Improving Image Search Relevance Through Context Analysis and Deduplication

Johnny Bui (172A)
Mentors: Randal Moss and Garrett Johnston
Jet Propulsion Laboratory
La Cañada Flintridge, California, USA

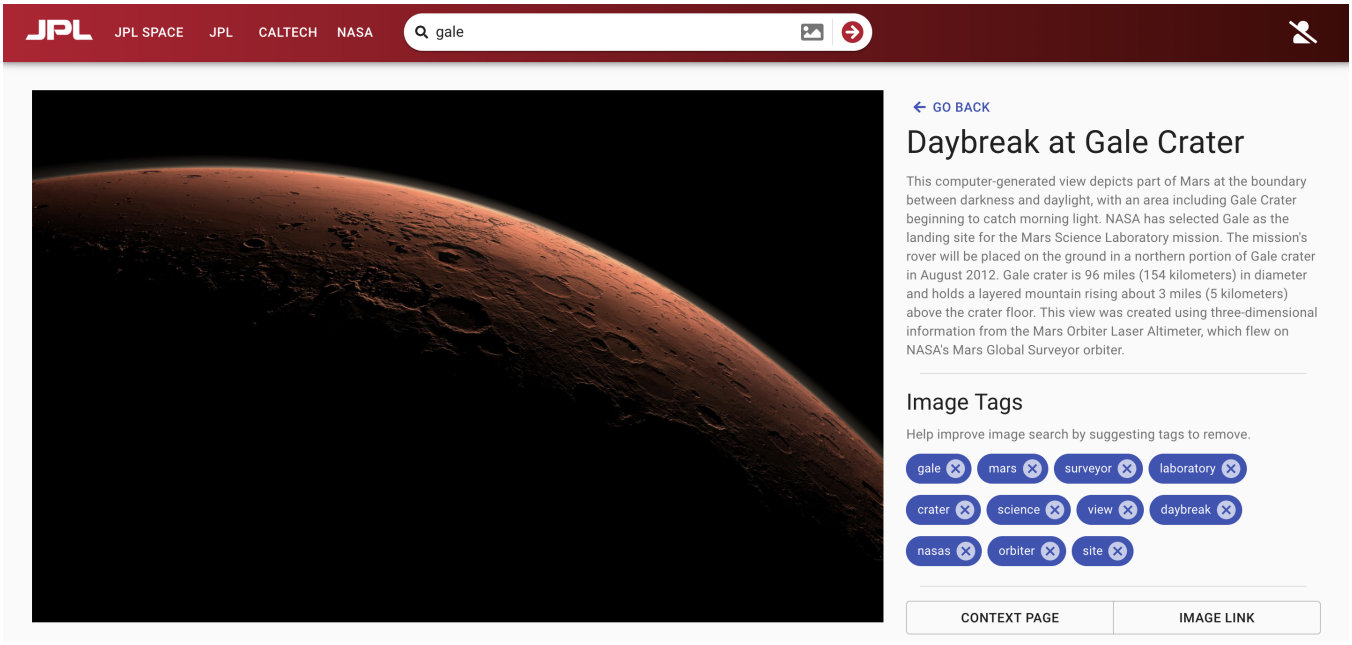

Figure 1: The detail page of the new JSearch showcases contextual information extracted from the image's source webpage.

ABSTRACT

In order to facilitate effective internal and external communication, many JPL employees and affiliates rely on lab-wide image search to return relevant images and graphics to complete their work. However, improving the relevance of image search results is challenging because most pictures do not have captions. Here we describe a performant library, Native Generation of Indexable coNtext for Images (NaGINI), that is capable of generating image tags using contextual cues from the webpage which the image is embedded. Natural language processing (NLP) techniques applied to the text from a source webpage summarize long articles and generate a list of topic keywords which are given increased importance in the search engine. We also implement image deduplication functionality within NaGINI through the use of an image-focused Locality-Sensitive Hash (LSH) function and an innovative database structure that is optimized for queries involving binary hamming distance. Future work is required to improve the output of the NLP techniques and filter out noise resulting from webpages that do not have significant textual elements.

KEYWORDS

image search, deduplication, contextual tagging, natural language processing, hamming distance

1 INTRODUCTION

In the age of virtually-instantaneous search powered by tech giants like Google and Microsoft, users are increasingly reliant on search engines that can quickly respond to search queries with relevant results. This is no different at the Jet Propulsion Laboratory, where JPL-ers constantly rely on search to quickly locate details about other projects and people to propel their own research forward. To meet their needs, the Office of the Chief Information Officer (OCIO) at JPL established the 172A - Collaborative Engineering And Information Capture group to develop and maintain an internal search engine called JSearch.

Since 2014, JSearch has been deployed to JPL-ers through JSpace, the homepage for JPL lab members connected to the internal lab network. Users begin a query by typing into the search bar found at the top of the page. Autocomplete suggestions automatically appear in a drop-down and the search results that populate the page are updated with each keystroke. Users can then choose the type of results they wish to view: webpages, documents, images, videos, or people. In the month of July 2019, over 3,000 unique users used JSearch out of the approximately 6,000 full-time employees [1]. In this report, we will focus on the current functionality of the image search engine and how it can be improved, both in regards to result relevance and user experience.

2 BACKGROUND

There are many pieces of software that the JSearch team rely on every day to provide a reliable and performant search experience to users across the Lab. Improving image search will require that the solution be concurrently interoperable with these frameworks, and optimally be integrated directly into the pre-existing search architecture.

Software and Architecture

JSearch image search, alongside web and document searching, is powered by a cluster of servers running the open-source search and analytics engine *Elasticsearch*. *Elasticsearch* stores schema-free JSON documents (which can be thought of as dictionaries with key-value pairs) that represent searchable content. Fields are analyzed to enable specific searching behavior, and the search engine returns a list of documents when a client performs a query. Interfacing with *Elasticsearch* is performed either through HTTP calls or using high-level client libraries for programming languages such as Python.

In order to populate the *Elasticsearch* cluster with web pages, documents, and images, we rely on a JPL-specific implementation of *Scrapy*, an open-source Python web crawling framework. *Scrapy* is able to methodically index the web resources available on the JPL network by following links embedded on each page, similar to how a searching algorithm traverses a graph. Every time it encounters a new resource, it sends the item through a pipeline which extracts data and performs analysis before indexing a document in *Elasticsearch*.

In order to improve performance and uptime while reducing operational complexity, software such as *Elasticsearch* and *Scrapy* is deployed to clusters and is managed by *Kubernetes*, a container-orchestration system that allows for simple deployment, scaling, and management of horizontally-scaled applications across a cluster of server nodes.

Current Process

Currently, image indexing and searching proceeds through the following process:

(1) *Scrapy* crawls JPL websites for webpages and other resources by recursively following links.
(2) When a link to an image is found, it is indexed into *Elasticsearch* with two fields: the URL and link text to that image (see Listing 1).
(3) When a user enters a query into JSearch and filters for images, *Elasticsearch* will query the search index for images that have that search query in either the URL string or the link text.
(4) Documents that have the query text will be shown to the client sorted based on two metrics: Term Frequency and Inverse Document Frequency (TF-IDF). Results that have the query term multiple times are ranked higher, but this ranking boost is offset based on the total number of documents that contain the query term.

Limitations

There are multiple severe limitations to the current process by which images are discovered and indexed. These design flaws result in many images not being indexed and reduce the relevance of search results.

First, only images that are **linked** through an HTML `<a>` tag are indexed by *Scrapy*. Displaying an image to a visitor only requires that the image be embedded using an `<img>` tag, so many images that are embedded in a page that are never added to *Elasticsearch*.

In addition, images that are linked are only saved with their URL and corresponding link text. Take a look at an example snippet of HTML:

Listing 1: The link text is "here".

```
<p>
  To view the image, click
  <a href="/assets/mars.jpg">here</a>.
</p>
```

In this case, the image `mars.jpg` will only appear in search results if the user searches for "assets", "mars", or "here". This is incredibly limiting, especially if the website designer chose to give the image a *less helpful* name, like `planet4.jpg`. This isn't purely theoretical however. In **Figure 2**, notice how the user does not get any results when the query "robot" is given to the image search engine. There are in fact images of robots in the search cluster, but none of them have the word "robot" in their URL or link text.

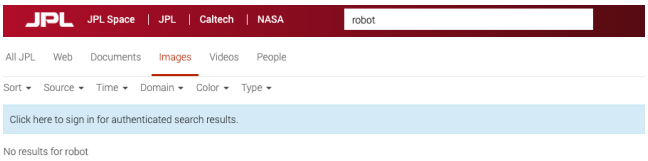

Figure 2: No results appear when a user searches for "robot" images using JSearch.

3 DESIGN PRINCIPLES

In order to develop a solution that can address the limitations of the current system, we must first discuss the design principles that are important to consider during implementation. The main goal of our solution is to improve the **accuracy** of image search results, specifically by increasing the overall relevance of the images returned. While relevant results are important, we must also consider the **performance** implications of our solution because increased query times are detrimental to user experience. And finally, our solution must be able to easily **integrate** with other pre-existing components of the search architecture to prevent the need to re-tool the current process.

Accuracy

In the context of image search, accuracy is maximized when the amount of relevant results returned is maximized and the amount of irrelevant results is minimized. Users of the image search engine should be shown as many images as possible that match the query, but if a significant amount of results do not match the query, the user's experience decreases dramatically since they have to sift through the results manually. In a perfect world, there would be no irrelevant results. However, we accept a small tradeoff between increasing the number of irrelevant results in exchange for returning more relevant results.

Performance

There are two bottlenecks in the image search process where performance must be considered: at crawl-time and at search-time. When the web crawler *Scrapy* encounters an image, processing must take place in order to increase the amount of indexed content saved in the *Elasticsearch* cluster. Although crawling performance is not directly-connected to the end user experience, excessive processing time would result in a decreased crawling throughput, resulting in outdated and possibly missing images.

Performance is far more important at search-time, when users enter a query and wait for the server to respond. Today, most users expect sub-second response time, with users navigating away or reloading after a few seconds of non-responsiveness.

When designing our solution, we must primarily focus on preventing large increases to query times, while keeping in mind that crawling time must not be excessive.

Interoperability

JSearch has built up its infrastructure since 2014, and as such, many tools and technologies have already been decided upon. Our solution should be able to operate simultaneously alongside existing search infrastructure without breaking any functionality. Preferably, we aim to integrate our solution directly in the existing toolchain to minimize implementation friction.

4 NAGINI

During the indexing of an image, we must augment the stored document with textual descriptors so that *Elasticsearch* can retrieve relevant image results in response to a text query. *Elasticsearch* is a text search engine, so improving image search relevance is synonymous with improving the relevance of supplemental text in an *Elasticsearch* image document. But where can we find relevant text about an image?

Finding Relevant Text

There are two distinct data sources of an image embedded/linked in a webpage: the image data and the contextual information found in the webpage where the image is referenced. While image data can be analyzed using techniques like machine learning to identify the objects within an image, it relies on having access to a large accurate training data set and a significant amount of computing power. While the second requirement can be arranged, it remains a challenge to obtain a pre-tagged data set of images that is JPL-specific.

Because many off-the-shelf image captioning models such as *Apache Tika ImageCaption* are trained using a consumer data set, they are woefully inadequate at identifying objects that are more common at JPL (see **Figure 3**). Beyond that, image data is also useful for extracting text from an image that contains rasterized text (such as graphs) through the use of optical character recognition (OCR).

With image data being a dead end for relevant text extraction (besides OCR), we turn to the other source of data: the context webpage. Often, images are embedded into webpages that pertain to the topic of the image. For example, pictures embedded on a webpage about Jupiter are likely to be of Jupiter. But how do we leverage this new data source?

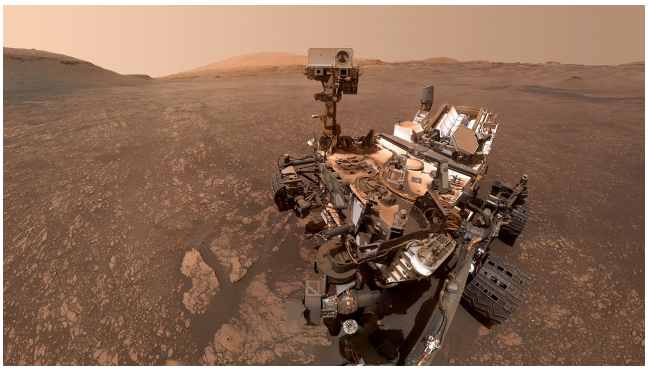

Figure 3: Apache Tika incorrectly captions this image of the Mars Curiosity rover as "a man riding a motorcycle on top of a sandy beach." Photo courtesy JPL/NASA.

Utilizing Natural Language Processing

Our new goal is to extract the important ideas out of a webpage in order to make assumptions about the contents of an image. The solution to our problem is also an active area of research: Natural Language Processing (NLP).

We first must be able to extract the main text from an HTML webpage in order to reduce the amount of noise from irrelevant elements. To accomplish this, we can use an HTML parser such as `beautifulsoup4`. We must then determine important keywords of the extracted text using NLP. This is done by removing stop words (words that add no meaning to a sentence), analyzing sentence structure, and inferring word importance from frequency and position.

Putting It All Together

To combine all of the ideas into a component that abstracts away complexity, we introduce *NaGINI*, a backronym for *Native Generation of Indexable coNtext for Images*. There are two main components of *NaGINI*: `NaginiContext` for context analysis and `NaginiImage` for image analysis.

`NaginiContext` is a thin wrapper around `newspaper` [1], an open-source parsing and NLP library. It checks the validity of context pages based on the URL, downloads the page's HTML, parses the page for the page title and most prominent image, and performs NLP in order to generate a list of keywords and a shortened summary of the webpage.

`NaginiImage` is a class that performs validation on an image URL before downloading and processing it with Tesseract OCR[2]. It also performs functions relating to image deduplication explained in the next section.

[1]https://github.com/codelucas/newspaper
[2]https://github.com/tesseract-ocr/tesseract

5 IMAGE DEDUPLICATION

During the development and testing of *NaGINI*, it became obvious that image duplication was a significant problem on many JPL websites. Image duplication can be seen when the "same" image is served over different URLs. However, while some images were bit-wise identical, many other duplicates were "near-identical" images that were simply resized. This means that while the images are visually-identical, the binary data of each image is actually very different.

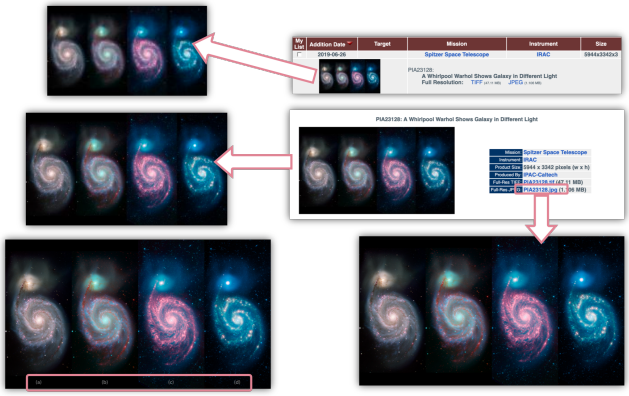

Figure 4: Four near-identical versions of the same image were found on Photojournal. Each version was either resized or had additional annotations.

Challenges to Traditional Deduplication Methods

A naïve-approach to detecting duplicate images is performing pairwise comparison of each image's binary data. However, while binary comparison would be capable of detecting images that are renamed, it is computationally-expensive to perform so many comparisons on an unbounded-size input.

To solve this, we apply a hash function to each image's binary data and compare each hash. A hash function maps an arbitrary-size input to a fixed-size output. One example of such a hash algorithm is the Secure Hash Algorithm (SHA), a widely-used set of cryptographic hash functions. One variant of this family, SHA-256, could map each image into a 256-bit binary string.

However, if we want to detect visually-identical images that are resized or in a different file format, hash functions such as SHA-256 fail us because the hash output will almost-certainly never match (or even be close) if the hash input is different. This behavior is one property of cryptographic hash functions known as collision-resistance. While this property is important in many use cases of hashing, it is not useful for finding near-duplicate images.

We now know that we need a different method of hashing that is capable of generating fixed-size outputs that are similar if the variable-size input is similar.

Locality-Sensitive Hashing

Since we have determined that cryptographic hashes don't fit the bill, it is clear that the problem must be approached using a different method. Locality-sensitive hashing (LSH) is an algorithmic technique where similar input data is mapped into a similar output hash with high probability. Early research of such an approach was published by Sadowski, Caitlin, and Levin in their article introducing `simhash`, a document-focused hash function where "similar files should map to very similar, or even the same, hash key" [3].

Perceptual Hashing

Given what we know so far, we now understand the requirements necessary to build an image deduplication system capable of efficiently and effectively detecting near-identical images. The remaining implementation details are described by Zauner, who implements a C++ library that integrates multiple steps in order to create such an LSH specific for the image domain [4].

In short, Zauner's perceptual hash function performs the following steps:

(1) Extract the pixel data from the image to normalize any differences in input file format and save into a 2-dimensional array.
(2) Convert the image color space to grayscale to reduce input data size while preserving most semantic data.
(3) Apply an averaging (blur) filter to the image to reduce high-frequency patterns that are unnecessary for image comparison.
(4) Resize the image to a 32-by-32 pixel square to eliminate differences in aspect ratio (stretching or compression along an axis) from affecting image comparison.
(5) Perform a Type-II discrete cosine transform and isolate the top-left 8-by-8 set of coefficients that represent the most low-frequency patterns of the image.
(6) String the matrix of coefficients together into a one-dimensional array, determine the median m of the coefficients, and map the coefficients x_i to binary using the following function:

$$f(x_i) = \begin{cases} 0 & x_i < m \\ 1 & x_i \geq m \end{cases} \tag{1}$$

(7) The resulting binary array is the 64-bit perceptual hash of the image.

Defining Duplicates

Once the image has been hashed into a 64-bit binary string, the final piece of the deduplication puzzle remains: using the computed hash to locate similar images in a collection. In the remainder of this report, we define similar images as images that are visually-identical, but do not necessarily have the same binary data. It is possible to accomplish this in linear time by comparing the `query` image's hash with the hashes of every other picture. However, knowing that our database will store tens of thousands of images, can we improve the scalability of our image search by finding a sub-linear search algorithm?

To quantify similarity based on our perceptual hash, we define the hamming distance of the two hashes to be inversely-proportional to the probability that the two images are near-duplicates. Hamming distance is the number of bit positions that two equal-length binary strings differ. In terms of bit operations, the hamming distance d of two binary strings A and B with equal length l is given by

$$d(A, B) = \sum_{i=0}^{i<l} A_i \oplus B_i \tag{2}$$

where A_i is the i^{th} bit of A.

More formally, given binary perceptual hashes A and B from two images, the probability P that they are near-identical is given by

$$P(A, B) = 1 - \frac{d(A, B)}{64} \tag{3}$$

To ensure that we do not incorrectly misclassify images as duplicates (since duplicates would be removed from search results), we attempt to optimize for reducing false positives at the cost of false negatives. Based on preliminary testing, resizing and small annotations do not change the perceptual hash in more than three positions. Therefore, when searching for duplicates of a given image with hash h, another image with hash h' is considered near-identical if $d(h, h') \leq 3$.

The astute reader will notice that the image duplicate searching process takes linear time for each query image due to the fixed-length bitwise comparison to each hash in the collection. While the performance of this process is not critical to user experience since it is performed at crawl-time, indexing time will suffer greatly as we load the image database with tens of thousands of images. The road to sub-linear performance isn't straightforward, since throwing a hash table at the problem will still result in poor performance due to the number of lookups required. For a hamming distance of up to three, $\binom{64}{1} + \binom{64}{2} + \binom{64}{3} = 43744$ lookups are required for a given query hash. This is due to the number of possible positions that bits can be flipped from the 64-bit hash.

Achieving Sub-linear Hamming Distance Search

We have introduced two different methods of searching a collection for hashes that have a hamming distance of three or less: linear-time search using bitwise operations and using a hash-table to lookup all possible hashes. In the former, performance is limited by the comparison on the linear number of hashes, and in the latter, the amount of queries is constant

but has a significant constant factor that is unsuitable for our use case.

In an article by Manku, Jain, and Sarma, they lay out the framework for solving this problem by chunking the binary data [2]. The intuition behind this approach is to both minimize the amount of comparisons and the amount of queries by splitting the binary string into chunks that are a subset of the original.

An example would best illuminate the ingenuity of this approach. Say that we split our 64-bit binary string into five chunks of lengths 13, 13, 13, 13, and 12. If we want to find all binary strings in a collection that are at most three hamming distance away, two of the five chunks **must** be identical. While we don't know which two chunks would be identical, we can query the collection for all $\binom{5}{2} = 10$ combinations. This can be done by creating 10 different hash tables that correspond to each combination of matching chunks and map the 25 or 26-bits of data to a list of all documents with those matching chunks.

While this increases the storage size by a factor of up to ten, it greatly narrows down the candidate hashes, since any hash that does not have at least two identical chunks **must be** over three hamming distance away.

Note that while all of the eliminated hashes had a hamming distance greater than three, it is not necessarily true that all remaining hashes have a hamming distance of less than three. If, for example, the first ten bits of a hash were different, only one chunk would be different and four chunks would be identical. Therefore, we must proceed with linear bit-wise comparison for all candidate hashes in order to filter out candidates that have a hamming distance of more than three from the query.

Although we described the process in detail above, we purposefully omit the explanation for how this technique greatly improves the performance of binary hamming distance search. Further theory can be found in [2], which is the source of this implementation. An evaluation of the real-world performance is provided in section 7.

Content-Based Information Retrieval

By using the perceptual image hash algorithm to reduce the dimensionality of the query image data, we are able to rapidly locate near-identical images from a database with high accuracy. Besides deduplication, another use of this functionality is to implement a rudimentary reverse-image search feature where a user uploads an image and the search engine returns all recorded instances where the image appears on the network.

Implementation of this feature was straightforward because all of the groundwork has already been laid out in previous sections, most importantly the hashing of images in the database. The impressive performance of the sub-linear hamming distance search allows us to query a hash from database of millions of documents in less than half a second (full performance data is found in section 7).

Implementation

The image deduplication function is implemented as a class function within *NaginiImage* by using the `imagehash` Python library available through `PyPi`. Images are opened using `Pillow`, hashed into a 64-bit hex string, and split into 13, 13, 13, 13, and 12-bit hex chunks, respectively. They are then stored in Elasticsearch with the following schema:

Listing 2: Snippet of Elasticsearch schema

```
{
  ...
  image_hash: "c73b3c3b4949494b",
  image_hash0: "18e7",
  image_hash1: "cf0",
  image_hash2: "1da4",
  image_hash3: "1494",
  image_hash4: "94b"
}
```

Notice that we store both the full image hash and the image hash chunks in hexadecimal form. This is to both preserve leading zeros in the full hash while also minimizing storage space by avoiding the need to store a 64-character string with only zeroes and ones.

With *Elasticsearch*, we do not have to query all possible combinations of two chunks that could be identical. If you recall, this was necessary because we know that at most three chunks can be be different but not which three chunks. Luckily, *Elasticsearch*'s query language allows us to search for documents where two out of five conditions are true using the `minimum_should_match` property on a `bool` compound query. Once all candidates from this query are returned, we use the `image_hash` field to determine the hamming distance between the query and the candidates to filter out hashes that exceed the hamming distance threshold of 3.

If there is a match, we must determine whether to mark an image as a parent or a duplicate. Parents are considered the "original" image, and all near-identical images to the parent are considered duplicates. Using a ranking function, we mark the parent as the image that has the highest resolution (as defined by the total number of pixels). If there is a tie, we arbitrarily break the tie by marking the most recent image indexed as the parent. All duplicates are marked with a flag and an additional field that specifies the URL of the parent to that image to allow for duplicate filtering at search-time.

With this implementation, search results will hide all images that are marked as duplicates by default. In addition,

each parent will be able to query the database for its duplicates if an alternate size is requested from the user.

6 USER INTERFACE

With tagging and deduplication completed, the final weeks of the project were focused onto the development of the user interface. The current user interface is not able to take advantage of the new fields that have been generated by NaGINI, hide duplicates, or hook into the reverse image search functionality.

To address these issues and also perform the first overhaul of the user interface since 2014, we decided to begin development on `JSearch 2.0`. By starting fresh, we are able to implement JSearch to best take advantage of new functionality and give search a modern look.

The new search frontend is built on `React 16.8` and uses components from the `material-ui 4.2` library. Substantial changes between the old and new interface include:

- A new, larger search bar that includes a button for reverse image search
- Results being displayed as cards to increase the amount of information users receive at a glance
- A details page for each result that gives users more information about the image and links to both the image URL and the source page.

7 EVALUATION

While designing and implementing *NaGINI*, image deduplication, and the user interface, optimizations and tradeoffs were made by prioritizing the design principles described in section 3. In this section, we discuss how accuracy, performance, and interoperability informed the function and design of image search.

Contextual Tagging

NaGINI's primary responsibility is to serve as a layer of abstraction around disparate libraries that fetch web resources, validate responses, and analyze webpages & images for relevant text to index. In general, our evaluation of *NaGINI*'s accuracy is more lenient on having a high rate of false positives (extracted tags that are irrelevant) rather than having a high rate of false negatives (tags that should be extracted but are not). The intuition behind this evaluation decision is because it is a superior search experience to have to scroll through some irrelevant images before finding a relevant image rather than the relevant images not appearing at all.

If a user requests images of a "robot", the old image search would not return any results (refer back to Figure 2). However, as seen in Figure 5, which implements the new image search, images that show robots are now returned to the user, even if the image URL does not contain that word. Implementing *NaGINI* has improved the relevance of image search results, and thus accuracy has improved.

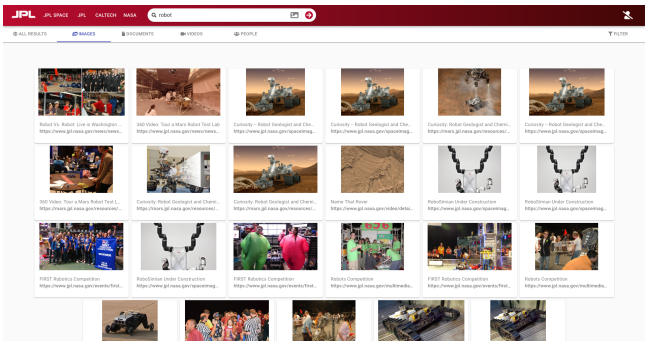

Figure 5: This screenshot shows the many results that are returned by the new image search platform in response to the query "robot".

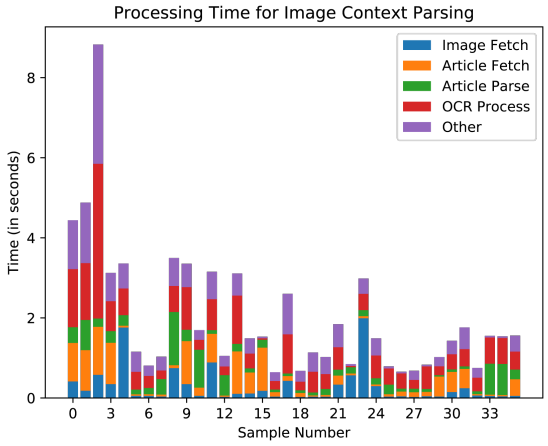

Figure 6: This graph visually represents the amount of time taken by each step in NaGINI. Each bar represents an image and context page that was fetched and analyzed.

We next consider the performance of *NaGINI* during the indexing of new images. While it is not incredibly important to optimize contextual tagging because it occurs at crawl-time, we would preferrably like to minimize the amount of crawling instances needed to crawl the network by increasing the crawling throughput. As shown in Figure 6, the additional time that is required for fetching and analysis is at most a few seconds, which is in line with the current amount of time required to crawl a webpage using *Scrapy*. While the performance of *NaGINI* is nothing to admire, it functions well with an acceptable speed.

In order to implement *NaGINI* into the current search indexing pipeline, it had to be integrated into the JPL web

crawler *Scrapy*. Because *NaGINI* is packaged as a library, integration was simple and only required additional imports and function calls in order to generate the contextual information. Scrapy is also able to pass in the HTML content of the context webpage in order to eliminate a redundant network transfer, improving performance considerably. Because of this, it is clear that *NaGINI* is designed to be highly interoperable with *Scrapy*.

Image Deduplication

The image deduplication functionality described in this report was implemented for a test crawl of the website https://photojournal.jpl.nasa.gov. On this website, duplicate images are prevalent because of the usage of thumbnails and annotated images. During the test run, over 6,000 out of the 19,000 images indexed were marked as duplicates, and after manual checking, it is clear that the deduplication is working as intended.

The deduplication performance is extremely impressive because of the sub-linear time algorithm used to query the database for hashes within a certain hamming distance. Because we decided to store image hashes alongside images in Elasticsearch, we are able to utilize the pre-existing search infrastructure and prevent data fragmentation. As shown in Figure 7, the time to add a new document to the database is negligable compared to *NaGINI* analysis. In Figure 8, querying a database of 1 million documents for hashes that are at most three hamming distance away from the query is achieved in less than half a second per query. This performance is exceptional and meets the needs of JSearch for both crawl-time deduplication and search-time reverse image search.

Because our deduplication process does not require additional infrastructure and minimal changes to the code base, interoperability with existing components is outstanding. Altering *Scrapy* to check for duplicates is as simple as adding an additional pipeline step within the crawler. This step will set a flag in the document if the current image is a duplicate to an image already in the database.

Although reverse image search relies on a Python proxy server in order to accept image files and return the perceptual hash, this is not required for basic image search functionality, and is relatively easy to write and maintain.

User Interface

The new JSearch user interface is intended to replace the current search webpage. It relies on an HTTP proxy to mediate communication between end clients and the *Elasticsearch* cluster. For the purposes of prototyping, a simple proxy server has been developed using `Flask`. However, for production environments, a rewrite of the proxy server is recommended to optimize performance, since the `Flask` server will likely become the bottleneck during periods of high search volume.

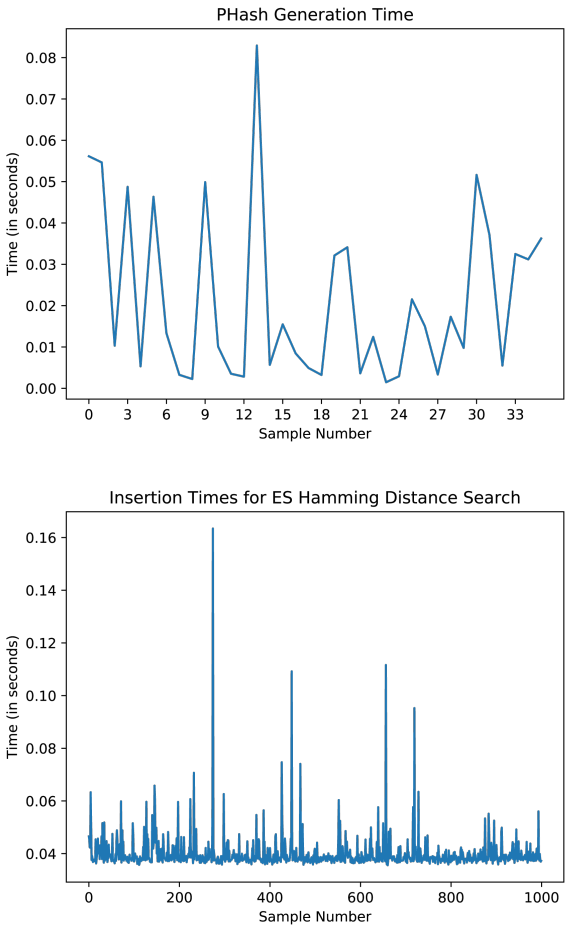

Figure 7: These graphs depict the time needed to generate and insert image hashes into an Elasticsearch cluster using a diverse sample of JPL images.

Client browser performance is heavily impacted when more than 100 image results are loaded at the same time. It is reasonable to conclude that this is because the images that make up the search results are original size, and not thumbnails. This results in high compute, memory, and network needs during loading and scrolling. Some recommendations on how to fix this are discussed in section 9.

8 STATUS

As of the publication of this report, the tagging tool `NaGINI` and the image deduplication engine have been implemented into the pre-existing search indexing pipeline and published on the JPL source control system. These two elements are ready for deployment to production, and they are backwards

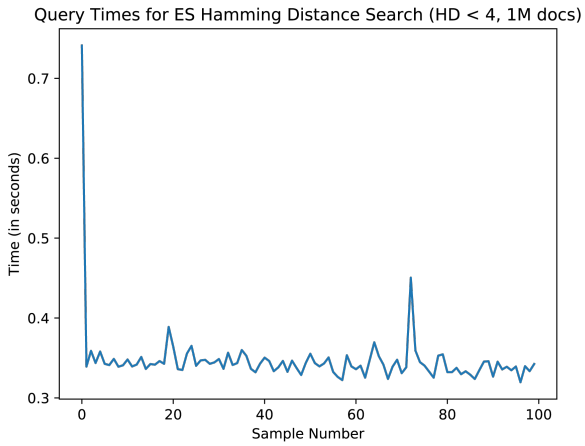

Figure 8: This graph shows the time it takes to query a database with 1 million documents for all hashes that are at most three hamming distance away.

compatible with the current user interface. The new user interface requires additional development to improve its performance and feature set, but is completely functional and serves as a minimally-viable product.

9 FUTURE WORK

While great strides have been made in improving image search, future work is required to implement additional features that build on the work described in this report. For *NaGINI*, the output of the natural language processing (NLP) library should be able to prioritize certain keywords (such as proper nouns) that improve the relevance of search. In addition, context pages should be scanned to determine if significant textual elements do not exist to prevent the NLP library from outputting useless gibberish. Photojournal catalog pages exhibit this problem.

While our current image deduplication process is able to detect resized images, it is not as sensitive to near-duplicates that have small crop differences. This can be solved by migrating to a different perceptual hashing algorithm or increasing the hamming distance threshold. Reverse image search should be able to return images that depict the same object, but are not necessarily near-duplicates. This image similarity platform is currently being developed for JSearch by Sohini Kar, and will require integration with the new JSearch interface.

Before sending the new JSearch interface to production, a performant and production-ready proxy service needs to be developed and deployed in order to handle demanding production traffic. In addition, features such as sorting and filtering have been rigged into the design but not integrated into the logical components. Lastly, other categories of search such as webpages and documents must be implemented into the search page for consistency.

10 CONCLUSION

Today, many problems with the JSearch image search platform prevent users from quickly retrieving relevant images, resulting in an unsatisfactory user experience. In this report, several improvements have been identified and implemented in preparation for `JSearch 2.0`, which will implement contextual image tagging, image deduplication, and a modern user interface that takes advantage of the expanded feature set.

Contextual image tagging is enabled through the use of the *NaGINI* library, which is integrated closely with *Scrapy* to tag images based on NLP analysis on the source page which an image is tagged from. Image deduplication relies on a locality-sensitive hashing function that quantifies image similarity based on hamming distance. This hamming distance search is possible using *Elasticsearch* because of a filtering technique called binary chunking. And lastly, all of these features are visible to end users through a sleek and modern interface built using React.

ACKNOWLEDGMENTS

Without certain people, the findings of this report would have never seen the light of day. Special thanks to Randy Moss for entrusting me with a project that is both awesome and has a substantial impact on JPL. Garrett Johnston was a fantastic guide in helping me aim the trajectory of my project so that I make the most out of my internship. Crispus Mwaura gave great insight into the best way to structure my *NaGINI* code for testability. Kevin Hwang was a great resource to bounce some of my outlandish ideas off of. And of course, the other Woodbury interns, without whom this report would have been done 2 week earlier.

REFERENCES

[1] National Aeronautics and Space Administration. 2019. Jet Propulsion Laboratory 2018 Annual Report. (April 2019). https://www.jpl.nasa.gov/report/2018.pdf

[2] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. 2007. Detecting near-duplicates for web crawling. In *Proceedings of the 16th international conference on World Wide Web*. ACM, 141–150.

[3] Caitlin Sadowski and Greg Levin. 2007. Simhash: Hash-based similarity detection. *Technical report, Google* (2007).

[4] Christoph Zauner. 2010. Implementation and benchmarking of perceptual image hash functions. (2010).