SaleTags

Abel Rodriguez

Abstract
SaleTags is a web application that allows men to perform a natural language search for men's clothing on sale at various retail stores. SaleTags re-creates the experience of browsing through a mall. Search men's clothing at discounted prices and click on a product that is linked to an online store where it can be bought.

Introduction
Nowadays, since we are living in a more advanced generation, we rely so much on technology and gadgets that even the simplest things, like shopping, can be done with the aid of Internet and computers. It avoids getting in the car driving to the mall, finding parking, and walking to the stores, which may take upwards of an hour.

However, online sale shopping can be tiresome task. When browsing the Urban Outfitters site to shop for men's sale, we must scroll through 1255 products. Using the filtering options provided by the website only helps narrow down the list to 660 products. On the other hand, the Banana Republic sale's page returns 266 items. However, when using the filtering options to display only shorts on sale, a mere 8 items are returned. A user must either spend hours navigating through dozens of pages or be limited to a few products that may be unsatisfactory. In addition, some sites such as HM.com do not provide a search box. When looking for an item in particular, we are forced to check every page. This issue has been brought up to HM, yet the only responses received state "a filter option can be found in each product category. Hope this helps!" Well, not really, but thanks for responding.

SaleTags attempts to solve the need for a personalized online shopping experience that makes bargain shopping easier. By indexing over 3000 thousand products from several retail stores, it eliminates the responsibility to bookmark each site and check in on every single one of them on a regular basis. It allows the user to enter a query for a product, using regular spoken language. The application will then return the 50 most relevant products matching the query. Furthermore, the application offers shoppers the ability to view and compare pricing information for similar products, thereby enabling users to ultimately find the right product from the right merchant at the lowest price.

Related Work
An existing system to my project is Wanelo, a site that lets users browse unique products sold on other sites. It is similar in that it brings together stores, products, and people into a single platform. However, it appears to have a few flaws in its search engine implementation. As shown in Figure 1, when a user enters the query "dress solid blue shirt," 8 products are returned. One would assume the first product listed would be the most relevant, instead, the product least expected to be relevant is displayed first. In addition, the implementation does not support a more natural language search. For example, a user might want to search "black jacket with front pockets" or "dressy solid blue shirt." As can be seen from Figure 2, no results are returned. One additional letter in the query can give a user eight relevant results or zero results. This application does not stem each token, thus not returning the best possible results to the user. Furthermore, my application is a more specialized search engine, which mainly attracts men who love to shop and find great sales online.

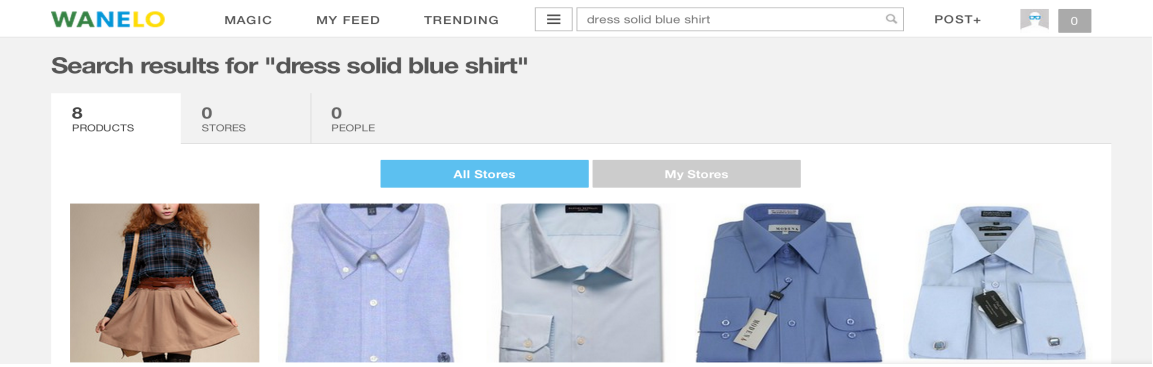

Figure 1

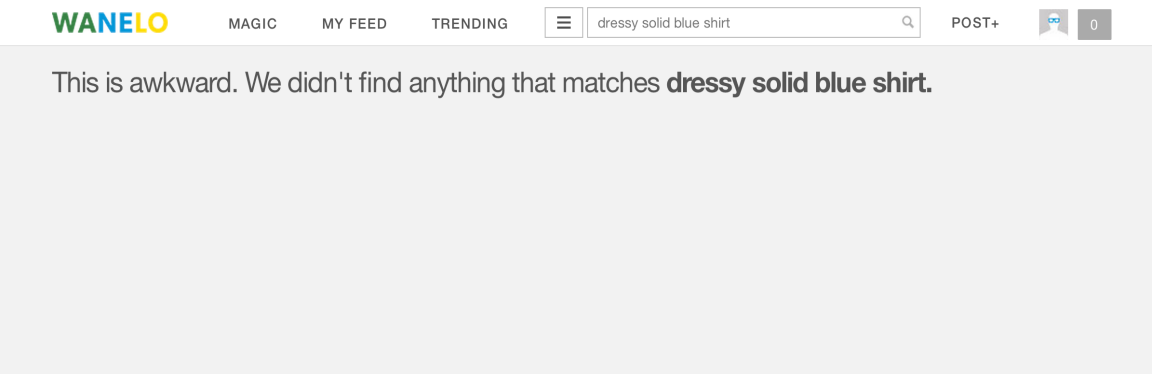

Figure 2

Problem Definition and Methods

In order for SaleTags to be of great usefulness, I needed to access several retail stores' sales page, which totaled about 3000 products. Since no public API was available to access the data from these websites, web crawling proved to be the best way to obtain all the necessary data. I selected Scrapy, a web crawling framework written in Python for crawling web sites and extracting structured data. The first instinct was to web crawl entire pages, but it was soon realized that pages were structured differently, so crawling the entire page would return JSON objects with different information. Another problem was that some data was loaded using Ajax. The solution to this was to use the URL of the file called by Ajax directly. Using the developer tool provided by Chrome, I searched under the Network tab to see which file was being called during a page reload. Then, I chose to only extract the title, description, colors, regular price, sale price, image, and URL of each product. This required a quick tutorial of XPath for selecting the data to extract from the web page HTML source. All the scraped data was next outputted in JSON format to generate JSON files.

To allow very fast full text searches, I used Elasticsearch, a powerful open source search and analysis engine based on Lucene. The index API from Elasticsearch made it easy to crate an inverted index for all JSON objects available in the existing files, where all data in every field is indexed by default. When a product is indexed, its full text fields are analyzed into terms, which are used to create the inverted index. I decided to use the Standard analyzer, which is the best general choice for analyzing text. This analyzer splits input text on word boundaries, converts all terms to lowercase, and removes stopwords.

A major challenge that I faced during the project was choosing the best ranking function and combination for the search engine. By default, Elasticsearch uses the TF/IDF model, taking into account term frequency, inverse document frequency, and field normalization. I experimented a few queries with this implementation, but realized treating each field within a document as a big bag of word could only tell us if that bag contained our search terms or not, it couldn't tell us anything about the relationship between words. The results by the default

implementation had a precision of 6/10. Additionally, requiring exact phrase matches would be too strict a constraint, so I had to introduce a degree of flexibility into phrase matching.

This led to proximity query, which incorporates the proximity of the query terms into the final relevance score. So a query that matches several documents, which contain the same amount of words, would give a higher score to the document where the words are nearer to each other. Still, I discovered that proximity queries would require all terms to be present, making them overly strict. If six out of seven terms matched, a document is probably relevant enough to be worth showing to the user, but the proximity query would exclude it. Rather than making proximity matching as an absolute requirement, I modified it to act as a contribution to the overall score for each document. So the final implementation would first use a simple term match query that will already have ranked documents that contain all search terms near the top of the list. Then it would rerank the top results to give an extra relevance bump to those documents that also match the phrase query. The search API in Elasticsearch supports this functionality via rescoring, so I only needed to add it to the default relevance scoring function.

Next, instead of implementing a multi-field search form such as an Advanced Search, I took the approach of letting users type all of their search terms into a single field, and having the application figure out how to map different query strings to individual fields. I encountered three scenarios: adding more weight documents that have as many words as possible in the same field and returning the score from the best matching field, indexing the same text in multiple ways into multiple fields where a higher score is given to documents that have more matching fields, and finding as many words as possible in any of the listed fields as if they were one big field. I opted with the cross-field queries as it takes a term-centric approach, quite different to the previous two scenarios. It treats all of the fields as one big field, and looks for each term in any field. An advantage of using this approach also gives the option of boosting individual fields at query time, specifically the name field for each product.

The results are returned sorted by relevance, with the most relevant documents listed first, given by the relevance score of each document. Twitter Bootstrap was used as a simple and elegant solution to display the results to the user.

Evaluation/Sample Results

I tested my application by entering the same queries used for wanelo.com, express.com, and jcrew.com. A query of “dressy solid shirt” delivers the results shown in Figure 3 while a query of “black jacket with front pockets” returns the list shown in Figure 4. One of the key qualities of this application is that results are returned at an extremely fast speed, which deals a lot with the way Elasticsearch is set up. This application is a definitive improvement over current shopping search engines, which only search the name of the product to match a query. SaleTags searches the name of the product, item description, and available colors. This lets a user be more specific with a query such as “black jacket with front pockets.” *Black jacket* could be a name for any product, but rarely will a product be named *Black jacket with front pockets.* The description field of an item provides much more information about a particular product, thus helping users narrow results as best as possible.

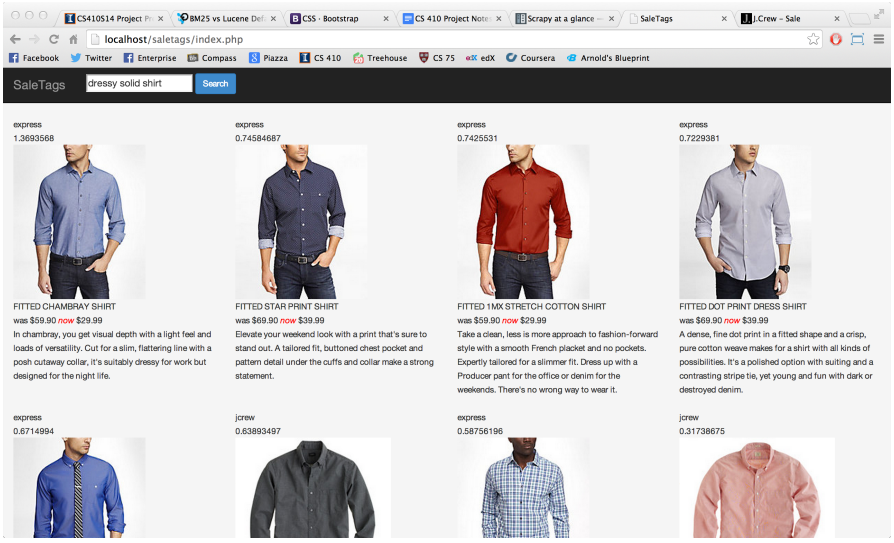

Figure 3

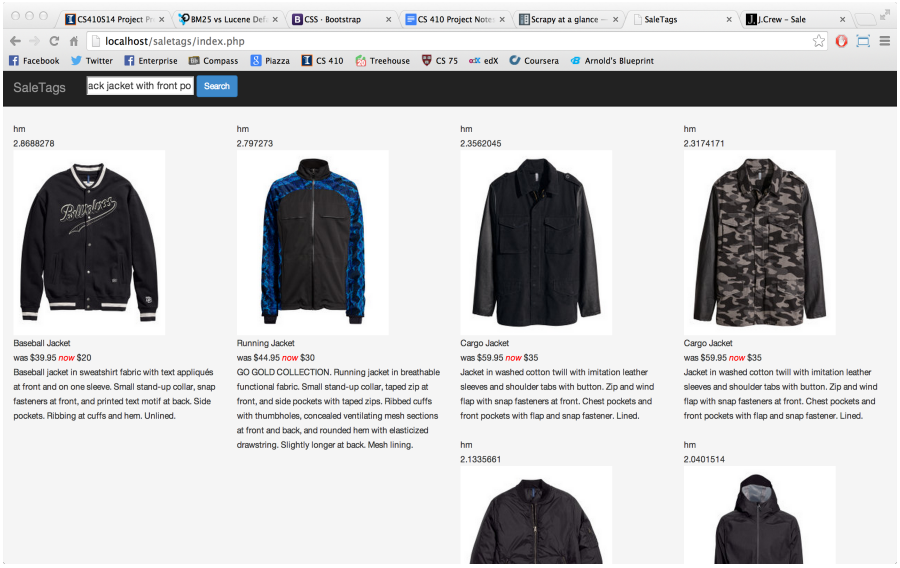

Figure 4

Conclusions & **Future Work**

This project definitely gave me hands-on experience on developing some novel information retrieval tools. This project helped solidify material from class such as web crawling, indexing documents, and evaluation of IR systems. I also learned about the various technologies available that lend powerful search capabilities to any application. There are several things that could be implemented in the future to make it a better overall system. An easy fix would be to add more stores/products, to decrease the results limitation associated with products like belts and shoes. Moreover, I could add a recommender system to produce a list of recommendations through collaborative filtering. Using a user's past behavior, the system could predict new items on sale that the user may have an interest in.

References

[1] http://wanelo.com/
[2] http://www.hm.com/us/department/MEN
[3] https://www.jcrew.com/mens-clothing.jsp
[4] http://www.express.com/clothing/Men/sec/menCategory
[5] http://www.urbanoutfitters.com/urban/catalog/category.jsp