MEng Project Report

Name: Philip Su (ps845)
Advisor: Serge Belongie (sjb344)

Introduction

For my Master's of Engineering project, I built a tool called COCO-Text Explorer. This project was completed during my second semester at Cornell - Spring 2016. My project advisor was Serge Belongie. I consulted Andreas Veit and Tomas Matera, members also involved in the COCO-Text dataset project, for resources and feedback.

The Computer Vision Group at Cornell Tech has compiled a large scale dataset of images of text in natural settings called the COCO-Text dataset. The dataset is based on the Microsoft COCO (Common Objects in Context) dataset found at: http://mscoco.org . The dataset consists of 63,686 images, 173,589 text instances (annotations of the images), and 3 fine-grained text attributes. The dataset consists of 2 types of objects - Images and Annotations. The Cornell Vision Group has created a COCO-Text API to query Image IDs as well as Annotation IDs.

The goal of this project was to create a COCO-Text Explorer: a search engine that displays images as well as related annotations for a given query. The interface queries the COCO-Text dataset through the COCO-Text API and then returns image results, which can be used to assist with future work in analytics and correlation studies for text recognition in natural images.

Dataset

The COCO-Text dataset and paper can be found at: http://vision.cornell.edu/se3/coco-text/. The dataset was not collected with text recognition in mind, so it is less biased towards text and thus very useful in improving methods of text recognition in natural images.

For each Image object, there exists 1 or more Annotations, and each Annotation contains information about the text that is identified within the Image object.

Annotations are primarily categorized by 3 attributes - *Class*, *Language* and *Legibility*. The *Class* attribute takes on 2 values: *Machine Printed* or *Handwritten*. The *Language* attribute takes on 3 values: *English, Not English or N/A*. Finally, the *Legibility* class takes on 2 values: *Legible* or *Illegible*. These 3 attributes are the basis for search query filtering for the COCO-Text Explorer. Additionally, Annotations have a *utf8_string* property that is the text the annotation refers to in the picture.

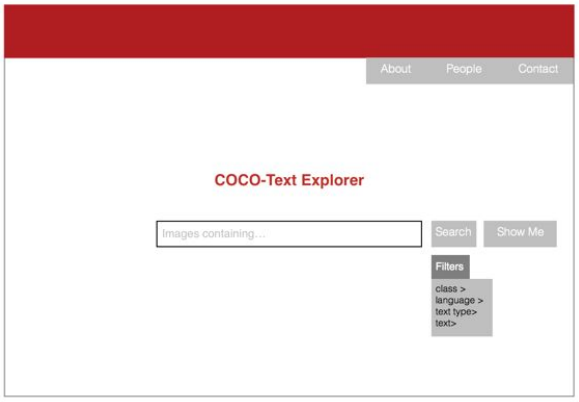
Tools

I developed the COCO-Text Explorer using primarily the Django Web framework, Twitter Bootstrap, and JQuery. I used the COCO-Text API developed by the Cornell Vision Group for querying images and annotations. Using the Image IDs returned from COCO-Text API, I loaded the actual images from the MSCOCO dataset stored on the Microsoft COCO server. The CORS Google Chrome plugin was used to allow the Explorer to make cross origin requests to the MSCOCO server. For version control and code storage, I used Github. Finally, I used Google Docs for progress updates, ideas and reports.

Process

Ideation Phase

I started the project by designing a markup of how the explorer would look like. The landing page is shown in Figure 1, with a search textbook as well as a dropdown for filters, a “search” button, and a “show me” button to teach people how to use the explorer.



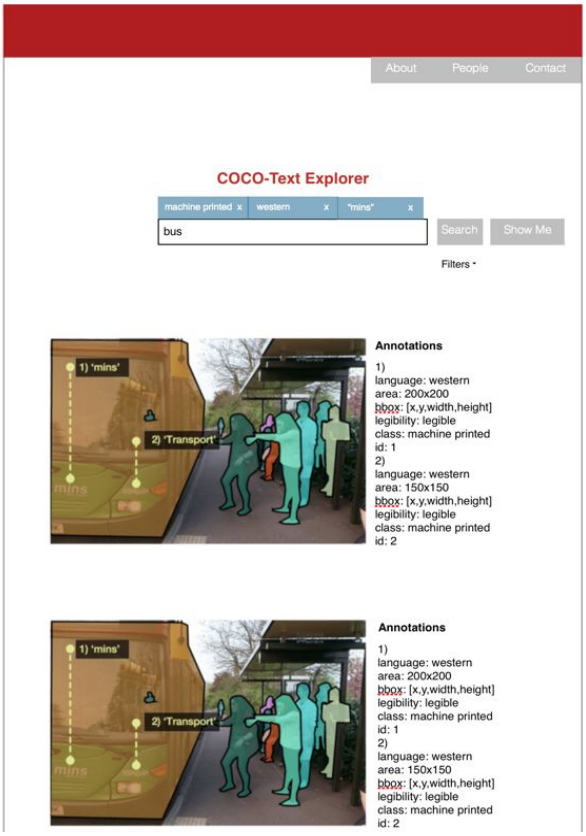


Figure 1(left) Markup of landing page of COCO-Text Explorer with dropdown filter
Figure 2 (right) Markup of results page of a search query

Figure 2 shows the result page after a user prepares the search query and clicks the “search” button. Each result contains an Image and all the Annotations that describe the text within the image.

Iteration Phase

After getting feedback from Serge and Andreas for the design of the Explorer, I began to write code and actually build the explorer. First, I noticed that some layout changes needed to be made. There were a large number of annotations for each image, so simply displaying a list of all annotations for an image was not adequate. To fix this, a scrollable list of annotations was implemented. Additionally, the main purpose of the explorer was to view the images and perform analysis, so the default behavior of the annotations were to be hidden, and they are only shown when the user requests them.

Furthermore, there were an extremely large number of results for some queries. For example, the search for the text “stop” with the filters {*Class* = Machine Printed, *Language* = English, *Legibility* = Legible} has about 320 result images. It would not make sense to have an enormous result page for all the results, so I implemented pagination with roughly 5 results per page. The user then chooses to view more results on another page. Moreover, because of the large number of returned results, caching was implemented so that paginated results would not require another COCO-Text API query.

Technical Walkthrough

Project Structure

The COCO-Text Explorer is built using the Django framework. Django uses the Model-View-Controller (MVC) model, but in the project there is no need for Models because the COCO-Text API was used to query data.

In the root directory of the project, there are 3 subdirectories:

- *coco_text_api/*
- *coco_text_explorer/*
- *explorer/*

The first directory, *coco_text_api/* contains the COCO-Text API developed by the Computer Vision Group at Cornell Tech. The files in there are *COCO_Text.json* which contains the Annotations and *coco_text.py*, code for the API.

The second directory, *coco_text_explorer/* is the main Django application for the project. It contains settings for the entire Django project in *settings.py* as well as the url routing for the project in *urls.py*.

The third directory, *explorer/* is the explorer application. The 2 main components of this application is the Explorer Controller and the Explorer View. These components are further explained in the next few sections.

Explorer Controller - Views.py

Caching Values

In *views.py*, there are 2 globally defined variables:

```
ct = coco_text.COCO_TEXT('coco_text_api/COCO_Text.json')
paginator = None
```

The object **ct** is the handle to the COCO-Text API and is initialized with the relative path to the annotations file. Queries are made using the **ct** object, which is initialized globally because the JSON file of annotations is large and initializing this object for every request would greatly slow the search engine.

The **paginator** is a Django paginator object (see: https://docs.djangoproject.com/en/1.9/topics/pagination/), and it is declared globally to cache results. The first time a query is made, the **ct** object loads the images and annotations. The list of all results is retrieved and the **paginator** object is initialized with the list of results. This way, if the user chooses to view the next page of results, the **paginator** object already contains the results from the search query, so another query with the **ct** object is not needed. Queries typically take a few seconds, so pagination would greatly slow down the Explorer if a request had to be made for every results page.

Index() Function Overview

The function **index(request)** in *views.py* handles the logic for searching with the Explorer. It takes in the request sent from the template view. Search queries are GET requests, so there are 2 types of GET requests - the first time a user loads the page and a query the user has submitted. To differentiate between these 2 requests, the existence of the string 'search' is checked in the **request.GET** dictionary.

If 'search' is not in **request.GET**, then the landing page is returned, as shown in Figure 3.

Figure 3. Landing page for COCO-Text Explorer

If 'search' is in **request.GET**, then the Explorer retrieves search results to display to the user. The Explorer checks if 'page' is in **request.GET**. If this value exists, that means the user is making a request to the next page in the result list. Therefore, the previously initialized **paginator** is used and the **paginator** is set to the results of a particular page index, performing some basic error checking for the page index. If 'page' does not exist, then the Explorer makes a COCO-Text API query. This is explained in the following section.

Making a COCO-Text API Query

To make a COCO-Text API query, the Explorer first extracts the filters that are contained in the HTTP GET request. The filters that are retrieved in **extract_filters(request)** are: *Class, Legibility, Language*. Note that the COCO-Text API currently does not support filtering by *utf8_string* yet (Andreas has been notified of this). Then the Explorer transforms this into a list of filters **cat_list** which can then be used in our COCO-Text API query:

```
img_ids = ct.getImgIds(imgIds=ct.train, catIds=cat_list)
```

The Explorer first queries Image IDs with Annotations that satisfy the filters. Then, the Annotation IDs for those Image IDs are queried:

```
anns_ids = ct.getAnnIds(imgIds=img_ids)
```

Then, the Annotations are loaded from these Annotation IDs:

anns = ct.loadAnns(ann_ids)

Next, all Annotations for an Image are aggregated. This is done by generating a mapping of Image IDs to a list of Annotations in **img_ann_map**:

{imageID1 : [ann1, ann2...],
imageID2: [ann10, ann11...]}

The Explorer displays more popular results first, so the metric used was Annotation count. A list of image IDs sorted by the number of annotations that the image was created - **sorted_img_ids**.

Next, the Explorer checks the HTTP GET request to see if the user provided a search term, which is mapped to *utf8_string* in the request. Then, in **extract_search_result()** all Image IDs are iterated through and checked for a matching *utf8_string* field in any of their Annotations. If a match exists, then the Image and all of its Annotations are added to the result list.

Finally, a Paginator object is generated for this result list, initialized with 5 entries per page. This object is cached and can then be used for future HTTP GET requests for the next page of entries for the same query.

Explorer View - Templates & Static files

For the front end view for the COCO-Text Explorer, Django Templates with Twitter Bootstrap CSS, Javascript and JQuery are used for design and functionality.

Index.html

This is the template for the website. The Explorer was designed to be responsive and takes advantage of Twitter Bootstrap fluid containers in their Grid System (see: http://getbootstrap.com/css/#grid).

The search functionality is contained inside an HTML <form> with the id “search-form” and it sends a HTTP GET request when the form is submitted. Inside this form, there is a text input that is labelled with *utf8_string*. This is the search box on the Explorer page and will allow the controller to look for matches in the *utf8_string* attribute with the text entered in this search box. The dropdown with filters, shown in Figure 4, are handled with Javascript in explorer_script.js, further explained in the explorer_script.js section below. The dropdown is an unordered list that the user can select filters from.

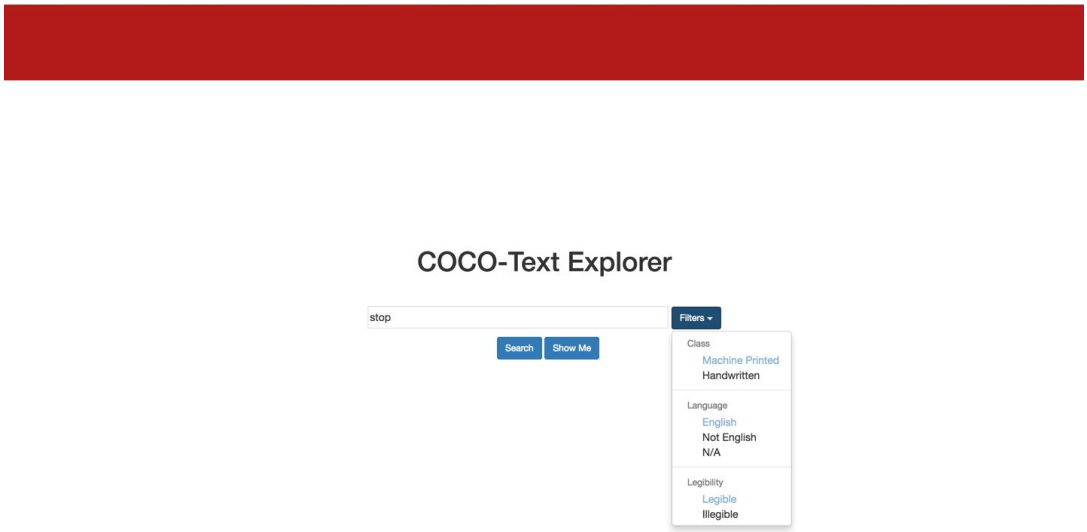


Figure 4. Dropdown of filters to apply to search query

At the end of the search form, there is a “submit-row” which contains 2 buttons. The “Search” button is inside an <input> tag and will send a search request to the controller in *views.py*. The “Show Me” button is inside a link <a> tag that generates a popover with a short description explaining how to use the Explorer, as shown in Figure 5.

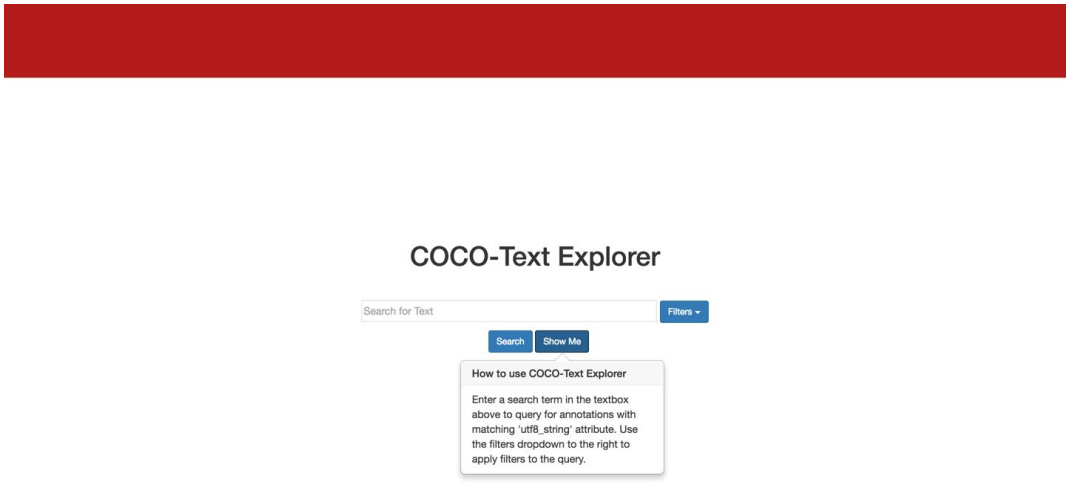


Figure 5. Show Me popover shows how to use the explorer

After the search form is the results list in the template. This is contained inside a template for loop:

{% for result in results %}...{% endfor %}

This part of the Django template will only be rendered if there are results sent from the controller *views.py* (in other words, the first time the user loads this page, the result list is empty and this part will not be loaded). The HTML for a search result inside the for loop is for a single search result as shown in Figure 6.

Figure 6. A search result in the explorer.

At the top of each result there is a “Show/Hide Annotations” button. This will toggle the annotations from its initially hidden state to a visible state, as shown in Figure 7. The annotations are contained inside a scrollable table because some images have a large number of annotations.

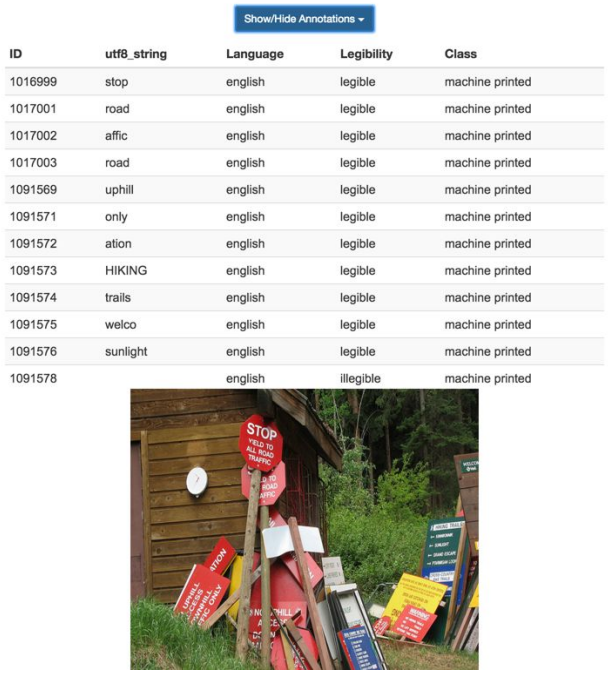


Figure 7. Toggled visible annotations for a search result

Next, the actual image is inside the “result-img-container”. In the template, an inline Javascript function is used to obtain the image from the MSCOCO server. The function queries the link http://mscoco.org/explore_visualization/ for the images. This is done because there are a large number of images and it would not make sense to download the entire image set locally. This script was provided by Tomas, and requires the CORS Google plugin to actually see the images (see: https://chrome.google.com/webstore/detail/allow-control-allow-origi/nlfbmbojpeacfghkpbjhddihlkkiljbi?hl=en). Note, this should be turned on only when viewing images as this makes browsers vulnerable to various security attacks. See Figure 8 below. At the bottom of each result is a red label for the image ID.

Finally, at the bottom of the *index.html* template is the pagination navigation. A left arrow if there are previous results, a right arrow if there are next results, and the current page number out of all pages, as shown in Figure 8 below.

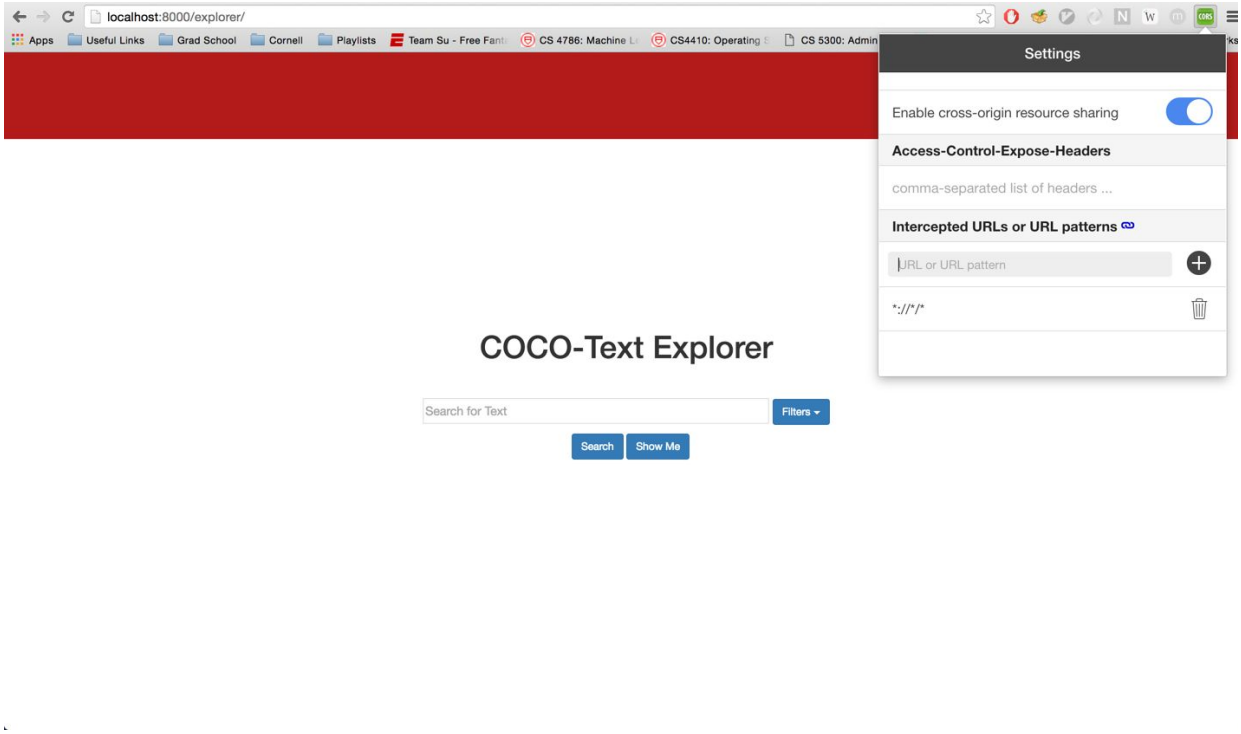


Figure 8. The CORS plugin must be used to view images

Figure 8. Pagination navigation for results at the bottom of the results page

explorer_script.js

The explorer_script.js mainly handles the filter selection in the filter dropdown. A global variable filterValueMap is maintained and whenever a dropdown-menu item is clicked ($(".dropdown-menu ul li").click(...)), the filter and the filterValue is extracted from the HTML template. If the filter already exists in the map, then the user is de-selecting the filter and the filter is removed from the map. Otherwise, the user is selecting the filter and the filter and its value is added to the map. The next function in the script adds background highlighting when a user hovers their cursor over a filter.

The submit function for the search form - $("#search-form").submit(...) - is called when the user clicks the "Search" button. This function appends <input> tags for every filter in the filterValue map to the search form so that when the request is sent to the controller *views.py*, all the filter mappings will be contained in the request.

Finally, there is a Javascript function that allows the "Show Me" button to display a popover of how to use the explorer.

Future Work

The COCO-Text Explorer is fully completed. The explorer can now be used for future work in analytics and correlation studies for text recognition in natural images. Users can make

queries on the COCO-Text dataset and visually inspect the similarities and correlation between the images in the results.