CSE 546 — Photo Poser

Harsha Sharma, Amit Kumar

1. Introduction

We live in a world dominated by social media which in turn is dominated by pictures. Almost everyone has some places on their bucket list which they want to visit. And, you haven't visited a place if you don't have an Instagram worthy picture for it. Photo Poser solves this modern day dilemma by providing suggestions to the user based on his photo in real time.

2. Background

The existing apps such as 'Posing App' suggest poses based on static input such as Wedding poses, Kids poses and Beach poses. The user has a bank of pictures he can see for ideas but there is no scope for dynamic suggestions. How does the user evaluate his clicked picture for happiness or joy ? How does the user see similar pictures ? How does the user get ideas for other popular poses for the landmark ?

Photo Poser solves this problem by accepting dynamic input from the user. The user uploads his photo and the app uses Google Vision API for landmark, face and web detection. Firebase is used for storing a dataset for popular tourist images of various landmarks. It maintains data synchronization across multiple devices. The uploaded image is populated to the dataset after successful landmark detection. The user can see popular tourist poses for the landmark which are retrieved from Firebase. Face detection is used to provide the user information about his expression like the likelihood of joy on his face, so that the user can get that perfect click.. The JSON from Google Vision API is parsed for providing the user with similar images to the one uploaded by him. The user can also evaluate previously clicked images using Photo Poser. In this case, he can also see the distance between his current location and that of the landmark. Google Maps API is used to provide a dynamic map functionality to the user where he can see his current location and the that of the landmark.

3. Design and Implementation

Application Architecture and Cloud Services:

Poser App is a scalable application and can support access from multiple devices concurrently. The application consists of four major elements:

a) Android Mobile application – takes image input and displays similar images and tourist posing images for the landmark

b) Google App Engine – main computing and web-hosting resource

c) Firebase – mobile and web application development platform which is used to store user images and poser images dataset

d) Google Vision API – used for landmark, face and web detection to allow dynamic analysis.

The mobile application uploads the image to Firebase which is a real time database and contains a dataset of popular images for various landmarks. The Firebase provides the URL of the uploaded image to the application. The core component of the application is Google App Engine which receives POST request from the android application. The App engine makes a call to the Cloud Vision API for landmark, face and web detection. The Vision API provides a JSON as an output to the App Engine. Google App Engine makes a call to the Firebase with the landmark identified by the Cloud Vision API and retrieves

four popular posing images for the landmark. The images obtained from Cloud Vision API and Firebase are displayed to the user as the output. The user can also use the Google Maps functionality provided by the application.

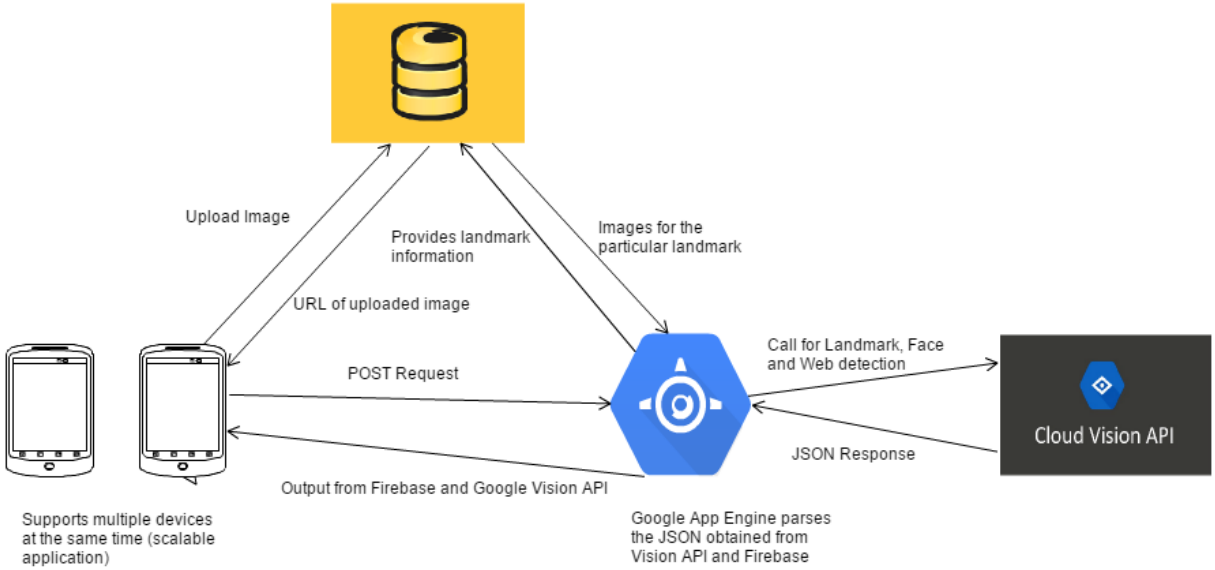

Figure 1: Poser App Architecture

Mobile Application – The user uploads the image from the gallery or camera to the android mobile application. The application sends a POST request to the Google App Engine and also uploads the image to Firebase.

Figure 2.1 *Figure 2.2* *Figure 2.3*

Figure 2.1 shows the screenshot of the input, output obtained from the Firebase database and face detection result

Figure 2.2 shows the screenshot of the similar images obtained from Google Vision API

Figure 2.3 shows the output from clicking on the Google Maps button

Firebase – Firebase is a real time database which synchronizes data across multiple users. It contains a dataset of popular tourist poses for various landmarks. The database is arranged according to the landmark name such as Eiffel Tower, Golden Gate Bridge, Louvre etc as shown in the below screenshot. The schema of our database consists of latitude, longitude and url. Latitude and longitude are used by the Google Maps functionality. url identifies the image stored in Firebase storage and Google App Engine parses the url and sends it back to caller android application to display posing images from the dataset.

Figure 3 : Snapshot of Firebase database which contains the dataset of popular tourist images.

Cloud Vision API : Cloud Vision API is the crux of our application as it provides the dynamic functionality of landmark, web and face detection. Google App Engine makes an HTTP request to the Vision API using filepath (Firebase URL). Vision API returns JSON to the App Engine for the particular request type. Vision API is the component which provides dynamic functionality to Poser App. It analyses the image in real time for face, landmark and web detection. Dataset is prone to incompleteness but Google Vision API provides the user with similar images even if the dataset doesn't contain tourist poses for that particular landmark.

Role of Google App Engine: Google app engine is the primary resource for developing and hosting web-applications in Google cloud platform. The mobile android app sends POST request to the Google App Engine which processes the request and uses Google Vision API to perform web, face and landmark detection on the uploaded image. It interacts with the Firebase to get the popular tourist images for the landmark detected by the Google Vision API.

```
JSONObject jsonObj = new JSONObject(resp);
JSONArray resparray = jsonObj.getJSONArray("responses");
JSONObject webdetection = resparray.getJSONObject(0);
JSONObject webdeJsonObject = webdetection.getJSONObject("webDetection");
JSONArray similariImages = webdeJsonObject.getJSONArray("partialMatchingImages");
JSONArray visuallySimilarImages = webdeJsonObject.getJSONArray("visuallySimilarImages");
```

Web detection is used to get the similar images by parsing the JSON obtained by Vision API. Parameters "partialMatchingImages" and "visuallySimilarImages" provide the similar images.

```
String line = httpResponseScanner.nextLine();
if (line.contains("detectionConfidence") || line.contains("joyLikelihood") || line.contains("sorrowLikelihood")
        || line.contains("angerLikelihood") || line.contains("surpriseLikelihood")) {
    resp += line;
```

Parameters “detectionConfidence”, “joyLikelihood”, “sorrowLikelihood”, “angerLikelihood”, “surpriseLikelihood” are used for face detection which provides the user with suggestions on the facial emotions of the uploaded image. This way the user can change his expression and get that perfect click.

Autoscaling:

appengine-web.xml file is used for implementing automatic scaling. Our project uses Gradle and appengine-web.xml file appears under /WEB-INF folder.

```
<?xml version="1.0" encoding="utf-8"?>
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
    <application>photoposing-165504</application>
    <version>1</version>
    <threadsafe>true</threadsafe>

    <instance-class>F2</instance-class>
    <automatic-scaling>
        <min-idle-instances>1</min-idle-instances>
        <max-idle-instances>automatic</max-idle-instances>
        <min-pending-latency>20ms</min-pending-latency>
        <max-pending-latency>automatic</max-pending-latency>
        <max-concurrent-requests>50</max-concurrent-requests>
    </automatic-scaling>


    <system-properties>
        <property name="java.util.logging.config.file" value="WEB-INF/logging.properties" />
    </system-properties>
</appengine-web-app>
```

<min-idle-instances> is the number of idle instances the App engine maintains during all times. It is set to 5 which is a trade off between maintaining low running costs and handling rapid spike in requests.<min-pending-latency> is the minimum amount of time that the App Engine allows a request to wait in queue before launching a new instance to handle it. 20ms is optimal for giving a responsive design to the user and also managing running costs. <max-concurrent-requests> is the number of requests an instance accepts before a new instance is created to handle the requests. It is set to 50. <instance-class> is set as F2 as specifying a higher class will result in higher running costs.

Solving the problem: Poser App solves the problem of providing dynamic output to the user based on the uploaded image. We created the database of popular posing images as such a dataset didn’t already exist. Poser App uses Cloud Vision API to provide suggestions on the face expression and similar images to according to JSON parsing. Dataset in Firebase is used for providing tourist poser images which provides the user with suggestions for modifying his pose.

Comparison with state-of-the-art: The current solutions have no provision for taking dynamic input from the user. Instead they use a static database of images which has no scope of any update. ‘Photo Poser’ is a significant improvement in the domain of photo posing apps. It provides dynamic functionality to the user. The user can upload a previously clicked image or can click an image using camera. Also, it provides Maps utility to get an idea of the coordinates of the landmark and current location of the user.

4. Conclusions

Accomplishments – ‘Poser App’ is one of its kind application which functions in real time. The existing solutions work on a static database and cannot provide landmark, face or web detection in the same time. We built an application using Google Cloud Services, Google App Engine and Google Vision API. Also, since no dataset existed for photo poses we built a database of popular tourist poses for various landmarks. We added the Google Maps functionality to the application which was not a part of the project proposal but a result of continual improvement and out of the box thinking.

Learning – We learnt how to implement Google Vision API, Google Maps API, Firebase and Google Cloud platform. We learnt how to use Android Studio for building android application. We learnt technicalities of JSON parsing which is an important skill as JSON is a popular format for data transfer. We researched various options for storage before selecting Firebase for building the dataset. Firebase is secure, robust, scalable and maintains synchronized data without any glitches. We learnt a lot about the pros and cons of other storage options such BigTable. It will be very useful while building apps in the future as we are already aware of the functionality provided by each kind of storage.

Scope for Improvement – We can build a larger dataset with tourist poses for landmarks all over the world to increase the scope of the app. The similar images parsed from the JSON of Google Vision API can be further examined for relevant pictures because the similar images are many times just images of the landmark with no poses. Landmark detection using Google Vision API fails in unclear images and images clicked from a distance. To handle this exception case we can add the functionality of suggesting nearby landmarks to the user by using his current location. The user can manually select the landmark and we can display tourist poses from Firebase. We can also provide integration with social media websites such as Facebook, Instagram and Snapchat and provide likecount for arranging the result based on decreasing likecount.

5. References

[1] "Vision API - Image Content Analysis ." Google., Accessed on 20 Apr. 2017. <https://cloud.google.com/vision/?utm_source=google&utm_medium=cpc&utm_campaign=2015-q1-cloud-na-gcp-skws-freetrial-en&gclid=Cj0KEQjwxbDIBRCL99Wls-nLicoBEiQAWroh6lGI2THRaoeCTwEsIVxsL9JDU9FKeIHF8fBVQOBDIEgaAsSi8P8HAQ>

[2] "Upload Files on Web | Firebase." Google. N.p., n.d. Web. Accessed on 20 Apr. 2017. <https://firebase.google.com/docs/storage/web/upload-files>

[3] "Sample Applications | Google Cloud Vision API Documentation | Google Cloud Platform."Google. N.p., n.d. Web. 18 Apr. 2017. <https://cloud.google.com/vision/docs/samples>

[4] Google.com. Developer's Guide http://code.google.com/appengine/docs/whatisgoogleappengine.html, Accessed on 15 April, 2017

[5] Google.com. Google App Engine Articles. http://code.google.com/appengine/articles/shelftalkers.html, Accessed on 22 April, 2017

[6] Zahariev, Alexander. Google App Engine (2009), Accessed on 22 April, 2017

[7] GoogleCloudPlatform. "GoogleCloudPlatform/dotnet-docs-samples." GitHub. N.p., n.d. Web. 24 Apr. 2017. <https://github.com/GoogleCloudPlatform/dotnet-docs samples/blob/master/vision/api/DetectFaces/DetectFaces.cs>