 80%, we could display a sidebar or box with information about the entity—in this case, information about Augustine along with a list of his most popular books.

Search results should be shown with enough context that users can make an informed decision of which ones to click. Result should have contextual information organized into one- to two-inch rows something like Google's search results. The results should include title, type of resource, author/source, entity keywords, and an excerpt or description of about 24 (or 20-30) words. We could consider adding denomination and a popularity rating. Google adds a down arrow beside certain elements in the result entry. Clicking the down arrow shows a little more information. For example, clicking a down arrow beside an author name might show a popup with the author's name, picture, and a biographical paragraph about the author.

Entities and Keywords

The majority of the searches that our survey respondents reported using were for topical keywords, so this is perhaps the most important means by which users search for resources. Unfortunately, most websites with preaching and worship resources don't provide an effective means of finding resources by topic, so this is a key area in which the Preaching and Worship Portal can contribute.

In order to handle topics intelligently, we will create an ontology of theological concepts and their relations. This ontology will contain a map of theological terms and relationships between them. For example, the concept "compassion" might have sub-concepts "compassion of Christ," "compassion of God," and "compassion of Christians for others." It may have relationships to other concepts, such as being related to "love." Each concept will have a number of terms that are used to search for the concept. The ontology will also have several other types of entities: books, authors, people, places, books of the Bible, events, etc. We will call all of these "named entities." We will want to have named entities for all the sorts of things that people search for.

One important way that named entities will be used is to assist search. Preaching resources will be tagged with entities from a select list that they are strongly related to. The idea is that if a preacher looking for resources for a sermon on "compassion" should be shown a given resource, then that resource should be tagged with the "compassion" entity. Initially, this tagging will take place manually, but as the site grows in popularity, search data will help us make these connections.

Users intending an entity such as "compassion" may type a search string that does not match the name of the entity. For example, "pity" may be used as a synonym for compassion. We will need a keyword system for mapping search terms probabilistically to named entities. We may solve this in part in the construction of the ontology, by adding search terms for entities. However, we will also want to use search data to derive these connection probabilities.

We will also want to be able to display a Web page and other views for each entity. For a popular search term such as "compassion" we may have a full treatment, created by experts, with helpful resources for a sermon on compassion. This may include sermon ideas, book references, citations to the most important classical literature, hymns, images, videos, etc. We may display this on a separate Web page for the entity. Other views might include the 1"-height view for search result lists, a 2"-wide "teaser" view of resources for the PWP home page, and a sidebar box view for show beside search result lists.

We will only be able to create a couple hundred full treatments. For other common search terms, we will have to show more limited pages, either with less custom content and other content selected from sources such as our ontologies, bible dictionaries and encyclopedias, DBPedia, Freebase, WordNet, commonly-clicked search results, etc. Custom content could include pointers to important literature references.

We will add the list of entities related to a document as a field in the search database. Sphinx supports “multi-value attributes” (MVAs) that store a list of integers in a single field. We will map the entities related to a document to integers and store them in an MVA attribute. Then, when a search is made for an entity such as “compassion,” we can convert it to its numeric ID (nid) and restrict the search to documents containing that nid in the keyword list.

This approach can be used for a desired season (e.g. Easter), occasion (a wedding), desired resource type (video), lectionary week, etc., create entities with keywords for such constraints. Whether this is entity abuse is left to your conscience.

Scripture passages will be handled similarly. We can map scripture verses to integers and store them in MVAs, with three digits each for book, chapter and verse. So, Matthew 5:1-5 would be stored for the search engine as 41005001 41005002 41005003 41001004 41001005. Later, if someone searches for Matthew 5:3, Sphinx would find 45005003 in the MVA. Note that resources may be “principally about” one or more passages, but they may mention several other passages. For example, a sermon on Matthew 5:1-5 may also mention several other verses. We will have two MVAs, one for the verses that the resources is “principally about” and one for verses that it mentions.

Search Specifications

This project involves building a search server that is separate from the PWP and CCEL code bases. It will use a Sphinx server as a key component, but there will be additional processing of the query besides that done by Sphinx. For example, a query at PWP for music related to a scripture passage may result in two searches, one in this new search engine and one at Hymnary.org, with the result lists combined in some way. Therefore there will be intermediate search functions that send queries to the Sphinx search engine.

The server will be used by two different websites, PWP and CCEL, and possibly also PWP partner sites. The results it returns will be different for the different sites. For the PWP and partner sites, it will return lists of preaching resources, which are pages at other websites, and other result types such as entity treatment pages. For CCEL, the results are authors, books, sections of books, forum posts, and a few other resources such as entity treatments and study bible pages. Therefore we will probably want to have separate Sphinx search indexes for CCEL and PWP-related documents. We will definitely have separate ranking functions and search statistics.

There will be a great deal of data on the semantic search server in addition to the Sphinx search database, including named entities, relations between them, facts about these entities, search statistics, and the like. These will be stored in a MySQL database. There is one currently under development called "semantics" (short for Semantic Search) on Julian. Perhaps we can use Nee as the new CCEL semantic search server, at least for development.

Data could be returned from the server as an HTML list of hits and facts about the hits. The appearance of the HTML result list could be customized by supplying a custom CSS stylesheet in addition to a default one. The CCEL semantic search server will also be responsible for generating the various views of entities. They could also be returned as HTML snippets.

The search server will have a number of different types of named entities, including books, authors, people, topics, sources, lectionary weeks, and more. For each, we will have a different set of data to store. For authors we need birth and death years, and for lectionary weeks we need scripture passages. The semantic web approach to this problem would be to have a large digraph of labeled relations between entities, stored in a "triple store," without any schema. We would simply add to the triplestore whatever named entities (nodes) and facts (edges) we choose to add.

While this approach is very flexible, it also moves the problem of knowing what facts are available and how to use them to the code, and it is slow. I think it may be better to create a separate MySQL database table for each entity type. More effort will be required to design and populate these tables, but once they are in place, constructing the various views is fast and easy.

The CCEL semantic search server should support the following features:

- Sphinx query language for full-text search
- Probabilistic search results
- Search logging with query and entity view statistics
- Keyword handling
- Scripture reference handling
- Autocomplete
- Spelling correction and stemming
- Learned relevance ranking function with global, cohort, and user models
- Sidebar or box with info on entity if one is likely enough
- Searching forums

Query language

The Sphinx query language is quite sophisticated, and it should be more than enough to meet our needs. Queries typed into the search box will normally be plain text, but it should be possible to add additional Sphinx search operators and constraints. If there are any additional parameters we need to pass to the search engine, e.g. only return results from a particular site, they should be handled by adding constraints in the Sphinx query language to the query.

Probabilistic results

There may be multiple searches whose results have to be combined, or even multiple different search modules. For example, we may have a scripture reference recognition module that finds scripture references in a query string and inserts a link to the CCEL study bible into the result list. [This is just an example; we may be able to incorporate scripture searches into the standard search.]

In order to combine and rank the results of various searches and search modules, they need to return comparable values. We will standardize ranking values as log-odds values, that is, the natural log of p / (1-p), where p is our estimate of the probability that the user will click the result. If we think there is a 10% chance the user will click a result, the search module should assign it log-odds of -2.2. The log-odds function is also called a logistic function, and these values are suitable for use with learning algorithms that are related to logistic regression.

Search logging and statistics

We will need to keep search statistics in order to improve future searching. Some of the uses include populating the autocomplete table, finding the probability of linkages between keywords and entities, entities related to resources, and ranking results in future searches.

To do these things we will need two tables. One will log the distinct queries and the number of times each was entered. The other will have a row for every query x result, and it will log the number of times the result was shown for the query and the number of times the result was clicked. It should also log the way the result was

generated—a Sphinx result, a result from the keyword system, a result of another search e.g. at Hymnary.org, etc.

We should also gather resource and entity view statistics per day for PWP and for CCEL. This would enable us to say what the most popular sermon illustration was for the last week. Entity and resource views can perhaps be logged in a RAM-based table and added as another day's stats each night.

Keyword handling

In addition to the Sphinx full-text search system, an important system for handling queries will be the keyword/entity system. If a user searches for "compassion," we will want to offer our compassion treatment page as a top result. If they type 'pity,' we may also want to offer the compassion page, though possibly with a slightly lower probability. We can handle this by having a table of keywords that lead to entities with a certain probability. Then, searches can be handled two ways: the standard Sphinx full-text search, and looking for the search string in the keyword table, finding entities thus indicated.

The word "john" might have three entries in the keywordToEntity table: one for our treatment of the apostle John, one for our treatment of John the book of the bible, and one sending the user to the CCEL study bible, opened to the book of John. Entities have disambiguation phrases, so these three matches would result in the display of three disambiguation phrases,

- john -- view in the CCEL Study Bible
- john -- book of the Bible
- john -- the apostle

Ideally, these keyword results should be mixed in with Sphinx results, sorted according to the probability that they will be clicked. However, if it is not possible to convert Sphinx rankings into comparable log-odds probabilities with sufficient accuracy, we could simply list any high-probability keyword matches first.

We can also handle navigation terms with the keyword/entity system. For example, users type terms such as "download", "free,", "mp3", etc. We can direct these queries to the most likely pages by adding appropriate keywords and entities. For "free", we could create an entity corresponding to the CCEL's browse page with the cost=free parameter set and add free as a keyword pointing to that entity.

Initially, we can populate this table with hand-created entities and keywords, optionally with estimated probabilities. However, search data can be used to populate the keyword and keywordToEntity tables, and this system makes a nice way to add popular entity search results for queries to future result lists. Through entity-resource linkages, this system will also improve our keyword-to-resource mappings.

Scripture reference handling

One of the key distinctives of both the PWP and the CCEL is the intelligent handling of scripture references. At the CCEL, users may be studying a scripture passage and wish to see how various authors have handled it. At the PWP, users may be preaching on a passage and wish to see related preaching resources.

The search server should parse incoming queries to find scripture references, converting them to our numeric format. We can handle the scripture references through Sphinx, searching for resources that have the verse in one of the MVAs. However, we can also handle scripture references through the keyword/entity system if we support a scripture reference variable in the keywords. So, for example, we might add a keyword "commentary on {scripref}" pointing to the CCEL study bible entity. The link would open the study bible to show commentaries on the appropriate passage.

Autocomplete

The autocomplete capability addresses many common search problems. It can save typing, show disambiguation options, and help with spelling. It's a key part of the plans for the search engine. Let's consider how it will work.

query	count
john calvin	3072
jonathan edwards	1352
john owen	1214
john wesley	1192
john	893
josephus	812
job	683
justin martyr	661
jerome	640
james	530

Suppose the user intends to type the search term "john". After typing 'j' the autocomplete box shows the most common searches starting with 'j': "john calvin", "jonathan edwards", "john", etc. Some of the entries will be ambiguous. "John" may refer to a book of the bible, a scripture passage, or the apostle. To disambiguate among those terms, we could show three entries with disambiguation strings:

- john -- view in the CCEL Study Bible
- john -- book of the Bible
- john -- the apostle
- john [as a full-text search, not referring to one of those entities]

query	count
john calvin	3074
john owen	1219
john wesley	1194
john	895
john chrysostom	420
john calvin commentary	264
john of the cross	241
john bunyan	234
john gill	221
john of damascus	184

With each letter the user types, the list changes. After typing 'joh', the most common queries are "john calvin", "john owen", "john wesley", etc.

If the user types the entire word and hits enter, the search engine dispatches the search with the specified string. If the user selects a disambiguation option such as "john -- book of the Bible", the specified entity should be searched rather than the string "john".

The scripture reference parser should run as the query is typed, so that a query such as "matt. 5:8 com" may have a disambiguation option of "matt. 5:8 commentary -- view in the CCEL Study Bible". The scripture passage can be treated as a variable,

{scripref}, for the purposes of query statistics. Thus, "{scripref} commentary" should have a count in the query statistics table that is the sum for all scripture references.

Spelling correction and stemming

Stemming changes word forms to their stems in text that is indexed. So, for example, "stemming" would be changed to "stem" before being added to the search index. That way, searches for "stemming", "stem", "stemmed", "stems", etc. would all match "stemming". We should support stemming in full-text searches. Sphinx does this by default.

Spelling correction is expected in modern search engines. The way a search engine such as Google implements it is to generate many variations (single-letter insertions, deletions, and substitutions, transposes, etc.) of a word that was typed and check to see if one of them is much more likely than the word that was typed based on prior queries. We should implement spelling correction, though it is a lower priority than some of the other features listed here.

Learned relevance ranking

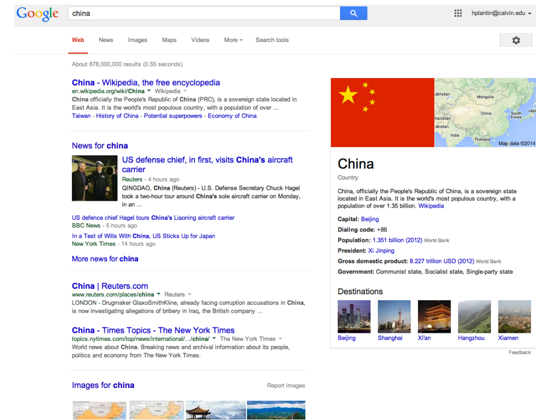
The key performance measure of a search engine is the extent to which it places the best resources in the first page of search results. This means that ranking search results is key to performance.

Sphinx computes a relevance value for documents based on the search query. However, in its default configuration, it simply measures a couple of factors such as how many of the search terms appear, how close together, and how close to the top of the document. This relevance value is helpful, but we have additional data available such as search history and user preferences.

One of the features we are hoping to implement is a ranking function that adapts its behavior to a user's preferences and search history. If the user selects the option "prefer resources from my own denomination," it should sort such resources higher. If the user has a history of clicking on resources from a certain author, they should move higher in result lists.

Another requested feature is a similar to the above, but for a group of users, i.e. a user's cohort. A preacher may work in a small group. If one user tends to prefer resources by a certain author, those resources should sort higher in result lists for others in the cohort. Therefore we will needed a learned ranking function with global, cohort, and user models all combined into a user-specific ranking function.

Sidebar info



If I type "China" into Google, the result page has a list of hits on the left and a sidebar of information on the right. We should do the same. If we discover that a user's search likely refers to an entity, we can include a box with information about that entity in a sidebar.

Searching forums

If the website using the search engine has forums, the search engine should be able to search forum posts as well.

Entity treatments

This semantics server will have information on many types of named entities, and this information will be used for a variety of purposes other than searching. For example, a Web page for studying the Bible at CCEL may need a sidebar with information about people, places, and other entities in the passage being studied. It should be possible to send an entityID to the server and get information about the entity.

Ontology

A more detailed description of the ontology, along with a draft of an initial list of named entities, is available as "PWP Ontology Plan."[1]

Cataloging documents

For the preaching and worship portal, we will have to gather information about resources and store them in the database. This information will be sent to the Sphinx search engine so that it will be able to select likely hits for queries. A specification of the information to be gathered in the cataloging process is available as specification entitled "Cataloging for the Preaching and Worship Portal."[2]

Sphinx indexing

Sphinx will be the full-text search engine underlying the search architecture. Sphinx will have to have separate full-text indexes for CCEL and PWP. For CCEL, the documents in the Sphinx index will include books, authors, full treatments, and the like. For the PWP, the index might include the following fields:

- Title
- URL
- Author/source
- Resource type
- Liturgical element
- Season
- Holiday
- Event
- Lectionary week
- Entities (MVA)
- Principle scripture verses (MVA)
- Additional scripture verses (MVA)
- Text (full text or description)

It is also possible that we may wish to omit some of these fields (e.g. event) and instead include event tags in the entity tag list.

Sphinx is installed on Nee, and it is configured to build an index for CCEL searching in the configuration file, /etc/sphinxsearch/sphinx.conf. The live Hymnary.org Sphinx server is running on Chesterton.

---

[1] CCEL Tech. Report #8, April 2014, http://www.ccel.org/tr/PWP_Ontology.pdf

[2] CCEL Tech. Report #10, April 2014, http://www.ccel.org/tr/Cataloging_for_the_PWP.pdf

Machine learning

Keywords and entities

There are several ways that we will use statistical machine learning. We can analyze prior search results statistically to compute the popularity of a search result as the log-odds that a click on a search result is a click on this entity. This can then become a feature in the ranking estimator. An even more useful value will be the popularity of an entity given the search query, i.e. the log-odds of a click on a result entity given the query. Another key piece of data will be the most likely search string given the first few letters of the string, for example so that we can know that someone typing “au” probably means “augustine".

To get these values, we will need to know the frequency of each query, the number of times each result entity was shown in a result list after each query (“hits”), and the number of times that result was clicked (“clicks”). Programs have been written to gather this data by analyzing web server access logs. However, for the new search engine, this data should be logged as queries occur. Each time a query is made, the results should be added to the database as negative examples (increment clicks). Any clicks on the result list should be changed to positive examples (increment hits). This implies that the query ID will have to be encoded into the result URL somehow so that when a user clicks on a link in a result list, we know what query generated the result.

Result ranking

Statistical machine learning techniques can also be used to improve the performance of the search engine by connecting keywords to entities and by learning improved ranking functions for result lists. Ranking the search results, for example, attempts to put the search results the user is most likely to click on at the top of the list. We can consider this a learning problem, where the ground truth is whether the user clicks on a result after a search. We try to estimate the likelihood by measuring several features and doing something like a logistic regression to develop a linear estimator function. We then use this function to rank search results, putting the most likely to be clicked first.

We can use an online learning algorithm such as “passive-aggressive online learning” to learn this function. The paper “The Learning Behind Gmail Priority Inbox” by Aberdeen *et al.* describes the use of this learning algorithm by Google for email priority ranking in a concise and understandable way. That paper, along with the Crammer *et al.* paper on passive-aggressive learning should be read before working on the learning aspect of this project.

Some examples of features that can be used to make this probability estimate are the search engine’s computation of the “relevance” of the document for the query, the popularity of the document, our own computation of the probability of the user clicking on the result after the query based on search history, and whether the denomination of the document matches the user’s denomination.

Logistic regression works by deriving a vector of weights that can be multiplied with a vector of features to compute the probability that the item is a positive example, or in this case, that the result will be clicked on. Logistic regression works on "log-odds" values rather than probabilities. So, if a particular document is clicked 13% of the time after a certain query, we can compute the "odds" of that result as p / (1-p), or in this case, 0.149, and take the natural log, yielding -1.9 as the log-odds value. A 50% probability has log-odds of 0 and a 5% probability has log-odds -2.94. This log-odds function is also called the logistic function. Note that multiplying probabilities is equivalent to adding log-probabilities, so these log-odds values are often added to estimate the probability of a sequence of events.

The ranking function will be a linear combination of these features. That is, for a particular query and result, there will be a list of numeric feature measurements such as relevance, popularity, etc. There will be a global ranking model, i.e. vector of coefficients to multiply with the feature values, to come up with an estimate of the log-odds of a click on the result. There will also be user and cohort models. These vectors can be added to the global vector before the multiplication in order to incorporate user and cohort models into the ranking. For example, some users may check a box labeled "prefer resources from my denomination." In this case, we would have a feature "denomination match" that is 1 if there is a match and 0 otherwise. We'd initialize the weight of that feature to say 2 for users who check the option, so that when the weight vectors are added and multiplied against the feature vectors, resources with matching denomination will have a boost of 2 in the log-odds. However, we could later modify the feature weight based on the user's actual search behavior, getting it closer to the value by which they actually do prefer documents with matching denomination.

Note that there will eventually be on the order of a million examples per year for training the global model, but far fewer for a cohort model and fewer still for the user model. This means that the global model should be modified by a far smaller amount after each search than the user model. The learning algorithm should probably be tuned so that a month's worth of usage largely trains the model. For the global model, that might be 100,000 queries, for a cohort model, 200, and for an individual, 20.

This also implies that we should have very few features in the user model, because as a rule of thumb 10 or 20 training examples might be needed per feature to train a model. Or, perhaps some features from the user model should not be learned but simply estimated, e.g. the boost for denomination match for users that check "prefer my denomination" could be set to a fixed value of two or a globally-learned value.

Sphinx computes its own relevance value as a function of document/query features such as the following:

- BM25 (default, bag-of-words relevance measure)
- Longest Common Subsequence
- Query word hit count
- Exact order

- Aggregate term closeness

There are several additional features that may be useful for ranking:

- Document features:
    - Popularity
    - Heat (popularity last week)
    - Author/source popularity
    - Freshness
    - Number of named recommendations
    - Featured?
    - Current season, holiday, or lectionary week matches doc
- Document/user features:
    - Author/source is in user's favorite list
    - Denomination match (doc, user)
    - User's recent search topics match doc tags
- Document/query features:
    - Popularity of doc given the query

It may be that some of these features don't have much predictive value and can be removed or that other features should be added. This list will no doubt evolve over time.

We will have many fields in the Sphinx index for documents, such as authors, sources, scripture passages that this document is primarily about, scripture passages mentioned, subject terms, tags, etc. Many of these will be used as filters. That is, if the query matches a named entity, then Sphinx will consider and compute relevance only for results that are tagged with that entity. If the user enters a scripture passage, Sphinx can filter the results so that only those with a matching verse are considered. However, we may want to use some of these attributes to affect the Sphinx relevance computation as well, if that is possible. Doing so would require more computation but may improve accuracy of results.

We should definitely customize Sphinx's relevance function for our purposes. I'm not sure of the extent of its customizability, but if it is flexible enough, we may be able to incorporate some of these additional features. However, others will have to be computed outside of Sphinx. The likelihood of a user clicking on a doc for a given query, in particular, would require a database access per document—not plausible when you are ranking many thousands of documents. Instead, we would have to query Sphinx to get a list of the most relevant documents and return log-odds estimates computed from Sphinx's relevance value. Then we would sort in results from other sources, such as likely results based on the most common documents visited after the given query or results from other search engines. Thus the learning and relevance ranking will probably have to take place outside of Sphinx.

Query parsing

Many queries are a single word, and for these no parsing is needed. Other queries are a string of words that the user is searching for in indexed documents. Again, no parsing is needed. However, part of our specification is that we should intelligently handle queries that specify a topic and a type of resource wanted, for example, "commentary on Matt. 5:8" or "sermon illustrations for Advent". Presumably, we will want to process such queries in two or more ways: one, treating it as a full-text search and letting Sphinx find relevant documents, and the other, parsing it and handling it more intelligently.

We should look for scripture passages and topic keywords to find ways that topics are specified. We should also look for resource types to find the type of resource the user is looking for. If it is possible to parse the query in that way, we can send a second query to Sphinx with the specified constraints and sort the results of the two queries together.

It may be that there are multiple possible parses of a query. For example, in the query "sermon illustrations for John", the word John could refer to various entities. We should send all interpretations that are likely enough to Sphinx.

Parsing a string to find scripture passages is a nontrivial problem by itself. There are many oddities in common scripture reference syntax. Sample perl code is available that works fairly well. In any case, we should develop a list of examples and make sure the code works on that list.

Testing

Search engines can be difficult to create because it's not obvious whether the results returned are correct. There may be even better documents in the search database that don't show up in the result list. Perhaps we can address this problem in part by creating a test program that feeds a number of queries to the search engine and checks the ranking of key resources in the result list. For example, if only one document has a long full-text match, it should be near the top of the list. Then this test program can be run whenever changes are made to the search algorithm to make sure that nothing breaks and results generally improve.

Architecture

Since this search engine will support both PWP and CCEL, it will need to be separate from the PWP and CCEL code bases. One option would be to put it on a separate server and have the PWP and CCEL sites communicate with it via HTTP. However, it might be more efficient to avoid the delay added by that HTTP link. Perhaps making it a Drupal module that can be included in the PWP and CCEL sites would be most efficient. For partner sites that use this search engine, perhaps it should be written in such a way that clients can request that query results can be returned as a block of HTML or JSON.

Analytics

Since searching is a critical component at the PWP and CCEL, we should have a way to assess how well search is performing. The goal should be to learn what kinds of searches users are performing, how well the search engine is giving the results users want, whether the pages thus found meet users' needs, what are the top queries that aren't being handled by the entity system, and in general how we could serve them better.

Basic search stats for a given timeframe that should be available include number and type of searches, common search terms, number of follow-on searches, amount of time spent with target page, a proxy for whether searches were successful, and the most common searches that were not successful. We could analyze search logs for much of this data, since it probably doesn't have to be real time.

Bibliography

Aberdeen, Douglas, Ondrej Pacovsky, and Andrew Slater. "The Learning Behind Gmail Priority Inbox." LCCC: NIPS 2010 Workshop on Learning on Cores, Clusters and Clouds. *http://research.google.com/pubs/archive/36955.pdf*. 2010.

Conway, D., and J. White, *Machine Learning for Hackers*, O'Reilly, 2012.

Crammer, K., O. Delek, J. Keshet, S. Schwartz, and Y. Singer, "Online passive-aggressive algorithms, JLMR, 7:551-585, 2006.

Pan, S., and Q. Yang, "A survey on transfer learning," IEEE Transactions on Knowledge and Data Engineering, 22(10):1345-1359, October 2010.

George A. Miller (1995). WordNet: A Lexical Database for English. Communications of the ACM Vol. 38, No. 11: 39-41.