

Improving the Software Package Search Experience for Developers

Final Report

Avery Nisbet, Joshua Choo, Mengshi Feng, Vivian Liu, Yidan Zhang

April 30th, 2018

Executive Summary

Living in the age of information, searching has become an important part of our lives. Around 80 million searches happen on Google.com in the US every second on an average basis (Fishkin R., 2017), not to mention the peak rate during big events. Hundreds of thousands of talented engineers are working on the core search engine in order to make information accessible to users worldwide with ease.

Though the core search function is already well developed, research showed that the current search interface does not perform well when it comes to software package search (Brandt, J. et. al., 2010). To solve this problem, we focused on developing a novel search engine. This search engine contains five distinct technical systems. A crawler was developed to retrieve relevant web pages. Feature extraction isolates package information that a developer may care about. The indexer acts as a searchable repository of software packages and web pages. The ranker ensures that the best results are shown to the user. Finally a user interface was developed that was specifically designed for software package search.

Up to this point, we have developed a minimum viable product for the proposed search engine. It improves the search process by surfacing packages that are relevant to a query, and organizing relevant information that a developer needs to choose a package. Furthermore, a results can further be filtered by the users to refine ambiguous queries. However, the current search engine only contains a snapshot of the internet and only works for software queries. Future work could make the search engine continuously update with new pages on the web, work for general queries, and include more package information that are more difficult to retrieve such as package dependencies.

Engineering Leadership

The Search Engine Industry: Few Players in a Rapidly Changing Technological Environment

The search engine industry is dominated by two giants, Google and Microsoft Bing. Yet despite the few players, search technology is rapidly changing and the industry is growing. Between 2009 and 2017, the industry revenues grew 8%, year on year (Search Engines in the US, 2017). Furthermore, Bing, despite entering the field in 2012 has tripled its revenue over the past five years (Search Engines in the US, 2017).

With rapid technological changes in a growing field, there are three market factors that point to the importance of continued investment in software development for a company's success. The first is the concept of vertical searches, where users can search under specific topics. Microsoft is using this design pattern to help cater to the needs of their users (Search Engines in the US, 2017), indicating that different algorithms are being developed for unique search domains.

The second factor is the growing trend for mobile search. The number of users accessing the internet through mobile devices has been increasing rapidly since 2008. This means that established companies need to continue developing their platform and changing their algorithms to prioritize websites that support mobile view.

The third factor is that most search companies obtain more than half of their revenues offshore. For example, Yahoo has less than 1% of the US market, but has the largest market share in Japan (Search Engines in the US, 2017). Yet typically, each of the industry giants are competing in multiple foreign markets. This suggests that in order to compete, software companies need to constantly develop its software for dynamic international trends, not just domestic trends.

For these reasons, each company is pouring investments into research and development. Any additional algorithms to help a company cater to its user base is consistently a sought-after tool to gain market share and increased profit.

Software Package Search: A Problem Left to be Solved

Nowadays, searching the web has become an important part of our lives. For programmers, one of the most frequent search activity is searching for packages they're currently using or planning to adopt. According to Google Adwords, the average number of searches for software packages is 1 million to 10 million per month (Google Adwords, n.d.). These facts show that there is a large potential user base in the market for software package search.

One might think that the search industry is a winner-take-all market: having big companies take over 80% of the industry, it is impossible for others to thrive or even survive. However, despite the high demand of market and the effort that major search engines have made to refine the search result they're showing, our research shows that the services major search engines provide does not suit users needs. There are still problems left to be solved for software package search and our goal is to transform how developers search for software packages.

To better understand our target users, we conducted over ten interviews, mostly Berkeley students studying computer science, to learn their experiences with software package search. We asked questions such as “Tell me about the last time you had to look up a new package to use?” and “how often do leverage third-party packages in your projects?” From these user interviews, we found that many developers find searching for suitable software packages for their software project a difficult and time consuming. At least six people we interviewed mentioned that they would not find useful information on their initial queries, and would have to continually look through less relevant links and refine their search terms to arrive at useful links. Other times, they may find it hard to formulate the query using suitable search terms to obtain the desired results. These results from our interviews show that current search model of major search engines may not be the best fit for programmers in terms of both search accuracy and information gathering. We believe our work could address these issues to save programmers time and effort and therefore help improve their efficiency.

The paper by Brandt, J. et. al. about how software developers have to scour through many web queries and web pages to find sample code or information that they need (2010) supports our findings from the user interviews. Currently there is no efficient method for developers to compare packages, besides doing their own research and opening up multiple tabs for a visual comparison. This tedious process shows that current search engines are not optimized to streamline this comparison process for software developers.

With our product, we aim to streamline this search process for developers. We hope to make it faster and easier for developers to find and get started with suitable software packages. Developers can then spend more of their time and effort on their projects, rather than spending it scouring the web for helpful documentation and tutorials to start using the packages.

Business Intelligence: How We Stand Out Among Competitors

The broader software industry is one of the fastest-growing industries in US. According to the research of Business Software Alliance in 2007, the software and its related services industries employed 1.7 million people in US (Business Software Alliance, n.d.). New job opportunities are opening up and employment for software developers is expected to grow by 19 percent in the next 10 years (U.S.News, 2017). The target users of our search engine are people who have some level of programming skills or people who are currently learning programming. Despite the promising future in the industry, there are several competitors in the search engine industry.

There are two main groups of competitors to our product, the major search engines and products that provide comparisons between software packages. Compared to our competitors, our product would include more specific metrics that would better suit the needs of software developers.

Currently, major search engines like Google and Bing perform very well when it comes to generic queries that people have in their everyday lives. However, searching for suitable software packages is a very specific querying task. Many times, developers would find the top search results to be unhelpful in helping them get started with a package or with selecting the most suitable package for a task. Our product includes features like faceted navigation, a technique for accessing information organized in semantic categories and query filtering to allow developers to narrow down and prioritize the type of results they wish to see.

Slant and libraries.io are the two most popular websites that allow developers to compare different software packages. Applications like these do not provide a meaningful and concise visual presentation of search results. Also, they do not have filters in their faceted navigation that would help developers find the most suitable packages for their tasks. We can serve our users better by having features like a carousel-like results interface that makes it easier for them to perform visual comparison between search results. Also, we include a wider range of facets for our users to obtain more relevant search results. These facets include the language the package is written for, the license needed to use it, as well as the number of collaborators working on the package.

Our final product will be a search engine for software packages with a special carousel display that allows users to compare packages by features they care about side by side. This design accommodates the advantages of our two sources of competitors: the user interface is designed to better display package information and provide easy comparison, yet the search engine maintains integration of information from a wide range of websites. With more and more packages becoming available online while users becoming less patient but still want to gain a decent understanding to product before trying it (Anderson M, 2018), we believe our product will offer a better service that suits users' need.

Social Context and Ethics: Contributing to the Open Source Community

In addition to technical implementation, there are also a few ethical issues that we want to be mindful of. First, since our project heavily leverages open source projects such as GitHub, Stack Overflow, and libraries.io, we want to contribute back to this community by not restricting the

use of our technology. That is, we will make the data we calculate regarding software packages openly accessible by other developers who may be able to use them in other ways.

Second, the ranking algorithm we use will be crucial in determining which packages to return for each user queries. We want to make sure our algorithm only considers factors that are truly important to developers. That is, we want to ensure a bias-free ranking algorithm that is user-centric and never temper it for any personal gains.

Lastly, we will stay vigilant against malware. We want to ensure we never recommend or provide links to malware for our users. This will require us to be very careful about our crawling and indexing. In the end, we want to build a search engine that truly provides values to developers by making it easier for them to choose software packages while knowing the recommendations we made are safe and bias-free.

Technical Contribution

Setting Course: Narrowing Down the Solutions with User Interviews

To determine how to best improve software developers' search experience, two key questions needed to be answered. Firstly, what metadata (package information) do software engineers use to evaluate packages? Secondly, what search engine design features would allow developers to quickly find what they are looking for?

Initial research suggested that developers' evaluation metrics wildly varied. There were some unifying themes: such as having links to source code, and an effective description (Gallardo-Valencia and Sim, 2011). However, metrics could change depending on who was searching, and what their current project was. We identified what metadata about software packages was useful to developers by conducting user interviews. The list of metadata we narrowed down according to our interviews includes:

1. Rating
2. Tutorial documentation
3. License
4. Forum response time
5. Publication date
6. Number of contributors
7. Size
8. Dependencies

9. Sample code-snippets.

Next, the myriad of information that the software search engine gathers must be presented in a clear and seamless manner. The team examined the presentation of search results on a variety of web pages including Yelp, Amazon, wikiHow and StackOverflow. Taking in inspiration from these sites, our resulting design included star ratings, disambiguation pages, filters and side-by-side comparison, faceted navigation and a carousel.

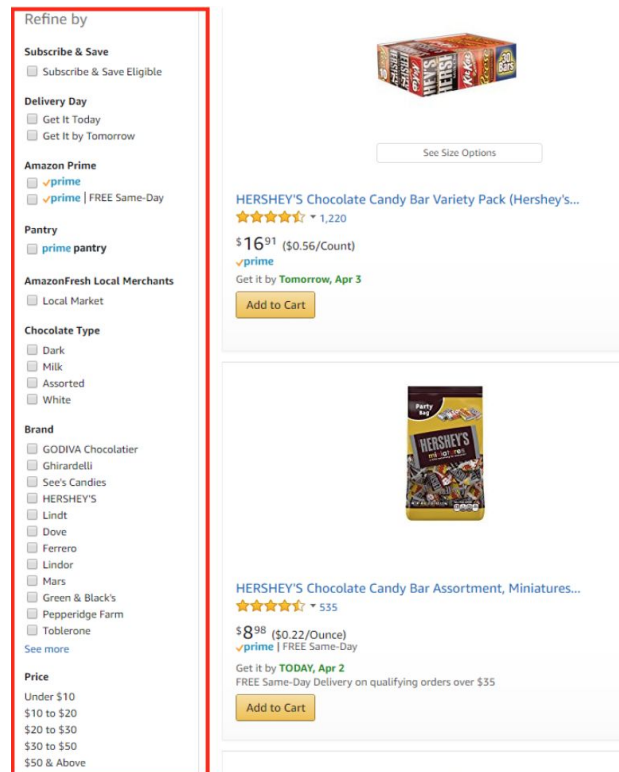


Figure 1 - Amazon's search interface includes a faceted navigation bar to the left, entitled "Refine by"

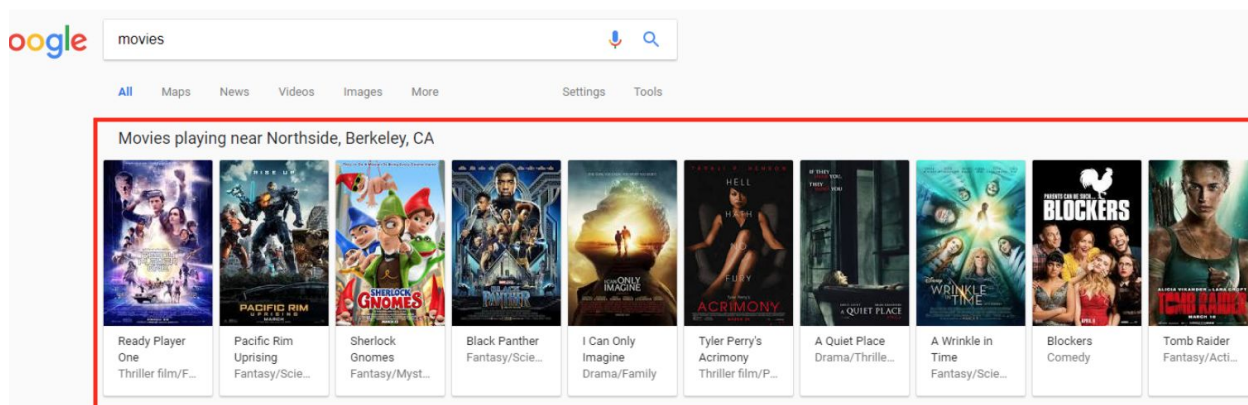


Figure 2 - Google provides a carousel to show all the movies relevant to a user's location

To determine which design features were best, paper cutouts of each design feature were created, along with a drawing of a basic search interface (Vasudevan, 2018). We decided to adopt this prototyping method instead of some prototyping tool such as Adobe Illustrator was that the paper cutouts were cheaper and easier to make changes. Our paper prototypes can be seen in Appendix II. When brought to end-users for interviews, the users could point to the paper to click through a sample search process. The low-fidelity paper prototype provided flexibility to easily drag and drop new features or change the positions of the features on the page. Additionally, we received feedback on how users might navigate through our webpages, what needed further explanation, and firsthand experience on how users would evaluate packages with our product. Our final design, below, includes a query bar, faceted navigation, side by side comparison, a carousel to display packages, and then a list of regular search results.

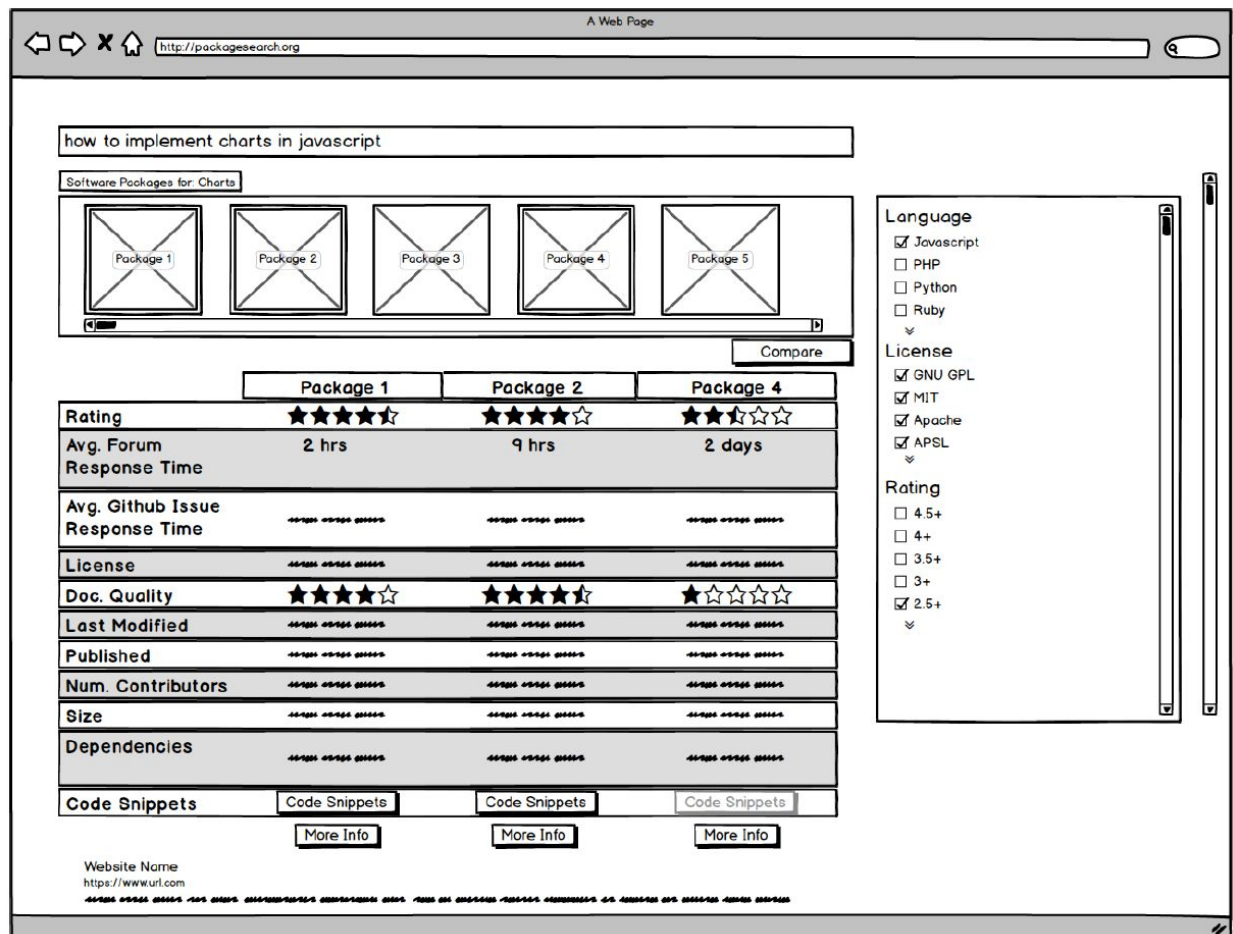


Figure 3 - The initial wireframe for the search engine

Success Metrics: How Do We Define Success

Defining the success metrics is important as we need a criteria to evaluate our final product. Overall success of the product depends on whether the search experience of developers is improved. This means that the product needs to be fast, surface relevant and a wide variety of information, and visually organize that information intuitively.

The speed criteria will be measured by the average time the indexer takes to return a list of information with different queries. The indexer must also display a variety of information and will be measured by the percentage of packages that is correctly indexed and accessible versus the number of packages we feed into the indexer.

Surfacing relevant information depends entirely on the ranker, which determines which search results are relevant to a query. For the ranker, the success is not easily evaluated due to the lack of labelled data. That is, there is no readily available benchmark to which we can compare our ranking results. Hence, we will attempt to evaluate it using existing search engine outputs. We will write a script to get questions and answers related to software packages on Slant, a product recommendation tool based on peoples' reviews. Another metric to use would be the feedback we obtain from others when we perform user testing when we get our first end-to-end prototype up. Users should feel that our tool helps them in some way that existing search engines do not. Some example metrics could be time taken to select a package, number of web pages browsed before selecting a package and usefulness of the navigation facets.

Progress Decoded: Where We are with Each Search Engine Component

To make our application accessible, we deployed our service on Google Cloud Compute platform. As one of the many available cloud services in the market, it not only provides us sufficient storage space, but also empowers our system by providing the scalable computing capacity which will eliminate our need to invest in hardware up front (Galarnyk, 2016). By researching the existing search engines' framework and inspecting the specialities of our system, we decided to split up the workload of our system into five major components: crawler, feature extraction, indexer, ranker and user interface. The detailed description of each components are as follows:

Crawler: gathering the relevant web pages from the Internet

The first component of our design is a crawler that could visit all relevant web pages from countless web pages on the internet. That is, to pull raw HTML text information from these

websites. To be more specific, we need a crawler that could distinguish web pages related to software packages from all other kinds of pages.

To define what is “relevant”, we leveraged information from libraries.io, who already gathered over 3 million packages from over 33 package managers. We extracted the package names and some other useful metadata that are less time-sensitive (we won’t have to periodically update this data) from the dump data that libraries.io provided into crawler’s output data.

Although there are many websites that can be regarded as relevant to our system, we started with GitHub and StackOverflow since they are known to contain software related information. The other advantage of starting with these two sites is that they both provide well-documented Application programming interface (API) for developers. We have customized our crawler for each of them using their public API respectively.

After we obtained the packages list from Libraries.io which contains over 3 million packages, we kept those package with 10 stars or more on their GitHub repo and discard others. Such cutoff helped us ignore those unpopular packages, which facilitated the later querying process using GitHub API. After doing so, the list shrink to a list with 110,000 popular packages along with their package manager information and GitHub repository name. We then utilized GitHub search API to run separate query each repo name in the list. The GitHub search API has different end point from the normal GitHub API which will return all the GitHub related metadata given a search query. We passed package names as well as owner’s GitHub usernames as the query term one by one and then concatenate them into one JSON file. One challenge we met here was the strict rate limit of GitHub search API, which is explicitly stated to be 30 requests per minutes. That means, we have to pause for two seconds everytime we send a query to the end point. It took us such a long time to crawl all GitHub data using this approach politely.

To crawl Stack Overflow pages into our system, we leveraged on the API provided by Stack Exchange, StackOverflow’s parent domain. Such API provides list of all StackOverflow questions and their corresponding answers ordered by question’s upvotes. We crawled first popular 220,000 questions and their corresponding answers using the API. The API returns metadata and markdowns of questions and the corresponding answers’ metadata and markdowns as list of children objects in a JSON format. We further extracted the code snippets from the html version of the page using Beautiful Soap with ‘code’ as the parsing keyword. One good thing about StackOverflow API was that, unlike GitHub search API, it has relatively mild rate limit which is 30 requests per second. The final crawled data was stored in nested JSON structure.

The other task for crawler was to obtain the image or icon of each package. This was done separately from package crawling since the image urls were not part of Libraries.io metadata neither GitHub's. The approach we chose was applying Favicon Grabber API to each package's homepage. The Favicon Grabber API will take the url as input and output the urls of the favicon if there is any. The returned value really varies among packages. Some packages had exact one icon url, some had none and the others had multiple icons with different size and format. It the later was the case, we simply chose the one has the largest size to generate the best resolution for its display in UI.

The tasks mentioned above were all written in Python scripts. The data crawled from GitHub is over 5 GB in size that contains around 110,000 packages along with their metadata. The pages crawled from StackOverflow are stored as around 150,000 JSON objects with each one object contains the contents and metadatas of one question and those of its answers. The crawler will perform consecutive crawling as the more data we have the better performance the ranker can achieve.

Feature Extraction: Interpreting Unstructured Data and Consolidate into Correct Format

The next component of our project is feature extraction, which is responsible for interpreting and consolidating data gathered by the crawler. Data crawled from different websites differs in format and contains pieces of information that when combined, make up all the metadata that needs to be fed into the indexer to evaluate software packages on. It is up to the feature extraction component to do this conversion and restructure the raw date in a sensible way.

Since we are building a search engine for software packages, the primary object we are interested in is package, and we would like to build out our data structure accordingly. That is, the output from our feature extraction should be a list of package objects with all relevant information attached. As mentioned, we crawled a large amount of data from both GitHub and StackOverflow, so to generate our desired output, the question becomes: given a package name and its attributes gathered from GitHub, how can we find all the relevant StackOverflow pages efficiently and put them in a format that could support all types of queries from the front end.

Our initial approach looped over all the GitHub objects first. For each GitHub objects, we then looped over all the StackOverflow objects looking for potential match. If a given StackOverflow post contained a package name, whether that's in the question title or in one of the answers, we considered it a match. To do this string matching, we leveraged a tool called BeautifulSoup. This approach gave the correct output; however, it was very slow since string parsing and

matching in python could be very slow itself. Given the amount of data we need to process, this approach turned out to be infeasible as it will take over 30 days to run on a single machine.

We identified the bottleneck for the algorithm above to be finding matching StackOverflow pages for each GitHub object. To speed up this process, we decided to use the indexing tool Solr to do the matching instead of BeautifulSoup. To do this, we first indexed all the StackOverflow pages using Solr, then for each GitHub object, we query the indexed posts with the package name. Solr will return a list of IDs for all matching StackOverflow pages. With this information, we can then easily look up and attach these matching posts to the correct primary GitHub objects.

It is worth noting that the reason why we decided to preprocess the crawled data from GitHub and StackOverflow and combine them into package objects is because we would like to have both of them influence our ranking algorithms. That is, for any package, we value both attributes that we get directly from GitHub such as star counts, as well as information we gather from StackOverflow such as total number of matching posts. Another reason to combine the two is that these StackOverflow pages can give us a lot more information regarding the usage or capabilities of a given package. For example, the ReadMe of a package on GitHub may not explain all the use case of a package, but it is more likely that someone used a function or asked a question regarding these use cases on StackOverflow. When we combine all these info into Package objects, it is easier for Solr to find appropriate packages based on StackOverflow posts and return the results.

The feature extraction tasks mentioned above are all done in python scripts. Since the crawled data from both sources are extremely large, we store both our input and output objects one per file. This way, we can also scale better in the future when we want to index more pages, as we won't have to re-write any of the previous output files. So far, we successfully processed information for over 110,000 packages and consolidated the data to fit the schema required by the indexer ([Appendix I: Solr Schema Design](#)). Although there are certainly many more packages out there, these sample packages allowed us to build a minimum viable product that returns useful results for many queries.

Indexer: Providing Quick Look-up on Formatted Data

The indexer acts as a hook between formatted data and selected results. We used Apache Solr, an off-the-shelf product, as the indexing tool, which not only allows easy indexing for different format of files, but also has a standalone cloud server and a REST-ful API, an API that uses HTTP requests to fetch data from websites, which makes it easy to connect with our front end

(Apache Solr Features, n.d). The indexer will generate a 'menu' for all the data we get from the previous step that allows quick lookup of information by different types of query term.

One of the most important piece of work for this step is designing the schema, that is, how the data looks like for each package. This establishes the type of queries we can perform as well as provide a guideline for the work of feature extraction and ranking algorithm. To decide on the schema, we first examine and identify the two types of queries we need to perform: one is getting a list of packages and its metadata based on query on the features they support, the other is getting forum information such as code snippet given query on the package name. Based on these query objection, we decided to use nested document feature, using the package information crawled from Github as our primary object, and storing all external resources such as Stackoverflow pages as child documents. The sample schema is shown in [Appendix I: Solr Schema Design](#).

There are 5 major things we need to accomplish with the indexer:

1. First we need to setup the server. To make the indexer accessible any time from any machine, we hosted the service on Google Cloud Compute Engine by setting up a virtual machine instance with `solr-7.2.0` installed. Two nodes are used to host the collections, with 8983 be default node and 7574 be backup node. To make the service hosted on Google Cloud accessible from all machines, the firewall rule of instance need to be properly setup so that port `tcp-8983` is open to public.
2. Second we need to set up the schema for the indexing collection and build lookup table on the nested information of Github and StackOverflow page. This is done by first creating a `collection`, then `add-copy-field` on all fields to make them searchable. Notice that for fields that we want to create faceting navigation on (e.g license, language), we need to set it to `string` type instead of `text` so the terms will be searched as a whole (exact match) instead of splitted to individual word. Once the schema is set, use `bin/post` to index all the files. Here we need to use `set -format solr` to make Solr understand the `_childDocument_` structure. The Solr interface of this collection can be view at <http://35.197.49.105:8983/solr/#/nestedpackage/query>
3. Since we decided to add in icons in later stage when the main indexing collection is already set up, we chose to use a separate collection to store all the icon information about packages. The Solr interface of this collection can be view at <http://35.197.49.105:8983/solr/#/icons/query>
4. Next we prepared several queries that will be used by front end. We have two different search task, one is for the carousel display which we want to perform search on all three levels of information: Github packages information, StackOverflow questions, StackOverflow answers, and return only the Github level information. We used "block-join query" and an example query is as follow:

```
http://35.197.49.105:8983/solr/nestedpackage/select?fq={!parent  
which= path:1.git }&q=[QUERY TERM]
```

For the web page results display, we want to perform search on StackOverflow questions and StackOverflow answers level and return StackOverflow questions level information. An example query is as follow:

```
http://35.197.49.105:8983/solr/nestedpackage/select?fq={!parent  
which=path:2.stack}&q=[QUERYTERM] (+body_markdown:[CONSTRAINT])
```

5. Finally we need to connect the indexer with the front end which is Django in our case. The connection is straightforward with the use of `urllib2` by doing an `urlopen` on the query url.

Up to this point, we have indexed 97578 github packages with matching StackOverflow question information processed by feature extraction. We have 14685 icons matched to the github packages.

Ranker: Reordering Information According to Relevance

Whenever a user issues a query, the indexer would return a list of all relevant indexed documents. Very often, this list can be extremely long as the number of indexed documents can be very large. The ranker's job is to rank these relevant results in terms of their relevance to the user's query so that the user can view the most relevant results. There are various models that can be used to implement this ranking, from simple linear models to complex models like neural networks.

There are two sets of items which require ranking in our project and one of them is nested inside the other. Firstly, our project aims to help users in the process of selecting the most suitable package. Hence, there will be a "package object" that is indexed into the indexer, and it will be ranked based on the user's query and the relevant metadata extracted by the feature extraction mechanism. Nested within the "package object" would be crawled web pages that are related to the packages. The ranker would also need to rank those web pages, based on how relevant the web pages are to the query given by the user. These additional information include things like relevant stackoverflow queries and answers and relevant queries from Slant.co.

Since all these web pages and package information are indexed into the Solr indexer, the ranker is implemented in the Solr framework as well. At the moment, a simple linear model is used, ranking the packages based on the features extracted from the crawled data.

Features

The key pieces of package metadata considered in the ranking algorithm are:

1. `repo_description`: A string provided by the Github API that describes what the software package does.
2. `readMe`: A markdown file provided by Github API that goes into a lot more details about how users can go about using the software package. Some of these details may include installation instructions, software dependencies, software features, useful links and more.
3. `homepage_content`: A HyperText Markup Language (HTML) file of the software package's homepage provided by Github API.
4. `slant_queries`: A list of queries (in string format) scraped from Slant.co, a community powered production recommendation website, that has this particular software package listed as one of its recommendations.
5. `stackoverflow_markdown`: Markdown files of Stackoverflow questions and answers that are related to and nested under the software package.
6. `stackoverflow_tags`: A list of tags (in string format) in the Stackoverflow questions and answers that are related to and nested under the software package.
7. `stackoverflow_code`: A list of code snippets (in HTML format) in the Stackoverflow questions and answers that are related to and nested under the software package.

These metadata are chosen because they contain the most information about the package that would aid in the ranking of the software packages.

While there are also many other useful metadata crawled, they are not considered in the ranking algorithm for various reasons. For example, metadata such as the name and language of the software package could be filtered out by parsing the user's query. For example, if the user's query is "python machine learning", our search engine would create a query in a way that causes the indexer to only return software packages that are written in Python, hence the ranker does not need to take these metadata into account when performing the reranking of the results from the indexer. This is similar for other metadata such as name, license, and package manager of the software package and many others. Similarly, with faceted navigation, we will also be able to give the users the ability to rank the results based on numerical metadata such as star count, fork count, last update of the software package and many others.

Scoring Function

Using the above 7 software package metadata, or features, we next calculate a similarity score between the user's query phrase and each of those metadata using the Okapi BM25 algorithm (Robertson, S. E., et al., 1995). The Okapi BM25 algorithm is a bag-of-words (word order of the query and documents do not matter much) scoring function that ranks documents in a similar fashion to Term Frequency Inverse Document Frequency (tf-idf) weighting schemes. This means that for a given query and document, the higher the frequency of the query terms in the

document, the higher the score of the document. However, this is also balanced out by the inverse document frequency value, where the higher the frequency of the query terms in all documents, the lower the score of the document. Intuitively, the scoring of the documents is based on both the frequency of the query terms in the documents and also its rarity amongst all documents.

Given a phrase query Q with terms q_1, q_2, \dots, q_n , the Okapi BM25 score of a document D would be:

$$score(D, Q) = \sum_{i=1}^n IDF(q_i) \frac{f(q_i, D)(k_1 + 1)}{f(q_i, D) + k_1(1 - b(b \frac{|D|}{avgdl}))}$$

$$IDF(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5}$$

where k_1 and b are free parameters, IDF is a function that calculates inverse document frequency of a query term, $avgdl$ is the average document length in the indexed collection, N is the total number indexed documents and n is a function that calculates the number of indexed documents containing the query term. In our implementation, we used the Solr default setting and set $k_1 = 1.2$ and $b = 0.75$. This is because we do not have any labeled data to tune these free parameters for more accurate results.

For features 1 to 4, we are able to directly apply the above mentioned scoring function. However, for features 5 to 7, we cannot directly do so, as it involves child stackoverflow documents. In our implementation, we first treat each stackoverflow document as separate documents, then score each one of them and calculate the average score of all stackoverflow documents nested under a particular package. This average value will be the score of that particular package. In our implementation, all features are weighted equally to derive the final score of the packages. As mentioned before, this is because we do not have any labeled data to tune these weights for more accurate results.

Slop Parameter

Very often, users would not enter queries with only a single term, but rather, enter a phrase query that contains several terms. However, if we naively parse the phrase query as a phrase, the default behaviour of Solr is to do an exact phrase matching with the documents when scoring them. For example, when the user enters “python machine learning”, we ideally want documents with instances of “python” and “machine learning” to be scored highly. Hence, to resolve this problem, we can make use of the slop parameter. The slop parameter is the

maximum allowed distance between query terms for it to be considered a phrase match. Hence, scores of irrelevant documents containing the query terms separately would not be inflated and relevant documents that do not contain an exact phrase match would not be discounted.

In our implementation, we the value of the slop parameter is 50 for all features. This value was derived based on observing the results of several queries that we were familiar with. Again, due to the lack of labeled data, we were not able to perform any rigorous tuning of the slop parameter.

Displaying Information Intuitively with a Dynamic User Interface

The user interface is essential to shaping a new user's first impression and keeping them on our site. If a user can't intuitively understand how to use the defining features of our search engine, or the presentation of information is poor, they will swiftly navigate away from the page.

The user interface has a a number of features that help a user easily navigate the page, the majority of which are listed below:

1. A query that has matching packages will return a list of packages in a carousel.
2. The packages can be further filtered by selecting a language or license.
3. If a valid filter is mentioned in the query, the associated filter will be automatically selected.
4. Clicking on a package from the carousel will display core details about the package.
5. Further details and side-by-side comparison can be performed by clicking the "Compare Side by Side!" button.
6. Stackoverflow posts related to the packages in the original query can help a user find additional details about the packages.

A few open source software packages were used in the development of the interface. Specifically:

1. Django provides much of the web framework.
2. Bootstrap provides components and default styling.
3. jQuery provides interactivity.
4. Slick provides the styling and interaction for the carousel.

Overall, the expected behavior for a user would first include writing a query. From here the user can either select packages or select github links to get some basic information. Afterwards they can narrow down their search further by requerying, or adding filters, both of which will repopulate the page with more refined results until the remaining packages should hopefully suit their needs.

Behind the scenes a query will trigger the following events. First, the indexer is queried for the possible filter values that a query might match on. Any key words that match up with filters are identified and a new indexer request is made to load the related packages, stackoverflow links and reorganize the data hierarchy properly format the table. A brief check is made to make sure there are no redundant stackoverflow links. Then, because images are stored in a separate index, another call to the indexer is made to retrieve the primary images for the packages. Finally the page is rendered, loading the html and static files for viewing by the user.

The number of indexer calls in this procedure are relatively large. In the future, this can be reduced by caching all potential filter values to extract from the query and by indexing the images to be a child value of each package.

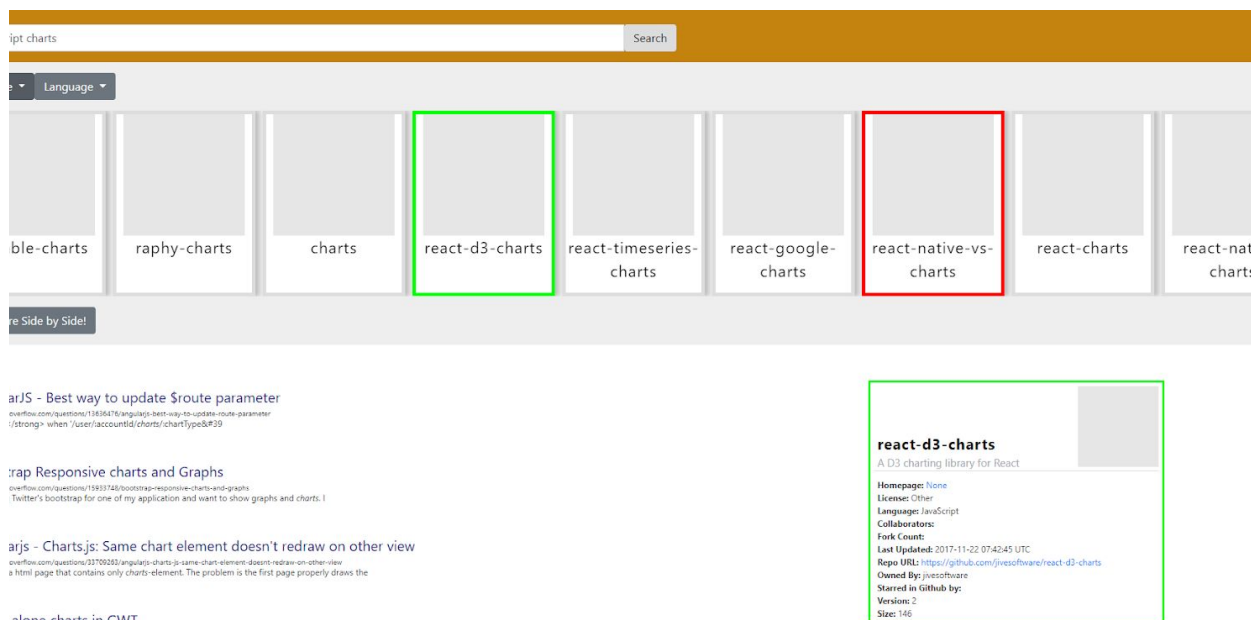


Figure 4 - The resulting user interface for the search engine.

Final Product: How Our Minimum Viable Product Betters the Search Experience for Developers

With the five components mentioned above combined, we are now able to offer a more seamless search experience for developers when it comes to software packages. Rather than combing through pages of search results for a package suitable for a given task, users are now directly presented with a list of relevant packages on a side-to-side comparison fashion.

Upon inputting a search request, users will be shown several the most relevant packages. From there, they can then view and compare all key metadata for these packages, such as publish

date, license, and much more. To further refine the recommended packages, users can also filter by things like number of collaborators, languages and more. By showing these package information that are most important to developers when choosing packages, our platform can save users hours of searching through the comparison and filtering features. Furthermore, to handle queries that are not directly related to software packages, we also provide a list of regular search results similar to those provided by search engines such as Google.

These features mentioned are all enabled by the data that we crawled and processed ahead of time. In order for the platform to be useful in the future, we will need to do continuous crawling and indexing to make sure the information we present are up to date for developers.

Next Steps: The Last Mile Till Success

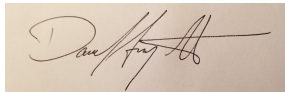
Now that we have completed the first iteration of our minimum viable product, there are certainly many more things we can add to make the platform more robust. Some next steps for the project includes:

1. **Continuous and Exhaustive Crawling:** Currently, only a handful of sites are crawled. Crawling more sites requires more uptime for the crawler, possibly multiple computers crawling, and additional crawling strategies for each website.
2. **Data Pipeline:** Currently, data from crawling is saved in a text file that is later processed by feature extraction. Developing automatic jobs to process raw data would make data generation automated, faster and make continuous crawling possible.
3. **Recognition of User Queries:** In order for our product to be a general search engine, it must be able to differentiate between software queries and non-software queries.
4. **Extraction of Metadata:** There are useful metadata that are pretty difficult to obtain. An example of these would be package dependencies from forums. In order to improve the performance of the search engine, more of such metadata have to be extracted from the crawled data.

In the end, we want to develop a product that can continuous crawl, process and index more data as more and more web pages are generated everyday. This will ensure we provide the most relevant data to our users. This project so far has identified the pain point for many developers when it comes to package search, devised and implemented a prototype to solve this problem, and layed out steps that we can take to further improve the product.

Pledge and Signature

We affirm that we are the sole authors of this report and we give due credit (i.e., use correct citations) to all used sources.



Daniel Avery Nisbet

04/02/2018



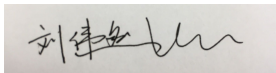
Joshua Yun Keat Choo

04/02/2018



Mengshi Feng

04/02/2018



Weiran (Vivian) Liu

04/02/2018



Yidan Zhang

04/02/2018

References

Apache Solr Feature. (n.d.). Retrieved March 3, 2018, from <http://lucene.apache.org/solr/features.html>

Anderson, M. (2018, January). "Try before you buy" purchase behavior. Retrieved February 11, 2018, from <https://www.thinkwithgoogle.com/advertising-channels/video/consumer-purchase-behavior/>

Brandt, J., Dontcheva, M., Weskamp, M., & Klemmer, S. R. (2010, April). Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 513-522). ACM.

Business Software Alliance. (n.d.). Software Industry Facts and Figures. Retrieved November 4, 2017, from http://www.bsa.org/country/public%20policy/~media/files/policy/security/general/sw_factsfigures.ashx

Fishkin, R. (2017, March 14). The State of Searcher Behavior Revealed Through 23 Remarkable Statistics. Retrieved February 11, 2018, from <https://moz.com/blog/state-of-searcher-behavior-revealed>

Gallardo-Valencia, R. E., and Sim, S. E. (2011). Information Used and Perceived Usefulness in Evaluating Web Source Code Search Results. Retrieved March 1, 2018, from <https://dl.acm.org/citation.cfm?doid=1979742.1979858>

Google Adwords. (n.d.). Keyword Planner. Retrieved November 4, 2017, from https://adwords.google.com/ko/KeywordPlanner/Home?sourceid=awo&__u=6106955674&__c=3038578379&authuser=0#start

Robertson, S. E., Walker, S., Jones, S., Hancock-Beaulieu, M. M., & Gatford, M. (1995). Okapi at TREC-3. *Nist Special Publication Sp, 109*, 109. Retrieved March 30, 2018 from <http://www.computing.dcu.ie/~gjones/Teaching/CA437/city.pdf>

Search Engines in the US, IBISWorld Industry Report 51913a. (2017). IBISWorld, pp.4-34. Retrieved November 2, 2017, from <http://clients1.ibisworld.com/reports/us/industry/default.aspx?entid=1982>

U.S.News. (2017). Software Developer Overview. Retrieved November 4, 2017, from <https://money.usnews.com/careers/best-jobs/software-developer>

Varna Vasudevan. (2018). Paper Prototypes. Retrieved December 9, 2017, from https://www.thedesignexchange.org/design_methods/21

Galarnyk, M. (2016, December 19). AWS EC2: Create EC2 Instance (Linux) – Michael Galarnyk – Medium. Retrieved February 11, 2018, from <https://medium.com/@GalarnykMichael/aws-ec2-part-1-creating-ec2-instance-9d7f8368f78a>

Appendix

Appendix I: Solr Schema Design

```
{
  "collaborators_cnt": [523],
  "fork_cnt": [8388],
  "repo_description": ["One framework. Mobile & desktop."],
  "version_cnt": [30],
  "pm_dependency_cnt": [0],
  "pm_description": ["None"],
  "repo_url": ["https://github.com/angular/angular"],
  "private": [false],
  "watcher_cnt": [34176],
  "created_time": ["2014-09-18 16:12:01 UTC"],
  "last_update": ["2018-03-19 21:37:59 UTC"],
  "pm_name": ["Bower"],
  "owner": ["angular"],
  "repo_dependency_cnt": [0],
  "name": ["angular"],
  "last_push": ["2018-03-19 21:46:19 UTC"],
  "id": "24195339",
  "license": ["MIT License"],
  "path": ["1.git"],
  "homepage_url": ["https://angular.io"],
  "readMe": ["Readme text"],
  "language": ["TypeScript"],
  "star_cnt": [34176],
  "size": [76544],
  "github_id": [24195339],
  "homepage_content": ["content"],
  "repo_url_str": ["https://github.com/angular/angular"],
  "name_str": ["angular"],
  "last_push_str": ["2018-03-19 21:46:19 UTC"],
  "repo_description_str": ["One framework. Mobile & desktop."],
  "pm_name_str": ["Bower"],
  "path_str": ["1.git"],
  "readMe_str": ["![Build
Status](https://travis-ci.org/angular/angular.svg?branch=master)](https://travis-ci.org/angular/angular)\n![CircleCI](https://cir
cleci.com/gh/angular/angular/tree/master.svg?style=shield)](https://circleci.com/gh/angular/angular/tree/master)\n![Br"],
  "language_str": ["TypeScript"],
  "pm_description_str": ["None"],
  "created_time_str": ["2014-09-18 16:12:01 UTC"],
  "last_update_str": ["2018-03-19 21:37:59 UTC"],
  "homepage_url_str": ["https://angular.io"],
  "_version_": 1595434507606949888,
  "owner_str": ["angular"],
  "homepage_content_str": ["<!doctype html><html><head><meta charset=\"utf-8\"><title>Angular Docs</title><base
href=\"/\"><meta name=\"viewport\" content=\"width=device-width,initial-scale=1\"><link rel=\"icon\" type=\"image/x-icon\"
href=\"/favicon.ico\"><link rel=\"icon\" type=\"image/x-icon\" href=\"\">"],
  "license_str": ["MIT License"]},
  "_childDocuments_":
  [
    {

```

```

"up_vote_count":["191"],
  "path":["2.stack"],
  "body_markdown":["content"],
  "view_count":["153599"],
  "answer_count":["17"],
  "tags":["['android', 'video-capture']"],
  "creation_date":["1300733559"],
  "last_edit_date":["1300771342"],
  "code_snippet":["content"],
  "title":["Capture Video of Android's Screen"],
"link":["https://stackoverflow.com/questions/5382212/capture-video-of-androids-screen"],
  "id":["24195339-2327"],
  "body_markdown_str":["content"],
  "tags_str":["['android', 'video-capture']"],
  "path_str":["2.stack"],
  "view_count_str":["153599"],
  "last_edit_date_str":["1300771342"],
  "answer_count_str":["17"],
"link_str":["https://stackoverflow.com/questions/5382212/capture-video-of-androids-screen"],
  "up_vote_count_str":["191"],
  "code_snippet_str":["content"],
  "_version_":1595434507606949888,
  "creation_date_str":["1300733559"]},
  "_childDocuments_":
  [
    {
      "path":["3.stack.answer"],
      "up_vote_count":["10"],
      "answer_id":[5382516],
      "is_accepted":[false],
      "last_activity_date":[1300735367],
      "body_markdown":["\r\nYes, use a phone with a video out, and use a video recorder to capture the
stream\r\n\r\nSee this article
http://graphics-geek.blogspot.com/2011/02/recording-animations-via-hdmi.html\r\n"],
      "id":["5382516"],
      "down_vote_count":[2],
      "creation_date":["1300735367"],
      "score":[8],
      "body_markdown_str":["\r\nYes, use a phone with a video out, and use a video recorder to capture
the stream\r\n\r\nSee this article
http://graphics-geek.blogspot.com/2011/02/recording-animations-via-hdmi.html\r\n"],
      "path_str":["3.stack.answer"],
      "up_vote_count_str":["10"],
      "_version_":1595434507606949888,
      "creation_date_str":["1300735367"]},
    }
  ]
}
]
}

```


Appendix II: Paper Cutouts for Low-Fidelity Feature Testing

The following cutouts were brought to potential end users. Users could drag a potential query such as “how do I make a JS chart”? To the query bar, and results come up. By selecting filters, they can decrease the number of packages available in the carousel. This was performed by folding the piece of paper to hide different packages. Finally, the side-by-side comparison table was expanded based on what packages a user selected to compare.

how do I make a JS chart? x

Jcharts

Charts +

Compare

Tutorial on Creating Charts
<https://canva.js.com/>

Rating	4.5 / 5	4 / 5	2.5 / 5
Avg Forum Response Time	2 hrs	9 hrs	2 days
Avg Github Issue Response Time	1 days	2 days	12 hrs
License	MIT	GNU GPL	MIT
Doc. Quality	4 / 5	4.5 / 5	1 / 5
Last Modified	Dec 2016	May 2017	Nov 2017
Code Snippets	Code Snippets	Code Snippets	////
Download	Download	download	Download
Get Started			

Filter

Language

- ☒ JavaScript
- ☐ Python
- ☐ PHP
- ☐ Ruby

License

- ☒ GNU GPL
- ☒ MIT
- ☒ APACHE

Rating

- ☐ 4.5 +
- ☐ 4 +
- ☐ 3.5 +
- ☐ 3 +
- ☒ 2.5 +

