Design Facebook Photo Tagging

Q: When you say reduce the effort of tagging people in photos upon upload, do you mean automatically tagging or just presenting suggestions on which users should be tagged in the photo?

A: Let's present a list of suggestions.

Q: Will these suggestions be in a dropdown menu, perhaps appearing when a user clicks on a face?

A: Yes, there should be a box surrounding people's faces in photos such that when uploading the photo, users have the chance to click the boxes and tag a user.

Q: Do I need to design the ingestion of these photos into some analysis system?

A: Assume that the photos you have access to come into an HDFS cluster from a batch ingestion each day.

Q: Do I have to design the system which serves the tag suggestions, or just the model(s)?

A: Let's not focus on serving the predictions.

Q: Okay, do I need to design the ingestion of these photos into some analytics system?

A: Assume that the photos you have access to come into an HDFS cluster from a batch ingestion each day from our PostgreSQL cluster.

Q: Can I assume that these images are in HDF5 format under 30MB with dimensions under 1500x1500?

A: Yes, that's fine

Q: Can I use pre-trained models?

A: Yes, but you have to explain how you'd train and use them, and how they work.

Q: Can I assume that we have access to a workforce which can label images?

A: Yes, but go into detail on how you'd use this workforce to label images.

Q: Can I assume that we only need to detect frontal faces, no occlusions, and no severe illumination problems?

A: To start out with, yes. At the end of the interview, we'll revisit this.

Q: Will we want to detect faces from various distances away from the camera lens?

A: Yes, you should design a system which support multiple scales of faces.

Q: I'm assuming that we want to be able to detect more than one face per image?

A: Yes. That's right.

1. Gathering System Requirements

As with any ML design interview question, the first thing that we want to do is gather requirements. We need to figure out what we need to build and what we don't need to build.

We're designing a system which allows users to more easily tag other users in photos they upload by presenting suggestions for which users are in the photo.

To accomplish this, we'll need to do 5 ML-related tasks to implement tag suggestions:

- Image and Label Collection
- Image Pre-processing
- Face Detection
- Face Recognition
- Performance Measurement

We'll also need to rely on systems which we won't touch on or design:

- The website's UI displaying the suggestions
- The inference service providing the suggestions to the UI
- The HDFS cluster and underlying processing cluster(s)

2. Image and Label Collection

Assuming we have access to images, we'll need two roughly equal sized samples of labeled images. One sample where each image contains one or more faces and another sample where each image contains no faces. The images with faces in them should contain faces at different scales, contrasts, poses, and facial expressions. To start it should include frontal faces with no occlusions or illumination problems. We'll start with roughly 5000 samples of each image class, faces and no faces.

To get the images labeled we'll need some annotation software which provides images and to a workforce so that the images can be labeled appropriately. The annotation software should provide examples of what frontal face images are and are not. This will provide labels for the images so that we can train a model which detects faces in images.

3. Image Pre-processing

This stage is relatively simple. We can start by standardizing the images. Here, that means finding the mean and standard deviation of all the pixel values across the entire training set. For each pixel of each image we will subtract the mean and divide by the standard deviation.

4. Face Detection

Before we can perform facial recognition which will enable us to provide tag suggestions for photos, we need to detect the faces in an image.

Generally, we would use a pre-trained model for this. Since we're dealing with only frontal faces, we can use OpenCV's implementation of a cascade classifier. It's a model which uses the Viola-Jones algorithm to detect objects. The algorithm, based on the Viola-Jones framework, uses a cascade of Haar-based features to determine if a subsection of an image contains a face or not.

Haar-based features are filters placed on top of an image where the sum of some parts of the image subsection are subtracted from another sum of parts in the subsection. This is done in an effort to detect areas of adjacent darkness and brightness. An example of why this is helpful in detecting faces is that, typically, the area below a face's eyes is brighter than the eyes themselves. Haar features are created from subsections of an image, or window. This window will have effectively every possible Haar feature tried on it of all possible sizes. This includes line features, edge features, and four rectangle features. To speed up the calculations of all these Haar-based features, Viola-Jones works by creating an integral image which stores the cumulative sums of pixel values in the image. This allows for efficient Haar feature calculations. To narrow down to the large number of features generated from the exhaustive search for all Haar features, each independent Haar-based feature will be used in a weak classifier. The weak classifier takes the difference between the sum of the pixels in each Haar subregion and learns the best threshold of that difference according to which threshold produces the best classification rate. A weighted sum of these weak classifiers will form a strong classifier. Adaboost is used to assign weights to each of the weak classifiers based on their ability to independently classify subsections (face or no face) of images correctly. This results in only a small subset of the Haar-based features being used.

Next, we will use these selected Haar-based features to detect faces in an image. Each image will be examined with a 24x24 pixel sliding window. The window evaluates each Haar-based feature stage in descending order by the weights determined by Adaboost. A stage contains one or more weak classifiers. If the sliding window does not pass the threshold of a stage, which is determined in training, the sliding window moves on to the next subsection to avoid performing unnecessary evaluations of subsequent stages. If the sliding window passes the threshold of each stage, then a bounding box can be placed around the subsection to indicate a face is in the subsection of the image. A bounding box is a series of points, typically four, to make up the corners of a square which forms a box around the object we're attempting to detect. In our case, this box will surround a face. To handle different scales, or sizes, of faces within an image we'll use an image pyramid. An image pyramid starts with the original image and down samples that image according to neighboring pixels. This downsampled image is then run through the same process as the original image.

5. Face Recognition

Now that we have detected the faces, we want to be able to recognize the faces. For this scenario, we generally want to represent the faces in an embedding space because we likely won't have labeled identities of the faces in the photos when they're uploaded. When someone uploads a picture to Facebook, the photo could have the uploader, friends, family, or even strangers in the photo. If we represent faces in an embedding space then we can eventually label these embeddings with identities as they are given. Here, that means waiting until users manually tag on another their photos or themselves in others' photos.

These embeddings can be obtained from a pre-trained model such as SphereFace. SphereFace is a convolutional neural network with a ResNet architecture. After several convolutions and some max poolings, images of faces are represented as embeddings. These embeddings are optimized with a softmax loss to have more distance between unlike faces and less distance between like faces. After the model is trained, the model can produce embeddings for unseen faces. To differentiate faces, a standard softmax isn't the greatest loss to use because there's a low inter-class variance and high intra-class variance. All that means is that two unlike faces may be smiling, have two eyes, a nose, and a mouth (low variance). However, two like faces, belonging to the same user, could be in different poses, lighting, contrasts, and be making different facial expressions (high variance). We need a loss function that fights this tendency in the data and attempts to explicitly separate unlike faces and compress the distances of like faces. The loss function which SphereFace uses is called angular margin softmax. The general idea of angular margin softmax is to project the embeddings on a sphere (hence SphereFace) and introduce a tunable margin which encourages large angular distances for inter-class faces and smaller angular distances for intra-class faces.

We'd use this trained model to recognize faces from the bounding boxes produced by the face detection algorithm. Let's say someone uploads a new photo to their account. The image would start by going through the face detection process we talked about. Then the user will have the opportunity to click on those bounding boxes produced by the face detection algorithm.

Meanwhile, the face subsection will be sent through the pre-trained SphereFace model which will produce a face embedding. This face embedding will be used in a nearest neighbor algorithm. Then, there's a few things which can happen

- We haven't seen the face yet, based on a threshold of distance or similarity in reference to faces we've already seen. So in our case, if the cosine similarity is below some threshold, then we won't supply a suggested tag for that bounding box.

- We have seen the face one or more times, meaning the threshold is not breached for an unseen face, but the other similar faces we've seen do not yet have an identity. In this case, we also won't return a suggested tag for that bounding box.

- The happy case where the KNN returns a series of identified faces. Some predefined rule, for instance the top m items found by the KNN algorithm match, then we'll return the suggested tag. To tune m, we'd do hyperparameter tuning through offline evaluation. If the m threshold isn't met, we can return more than one tag suggestion or none if there is generally no consensus around the top m.

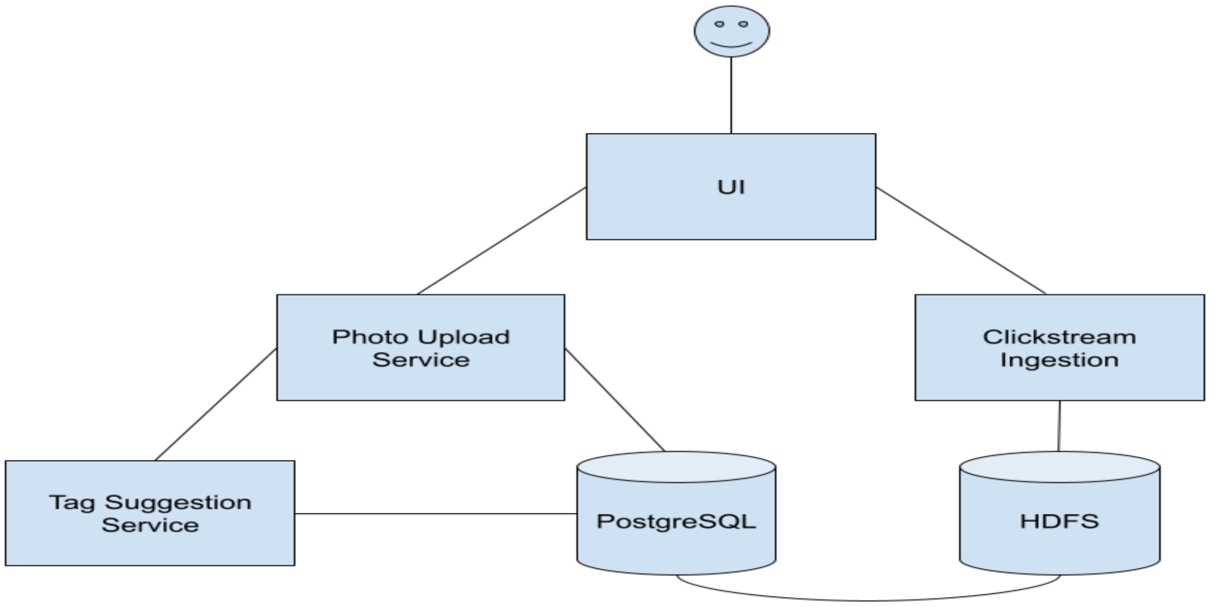
6. Performance Measurement

At the model level, it's good to measure the accuracy, precision, recall, and f1 score at ranks 1 through k. Here, the k-th rank is the k-th face in the k-nearest neighbors of face embeddings. At the business level, we're assuming the feature lowers the friction of a user wanting to tag people in their photos. If we can enable more users to be tagged in more photos, it will then show up on their friends' timelines providing more content to users' feeds which encourages users to stay scrolling longer which can likely be related back to a lift in ad revenue. We could verify this through A/B testing.

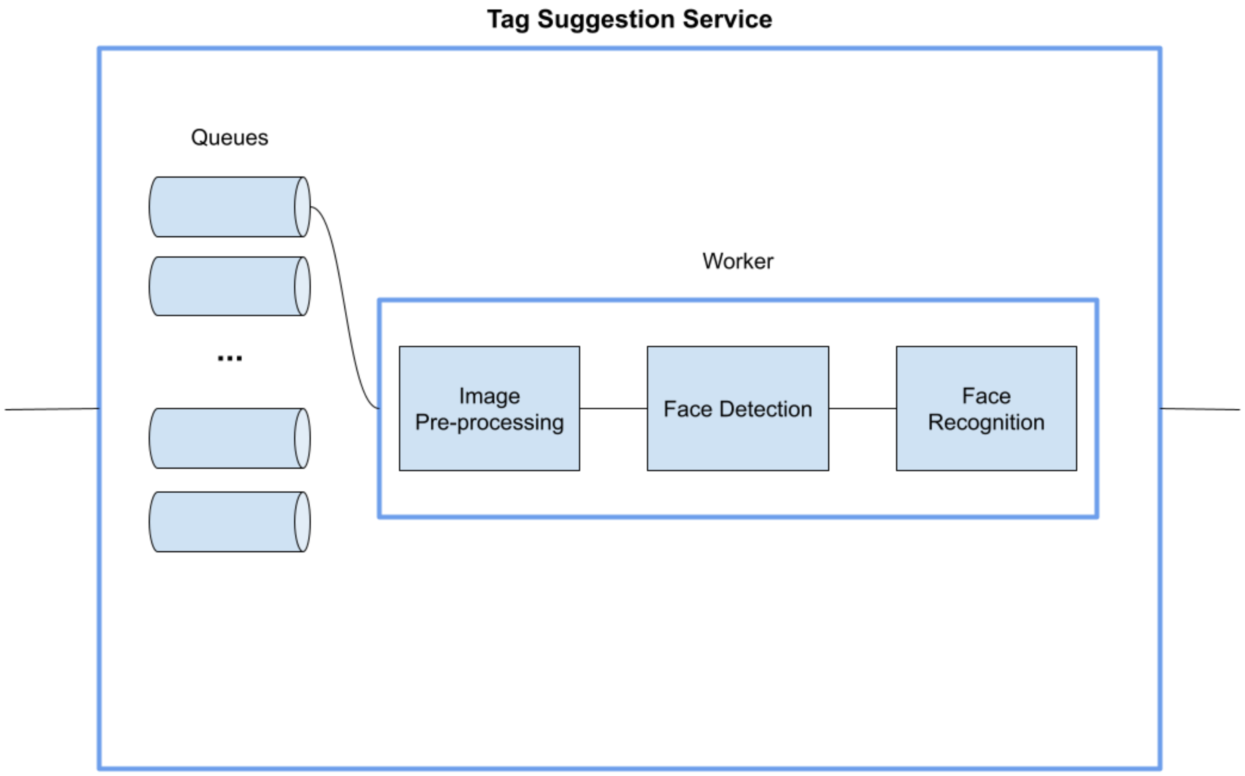
7. Bonus

If we assume there are non-frontal faces, we need a model to perform alignment. This process is sometimes called 'frontalization'. Generally, the goal with side faces, some small occlusions, and maybe images with illumination problems is to find landmarks on the face. We can use a pre-trained Multi-task Cascade Convolution Network (MTCNN). It works by generating and refining candidate bounding boxes with 3 stages of CNNs. It outputs 5 landmarks. The benefit with MTCNN is that it frames face detection and alignment as a joint problem.

8. High Level System Diagram



9. Tag Suggestion Service Diagram


Tag Suggestion Service
Queues
...
Worker
Image Pre-processing
Face Detection
Face Recognition