WHAM! Lessons Learned

Original Google doc link

Usability/User evaluation high level takeaways

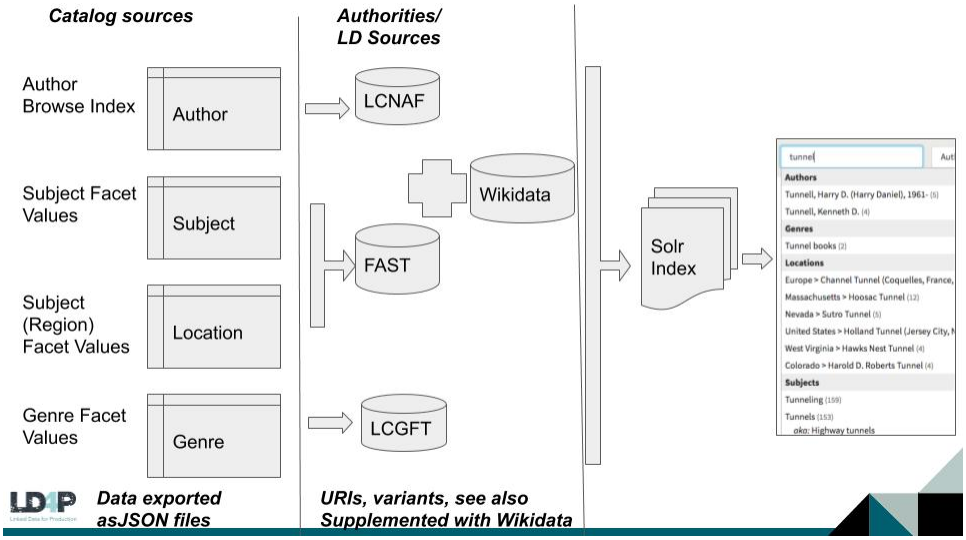
- Given the current situation with the pandemic, we did not schedule user tests with students and faculty but hope to do so if exploring this feature in the future. With the aid of the Usability Working Group's expert facilitation and help, we conducted think alouds with five user representatives. The tasks we used with example screenshots of autosuggest answers are captured in this set of slides. Details on the usability tests are available here.
- High level results and takeaways
    - In general, participants were able to use the information displayed in the autosuggest to distinguish between types of entities and to identify headings related to variant labels or connected using "see also" properties.
        - Participants were able to find authors, subjects, locations, and genres when asked to find entities from each of these types in the tasks.
        - Participants were able to use the number in the autosuggest results to answer the question regarding the number of items in the catalog for a particular topic.
        - Participants were able to use the descriptive text retrieved from Wikidata to distinguish between authors with similar names but different occupations. One participant did want to see if typing in the occupation along with the name would help in retrieving results, but we noted that we are not matching on occupation text.
    - For both authors and subjects, all participants were able to search for a variant label and find the preferred heading. Most participants understood what the term "aka" stood for in the results and thought this display of information was useful. When searching for an author, one participant indicated they weren't sure which heading was authorized since both the variant and preferred versions ended with date strings.
    - For the shared pseudonym example, participants were able to find the names linked with "see also" but it was not clear to all that the searched name was a pseudonym shared between the linked names. Suggestions for clarifying this connection included using the term "pseudonym".
    - 4 out of 5 participants appreciated the resulting knowledge panel on the results page as a way of confirming the search they had conducted.
    - One participant noted the discrepancy between the authors and subjects where the former shows descriptive information from Wikidata while the latter does not.
        - We suggest that future work should explore how to provide consistent information about the same person regardless of whether the URI returned is for an authorized name or for a subject heading.

Data, API, and Indices

- The starting point for building the autosuggest index was querying indices in the catalog itself. Using a controller and a simple UI added to the Blacklight application, we created JSON files where each item consisted of three fields: a label, a rank and an "authority" boolean identifying whether the label came from an authority file. When building the JSON files for authors, the controller retrieved this information from the author browse index, which exists separately from the main catalog search index. When building the subject, location and genre files, the controller used the corresponding FAST facet from the main search index. Because of the sheer number of documents, especially for authors, we had to create numerous JSON files. Looking ahead, we would have to figure out how to incorporate updates made to the main catalog index. It would be best if this could be done incrementally, creating JSON files for only the new documents and facets, rather than creating all new JSON files for the entire index.
- To experiment with auto-suggest approaches, we set up a separate index using data from the library catalog for authors, subjects, genres, and locations.
    - We explored both Solr's built-in auto-suggest functionality which relies on a suggest request handler and search component as well as utilizing the regular search request handler. Our final implementation relies on the regular search request handler as that solution functioned well for us and allowed us the flexibility to query and retrieve the information needed for the front-end display.
        - Although the suggest search component worked on a smaller test index, we encountered some errors with what appeared to be memory issues. Future work should explore how to resolve those errors. In addition, although we could configure the suggest handler to return additional information from the matching solr documents in the index, the handler's results are comprised of all the text that matches the query. The regular search request handler instead gives us the Solr documents as the search results. Since the documents correspond to the entities of interest, these results allow us to display the label but also retrieve related information such as see also headings, variants, and Wikidata descriptions.
        - We supported matching the beginning part of words by setting up a special type of field that could allow for the appropriate analysis at the query and indexing steps. (More information can be found here [TO DO: link document explaining indexing approach and information]). All text fields (i.e. those ending in the "_t" suffix which mark them as dynamic text fields) are copied over to the "text_suggest" field which is of the type that allows matching against the beginning portion of words. Additionally, we configured the defaults for the search request handler to match against all the words in the query. The schema and field configuration is stored in GitHub here: https://github.com/LD4P/discovery/tree/master/solr_config/wham/suggest
        - We aimed for setting up and using as simple as possible a configuration with which to conduct these experiments. Alternatives exist such as sematext (which is used by the University of Ghent) and could be explored later if we wish to finesse query matching. The catalog and browse indices don't contain any stemming or phonetic matching and we did not include these features in this round of work. That said, misspellings and phonetic matching may be useful to explore in a future round for certain use cases. .
    - As noted above, heading strings authors, subjects, locations and genres were first retrieved catalog indices and then saved as JSON documents. Some of the details on the process of parsing and using this information to populate the index are described below along with challenges and workarounds where relevant. High level results are presented here:
        - Matching headings against lookup sources has to take into account the possibility of throttling at the vocabulary lookup API server end. Additionally, ending punctuation is not consistent between the browse index, facet values, and LCNAF. For example, the script may have to take into account that a corresponding URI may exist for a heading but without an ending period. Having URIs present consistently within the MARC itself would have reduced the need for these additional lookup steps.
        - Saving results where URIs were not found also helps with reviewing those results and manually confirming whether or not there were matches. Where matches are overlooked, the script can be updated to address those cases where possible.
        - Library of Congress data retrieved for a given entity may contain more/additional relationships (e.g. relationships to Wikidata URIs) that are not available through Dave's SPARQL endpoint for that data source. This may be a result of the aggregate data

downloads from LOC not capturing data that is available at the entity level, or that the most recent data downloads don't contain all the relationships at the entity level.

- The image below summarizes the process for retrieving information and populating the autosuggest index.



  - Ruby scripts processed a selection of these JSON files and queried the corresponding vocabulary sources to retrieve URIs for these strings.
    - This query was done using suggest API requests for the relevant vocabulary. The label of the result was also compared to the original heading and URIs were returned for only those situations where the labels matched.
    - Queries that did not match were saved to a separate file. Reviewing this file indicates that there were possible matches that were not saved. Possible reasons include the suggest API request not being available (perhaps due to throttling as the script processed many headings one after the other) as well as mismatches due to punctuation differences. For the former, the script was updated to include a time delay between sets of requests. For the latter for LC sources, the script checks first against the heading value as retrieved from the data and, if no result is returned, then re-executes the lookup with the ending period removed.
  - Where the script returned a URI, information was retrieved to generate a Solr document representing that entity.
    - Queries were performed using Dave's SPARQL endpoint to retrieve variant labels for those URIs in their respective vocabulary sources. These variant labels were added to the Solr document.
    - Additionally, for author entities, see also URIs and their labels were retrieved for LCNAF using queries against this same SPARQL endpoint. This information was also added to the Solr document for those entities.
    - For authors, Wikidata was queried to see if any Wikidata URIs corresponded to the Library of Congress URI (or local name) and description, image, and pseudonym text were retrieved where available for those Wikidata entities. Wikidata information was also added to the Solr document.
      - Although the Library of Congress individual entity data shows many Wikidata relationships, the data queried through Dave's SPARQL endpoint does not seem to always contain that information. We queried Wikidata directly to retrieve equivalent entities for LCNAF URIs.

Development-related and UI-relevant results

- The user interface relies on two main components, a javascript file and a controller class.
  - Using the jQuery autocomplete feature, the javascript makes an AJAX call to the controller, passing the query term as a parameter. Once the controller returns a response, the javascript overrides the default autocomplete _renderItem function. This step is necessary to display the the type headings in the autosuggest dialog, as shown here:

    Another customization of the jQuery autocomplete code involves the select function. When the user makes a selection from the dialog, three things happen: the function (1) maps the URI of the selected item to a hidden field within the search form, (2) sets the search type option based on the type of item the user selected, and (3) submits the form to initiate the search.

  - The AutosuggestController performs the actual Solr query and then processes the results. The controller takes the response and groups the documents based on their type (author, subject, etc.), and it then sorts both the groups and the items in each group alphabetically. It is within this controller that we identify variant labels, applying the "aka" label, and also pseudonyms, which we append with a "see also" label. This controller applies some HTML to the items that it returns to the javascript. It's this step that makes it possible to have pseudonyms be separately selectable items, but variants part of the main, preferred label.

    Even with the packaging work we have done, it's likely that at least some UI customization would be required to meet the requirements of a given institution. That remains an open issue going forward.

- Knowledge panel result based on autosuggest selection
  - Tim updated the autosuggest JavaScript to pass the URI for the selected auto-suggest result to the search results page. We added a small knowledge panel on the top left of the results page which pulls in the image and description from Wikidata, the heading or label from the query, the works by and about sections from the author browse index page for the heading, and digital collections results using the heading as a keyword query.
  - Generally, usability test participants were able to identify this knowledge panel as corresponding to the result they selected.
  - At least one participant mentioned the discrepancy between the numbers displayed in the knowledge panel for works by and the number of results from the search. The three sets of numbers involved come from different sources:

        - For authors, the count in the auto-suggest comes from the author browse index which should correspond to the “works” by number in the knowledge panel, but the search results count comes from searching for the heading in the author field. These numbers may this be different.
        - For subjects, genres, and locations, the facet counts for these types of information provide the count for auto-suggest. Selecting any of these will again perform a search with the selected label in the corresponding field type (i.e. picking a subject from auto-suggest will perform a search in the subject field). The search result count and facet value count may be different.
- RSpec tests
    - We developed two sets of Rspec tests for the autosuggest functionality. The first set tests the contents of the Solr response to ensure that the documents include all of the expected fields. These tests perform actual queries against the index. The second set of tests is for the methods that process the Solr response and that produce the JSON that is returned to the javascript file for display in the UI. This second set uses fixtures instead of actual Solr queries.

Packaging

- A modular way to package the feature is with a Rails engine (a type of Ruby gem). An engine can be installed in any app that uses Blacklight.
- Per an example brought to our attention by Duke University, it is possible to override Blacklight functionality with another gem. This means we could override Blacklight’s existing autocomplete feature, modifying it to support the more fully-featured autocomplete / autosuggest feature that we’ve built.
    - Blacklight uses the Bloodhound suggestion engine, which is based on the jQuery typeahead plugin.
- In the interests of saving time, instead of rewriting the feature using Blacklight-native Bloodhound techniques, we ported our existing proof-of-concept code into a Rails engine. This engine, we named Nectarguide, is intended to be used with Blacklight’s built-in autocomplete switched off.
    - Code repository for Nectarguide
    - A video demo of Nectarguide in action
    - Details of learning about Rails engines

Possible enhancements and areas for future improvement

- Given the discrepancy between author browse index counts, facet value counts, and the search result count when searching for a label in a particular type of field (e.g. subject or author), we should explore which information we display and how to make a consistent experience for the user.
- When searching for a person, whether subject or author, should we display the knowledge panel so it offers a consistent experience across search types or vocabulary sources?
- Future work could also include the ability to add support for applying facets and filters to the auto-suggest results. In other words, the auto-suggest could show which authors and subjects are available given the facets already selected, thereby reacting to choices already made by the user.
- Packaging improvements include modifying the Nectarguide engine to repurpose more of the existing Blacklight autocomplete functionality; for example, rewriting the UI using the Bloodhound API that Blacklight uses for its autocomplete.
- Additional evaluations should be conducted around accessibility to ensure the autosuggest solution is accessible.