Practicum 1 Memo - Team Anonymous

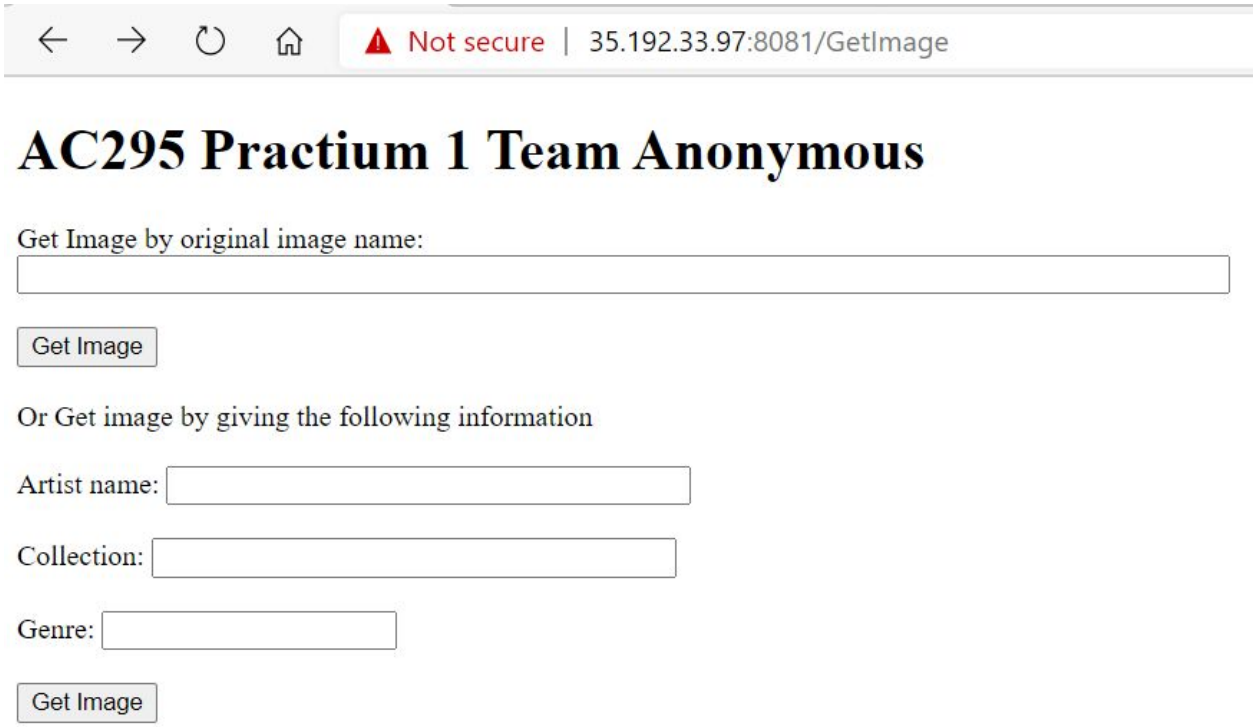
Jason Zhang, Erin Yang, Wenjie(Bianka) Gu, Shaoling Han

Video link: [practicum1_TeamAnonymous.mp4]
Github Repository:
https://github.com/hanshaoling/AC295_TeamAnonymous/tree/master/submissions/practicum1_TeamAnonymous

In this practicum, we developed an art search engine that has 2 basic functionalities, 'the simple query' and 'the similarity query'. We also implement 3 extra features. The first one is, rather than two frontends, we deploy everything with **one frontend**. The second one is that a **new metric** generated by an autoencoder is implemented to get similar images. The third one is that we can handle **additional queries** by having users inputting artist name and/or collection and/or genre.

Implementation of Basic Functionalities with One Frontend

To handle both simple query and similarity query in one frontend, we use Flask-restful and extension of Flask. Each functionality will just be a class in the frontend script and we specified resources for these two classes. The simple query uses the resource /GetImage and the similarity query uses the resource /SimilarImage. Screenshots are attached below:



On the backend side, we handle the simple query by matching the user input with the original_image_name column in the metadata. The backend will find the matching row and send back the corresponding image by extracting the file path in that row. We noticed that all entries in original_image_name have .jpg extension. As in practical users may not provide the image name with the extension, the backend will automatically add the .jpg extension if the user's input does not have that. We also take advantage of dask dataframe to speed up the searching

Implementation of Cosine Similarity

For an input image, we calculate its cosine similarity scores with all the images in our dataset. Before calculating the cosine similarity, we resize each image in the dataset to a 32*32*3 tensor and flatten the resized images to an array of size (3072,). To speed up the computation, we used dask array to calculate the cosine similarity between each pair of flattened image arrays.

Implementation of New Metric

Besides downsizing the images and calculating similarity scores, latent space generated from autoencoder is used to query similar images. We trained an autoencoder locally using all the images, and saved the model structure and weights after completing the training. In order to reduce the runtime of a single query, we saved the feature map for all the images and uploaded them to our database server. In that way, we only need to extract the feature map of the uploaded image. We also take advantage of the properties of dask arrays to speed up the calculation process. A dropdown menu is offered for users to select which metrics they want the query to be based on.

The latent space of our autoencoder is a 4*1 vector. By examining the quality of the 'get similar image' feature, we find out that it mainly tries to match images that share similar lightness of colors across different locations. For example, in our video, we uploaded an external cat photo.

The bottom half of the resulting queried image and the cat photo both share blue-green color, and the upper half of both images are in lighter colors.

Implementation of Handling Additional Queries

Our search engine can also handle additional queries by having users inputting artist name and/or collection and/or genre. When users do the simple query, they can either submit the original image name or provide information about the artist,collection or genre. If they provide both, the backend will only use the image name to select the image. Otherwise, if the image name is blank, the backend will try to find a row that matches the user's input of artist, collection and genre and return the corresponding image. If there is no match, the backend will return a message saying that there is no image that matches the criteria.