Table Of Contents

1. Introduction

The program I have chosen to create for my senior project is a photo library organizer for Android Devices. Smart phones and tables have been ever improving ever since the first tablets came into the market in the early 2000's. Ever since then competitors have been trying to improve their devices so stand out on the market. Tablets and smart phones over the years have been improving upon internal storage and camera quality. With this increase in the amount of storage a device has it has become easier to take hundreds if not thousands of photos. I have noticed my self taking more and more photos with every new smart phone I buy. Smart phones rarely move backwards in terms of the total amount of data storage and because of this phones and tables today have more storage than a user could fill in a reasonable amount of time. Even if a user fills their device, many come with the ability to add an additional SD card to increase the total storage even more. With users taking more and more photos, it becomes exponentially more difficult to find a specific photo.

I have created an application that can analyze an image for its content and then create tags that define what is in the photo. By analyzing users photos for their content it allows a users to search for a specific photo much easier. The application starts up by displaying the photo gallery. The photo gallery consists of every photo on the users device. When the user turns on the Auto Analyzation option all images will be analyzed in the background for content tags and faces. While these processes run in the background they create tags on every image. Faces will be detected so the user can create people and view all images containing a specific person. Once all images have been analyzed the application can be used to its full potential.

When a user selects a photo from the photo gallery it will display the photo large, with similar photos displayed below. The program will analyze what tags are present in the selected image and from there determine what other images are similar and display them. If there is a person present in the current image than the application will display other images that person is in. This allows a user to select a photo and filter through similar images. This way of browsing photos feels natural and creates a sort of mood. The mood is the current types of images being displayed. At one point the user can be looking at images of the outdoors, then select a picture with a car in it, then start browsing all other images of cars making the user experience a sense of exploration. Comparing that to the normal linear display of users photos where photos are displayed based on when they were taken, feels stale and doesnt allow for older photos to show up. The user can search the image library using a search string.

This search feature allows users to search for content within their photos. A user can search for “Dog” and, if the user has photos of dogs, will show up with out the hassle of labeling ones own photos. The user does have the ability however to edit tags given to an image. The user can add, or remove tags from an image as they see fit. New images can be taken through the application by selecting a camera icon in the top right. These images will be analyzed just like the rest of the users photos.

A user can switch how images are displayed by switching between different galleries. The default is the Image Gallery. This displays users photos in a grid allowing for the most use of space as only images are displayed. Listing the photos by date will break up the gallery into individual days that photos were taken. This is great for giving the app a cleaner look as there are less images being displayed. Beyond these two basic ways of displaying images, images can be

listed based off of what the analyzers have found. The Faces library contains every image that a face was found in. This is great for browsing images of friends and family, and puts it all in one easily accessible library. Along with displaying images that contain faces, a People library will display all people within the application. Currently the user has to create people and select faces to define a person as the application only supports face detection, not facial recognition at this time. Finally the last generated library is the Photos by Tags library. This library will display a few of the most popular tags along with images that apply to that tag.

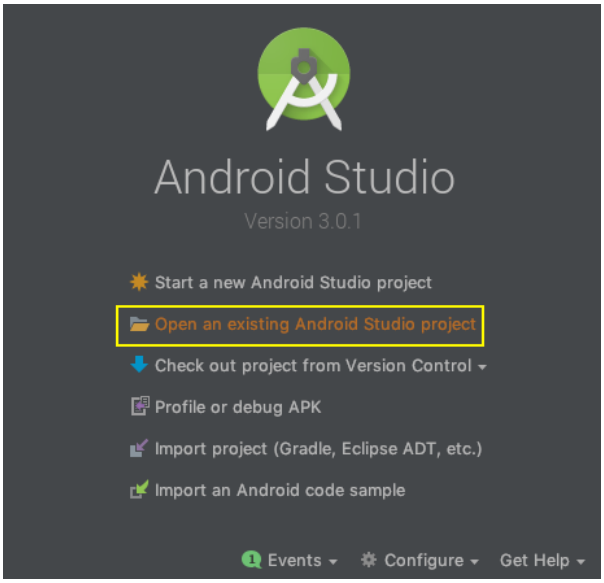
Instructions on how to run the application on the next page.

2. Installation Guid

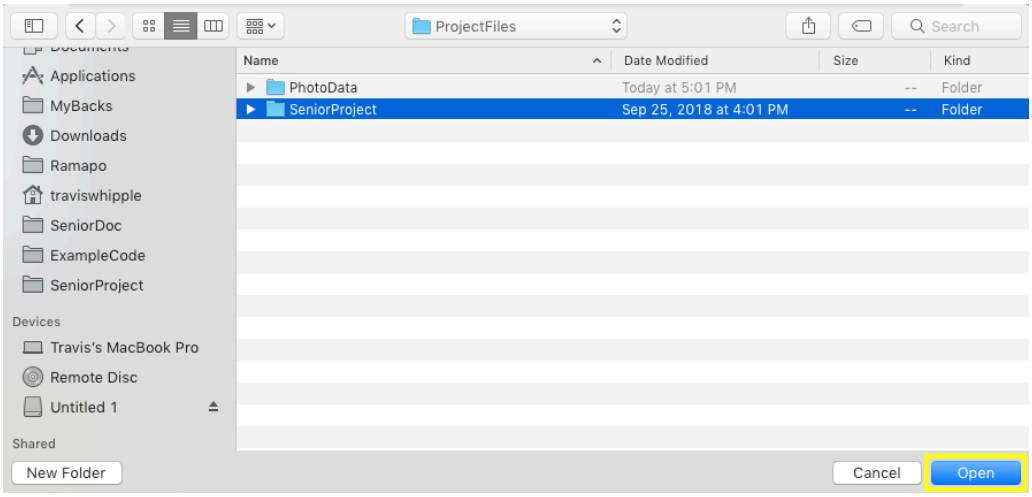
The installation process is defined below step by step. It has been made easier for the user by the inclusion of images and examples. The installation process has been divided into 8 parts to make the installation process easier. Installing the application should take no more than 5 minutes.

2.1 Loading Project Into Android Studio

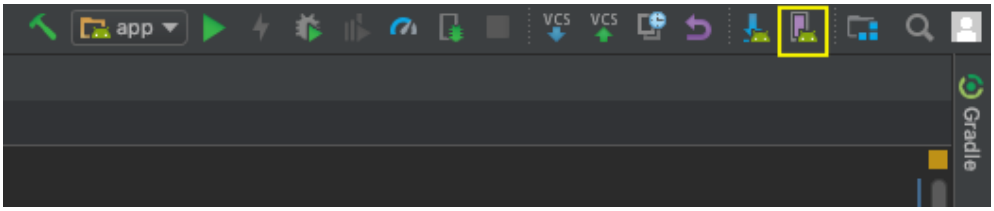
First launch Android Studios, and select and select “Open an existing Android Studio project”



Then select the “SeniorProject” file included on the memory stick and click open.



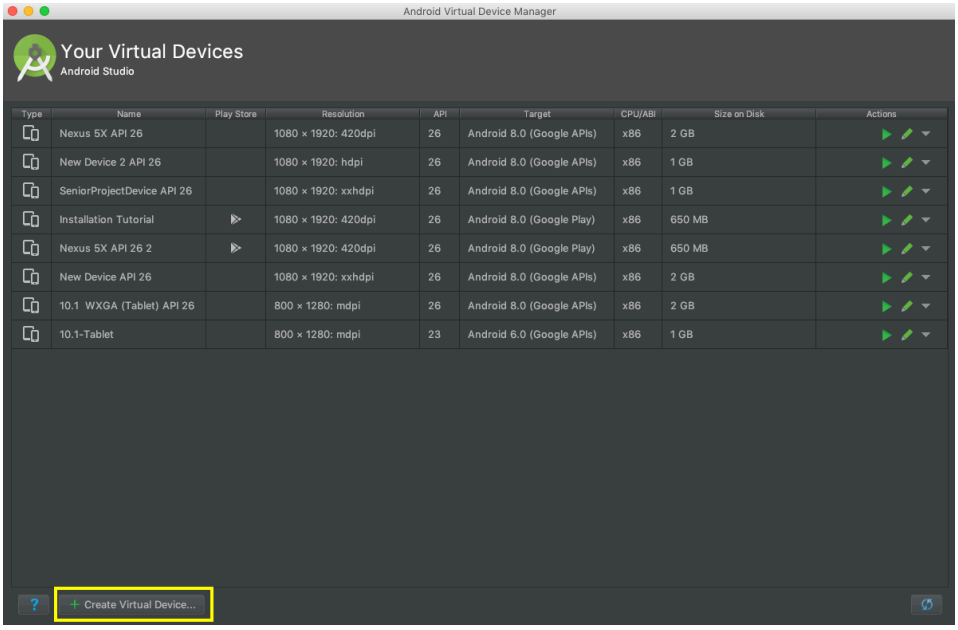
Once the project is finished loading, select the AVD manager button in the top right.



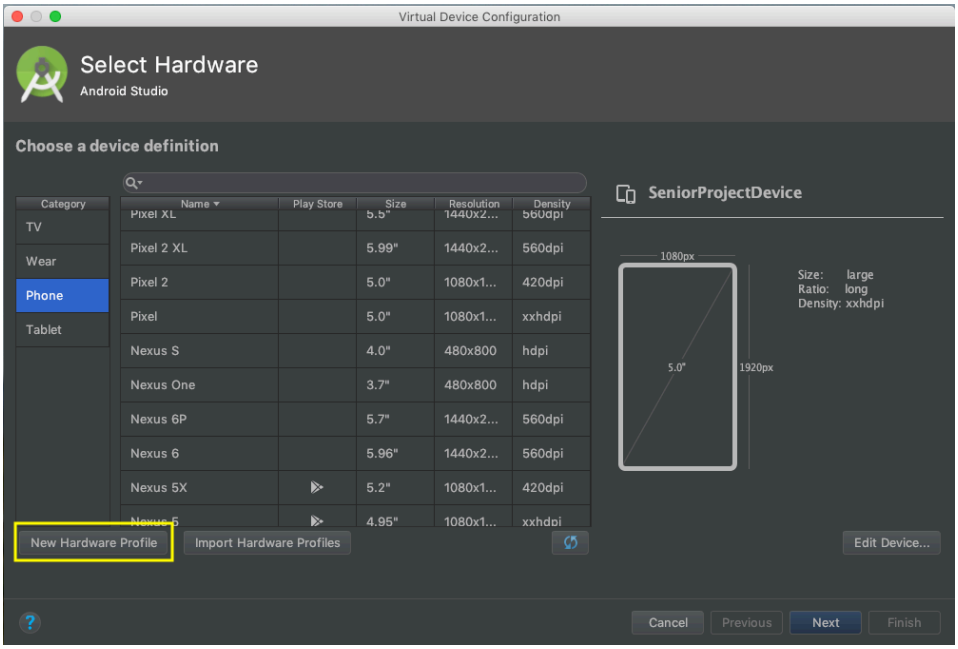
If no devices exist in this list whos target API is not Android 8.0 Oreo, then follow the below instructions, otherwise continue to the “Running Android Virtual Device” section. It is recommended to follow the below instructions to ensure a compatible Device to run the project.

2.2 Creating a new Virtual Device

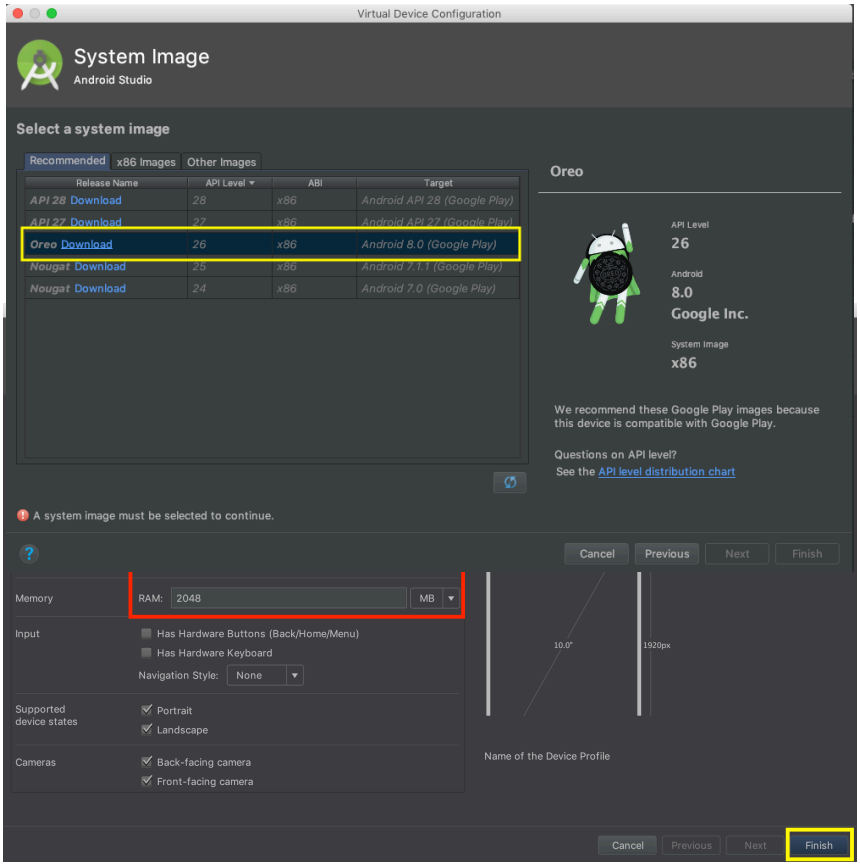
We will now have to create an Android Virtual Device with an API level of 26. Select "Create Virtual Device" from the bottom left most corner.



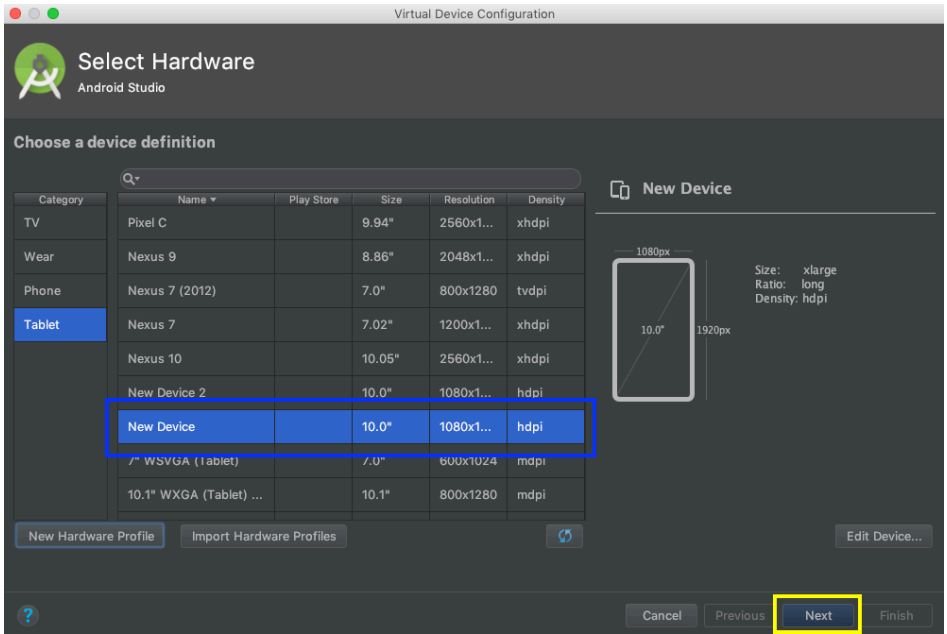
Then select "New Hardware Profile" to create a new Virtual device.



Inside the Configure Hardware Profile menu, ensure that the device has a screen resolution of 1080 by 1960 and that there is at least 1024 MB of ram allocated to the device. This is to perfectly emulate the device used to demo the project. The Screen Size should be between 8”-10” this will allow for more screen room, and prevent the program from being cluttered. Once these parameters are set, select the “Finish” button on the bottom right.

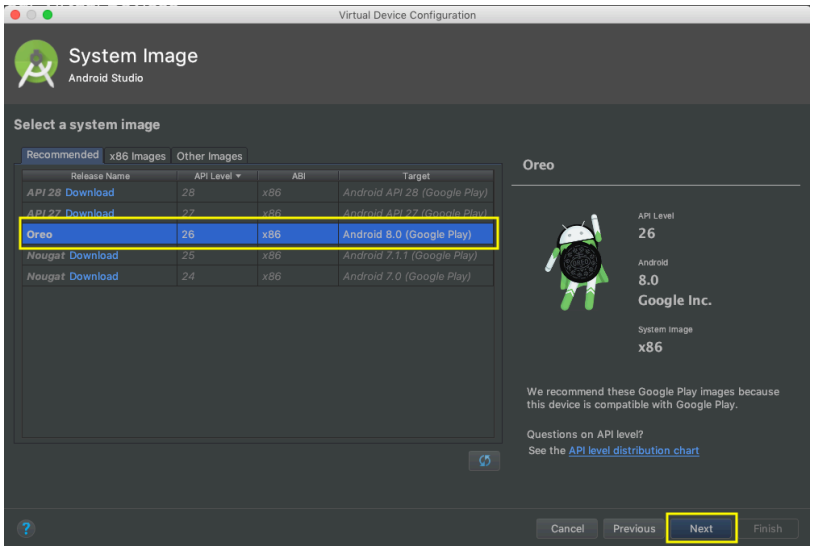


Select the new device you just created and press “Next”

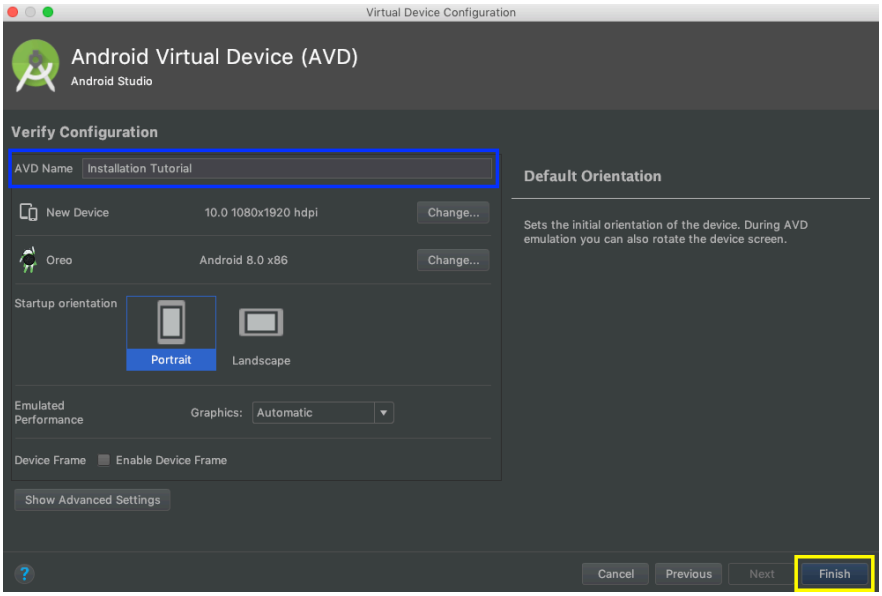


We now have to specify what System Image (Operating System Version) to install on the device. From the menu select “Oreo.” If it displays “Download” next to the Release Name then it has not been installed yet, download Release version “Oreo” now. After installer finishes, select “Finish” to complete the installation.

If / when the System Image is installed, select “Oreo” from the list and select “Next.” The “Download” option should not be present at this time, if it is, revert back to the above step.

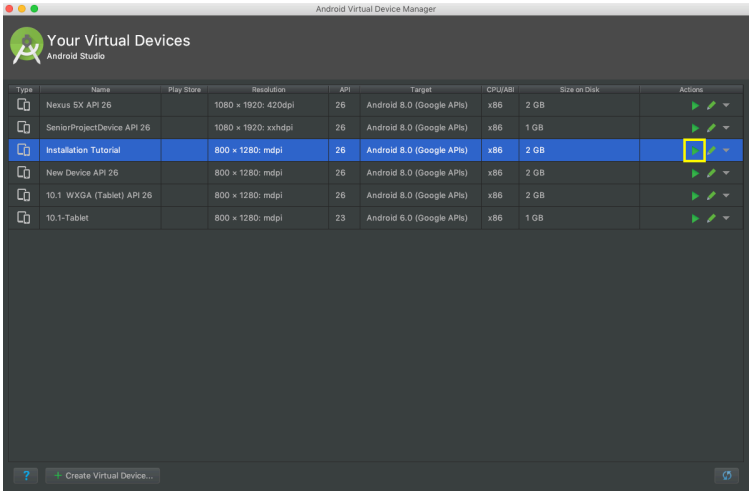


Next you can name the Device anything you like, for this example I will name mine “Installation Tutorial”. Check that the Device will start in “Portrait” mode. This app is designed to run in a Portrait orientation. Then finally click “Finish” in the bottom right.

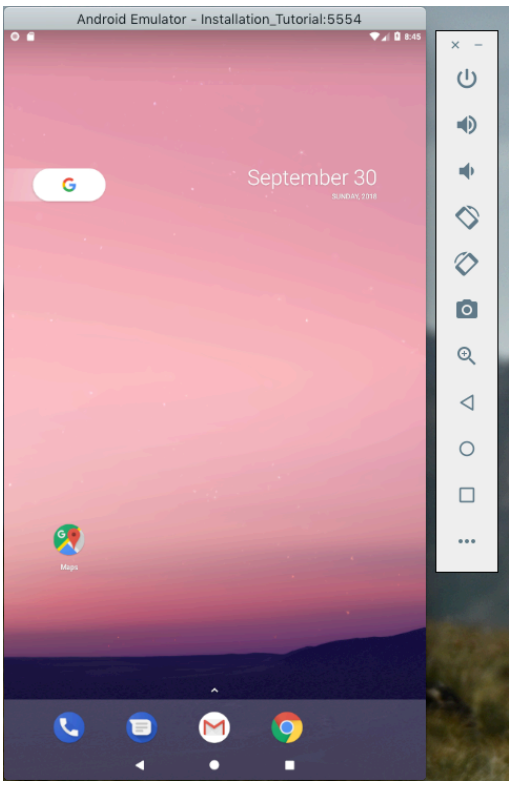


2.3 Running Virtual Device

To run the Virtual Device select the “Play” button next to the device you just created. For this example I have created my Device under the name “Installation Tutorial”



The Virtual Device will now power on in another window. Here is an example of what yours may look like:



On the right hand side we have a menu for hardware inputs. This is to access hardware functionality of the device such as, volume buttons and rotate device. For this project these inputs are not necessary.

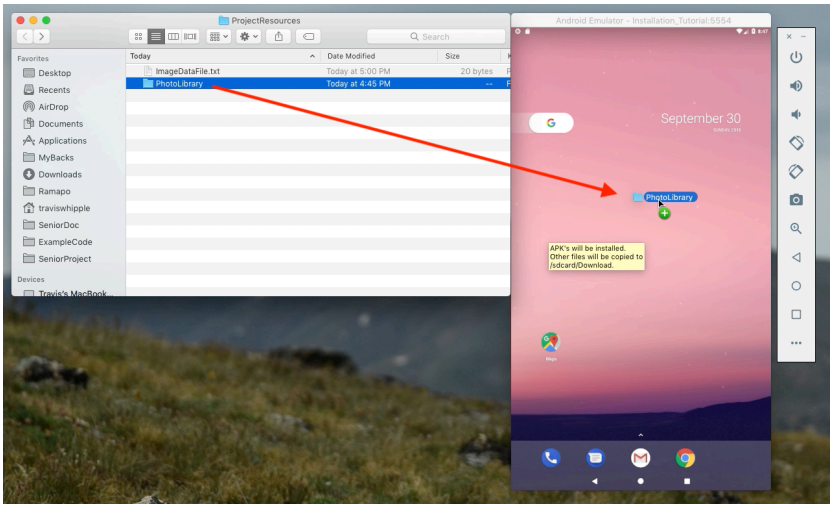
To close the Virtual Device, select the small x on the top of the Hardware functionality buttons.

2.4 Initializing Data

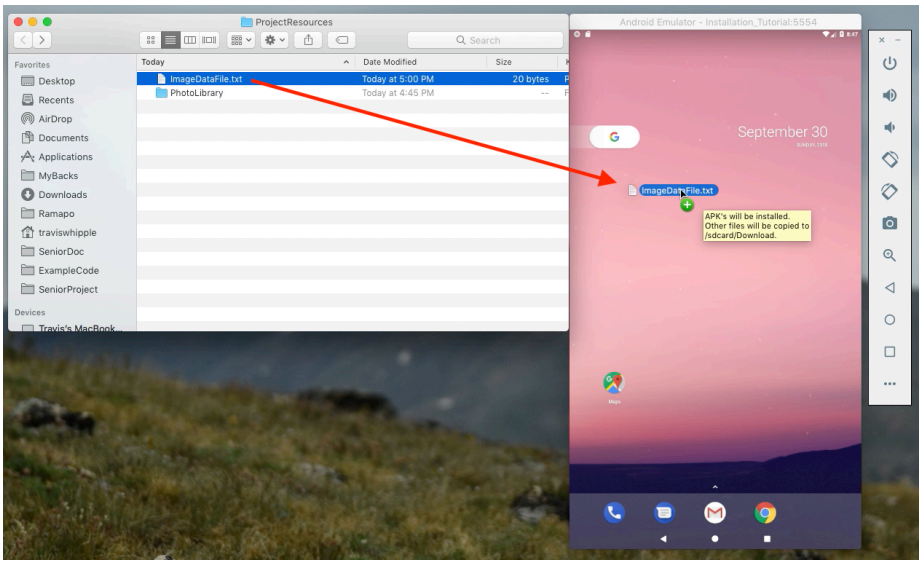
Once the Virtual Device is up and running we can now add the PhotoLibrary included in the ProjectResources folder. This folder contains a PhotoLibrary folder and a ImageData.txt. The PhotoLibrary folder contains all images that were used during the demo. Since the Virtual Device contains no images, we can simply add them by dragging and dropping the PhotoLibrary folder onto the Virtual Device. This will add the folder to the downloads folder within the Virtual Device. It is important that the PhotoLibrary folder be added as is and not the individual images them selves as the program will only read images from the PhotoLibrary folder.

The ImageData.txt file can also be added. This is optional but recommended, as the ImageData.txt file Without this file no tags will be associated to any image, meaning that the program will have to analyze each image for tags which is done in the background. For instant access to all of the programs features, add the ImageDataFile.txt in the same manner as the PhotoLibrary folder.

Adding the PhotoLibrary folder:

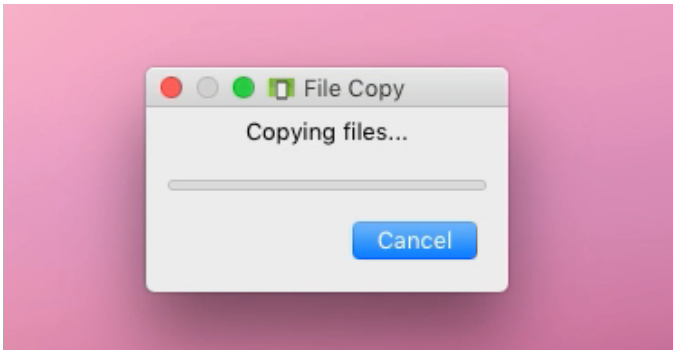


Adding the ImageData.txt file in the same manner:

A 

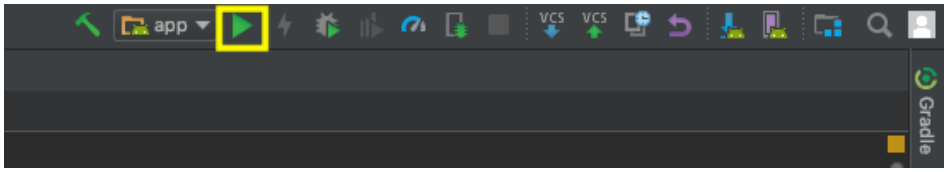
small text box will appear while adding a file: “APK’s will be installed. Other files will be copied to / sdcard/Download.”

After adding a file wait until the “File Copy” window disappears.

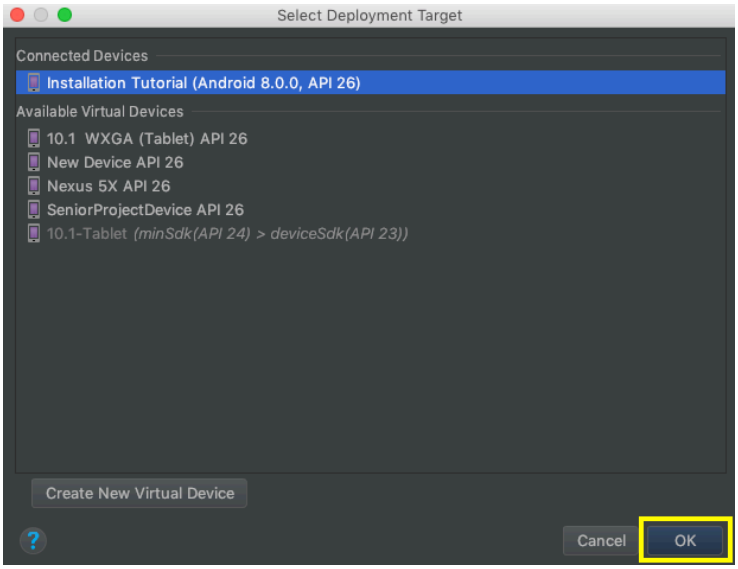


2.5 Running The Project

Finally we are ready to run the program on the device, to do this click the “Run” button at the top right of the screen inside of Android Studio.



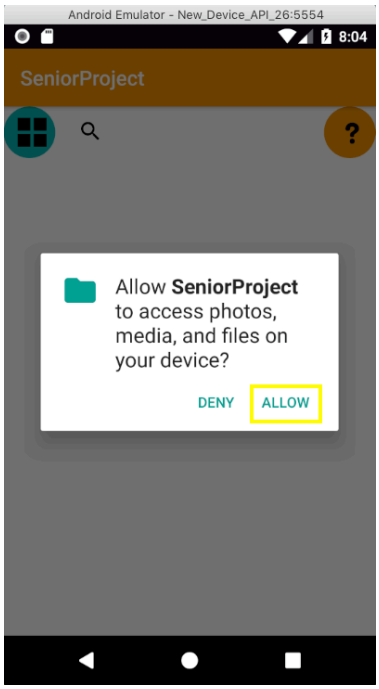
Then select the device from the list of “Connected Devices.” This list should include the current Virtual Device open, as we can see “Installation Tutorial” is connected as it is currently running. Once selected select “OK” and Gradle will build the project and the project will start.



2.6 Accepting Permissions

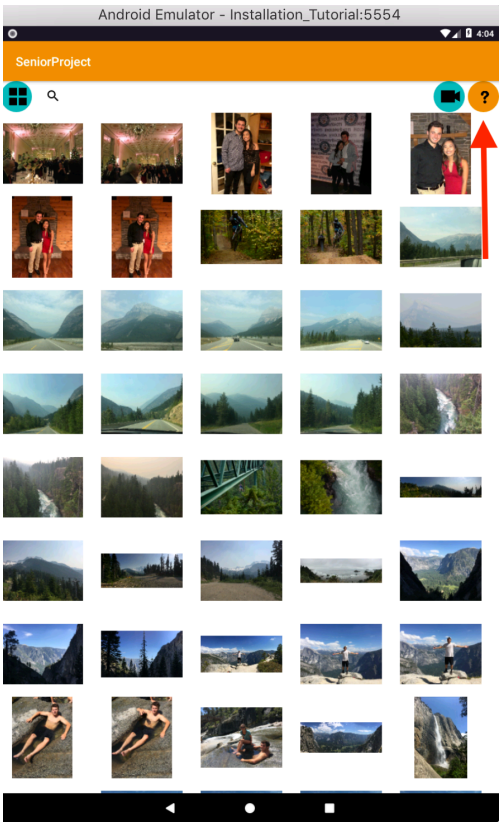
Because this app needs access to users files and access the camera, a read permission, write permission and camera permission are prompted to the user. If a user declines access to files then no images can be loaded/saved and the app will display a blank photo gallery. If a user declines camera access then the camera button will be hidden and no new images can be taken within the app.

After allowing the permissions, the app will load all images into the view and start analyzing images for faces.

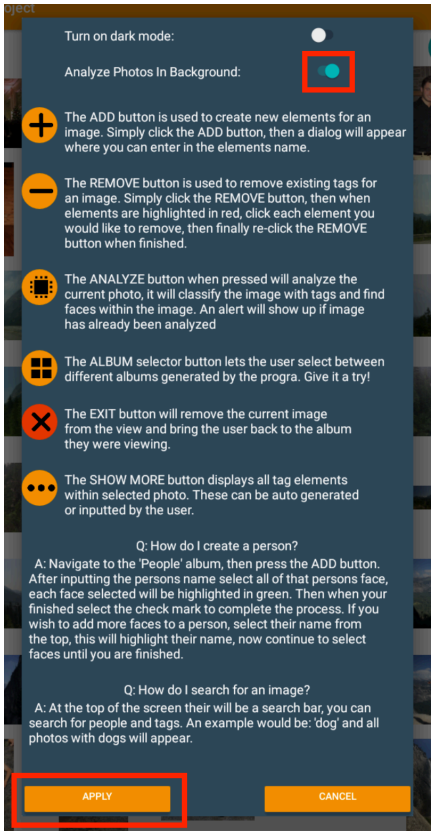


2.7 Analyze Photos in Background Setting

Select the Help menu at the top right of the application. This will display a small tutorial of what buttons do and how to generally navigate the application. On the right we can see that the Analyze Photo in Background is checked. This feature is turned on by default but can be turned off at any time. Analyzing all photos in this library takes around 3.5 minutes on a device, although emulators run slower pushing the time closer to 4.5-5 minutes.



2.8. Trouble shooting



A Gradle build error might occur if Android Studio was used in the past with libraries that conflict with the libraries used by this project. This is because Gradle's cache file contains old project libraries. If this error occurs, usually displaying "Execution failed for task ':app:transformDexArchiveWithExternalLibsDexMergerForDebug' " or a similar "Unable to merge dex" message, a simple fix is below. This error is not caused by the project but by Android Studio its self. This can be a confusing error as it is not caused by anything within the project, instead it is an error from Gradle.

Fix: Simply select from the menu bar: Build > Clean Project. Then re-run the program!

3. Project Design

I started my project by creating the Activity that will be initialize when the program starts, MainActivity. Within this activity I created GridView and attempted to load images into the view. This worked well in the beginning while my project was still small as my library only consisted of around 20 images to increase the start up time of the application. I started by loading a full image into a small ImageView that was only about 400x400 pixels. After adding more than 25 images my program start to run extremely slow. At this point I only had a basic gallery viewer. I found that my application was consuming an enormous amount of memory and if I tried to increase the size of the library, the application would crash due to out of memory errors. I tried compressing down the images into a bitmap and rescaling the bitmap to be only 50x50 pixels. This fixed the problem only temporarily, as with adding more images the same error would occur. I later found out that this is due to how Androids Views are stored in memory.

A GridView in Android will try to write every element to the screen even if the specific element was out side of the rang of the screen. The GridView would have to initialize all elements before it could be displayed to the screen. This halted progress on my application for awhile as I was trying to find another way to load images into the view. Finally I found this library called Glide. Glide does all memory management for displaying images to the screen. Glide is especially efficient at this because it will cache images it had displayed in the past to allow that image to be loaded into the view faster as well as will load images on an as needed basis. When a view goes out of memory, Glide will release it from memory and load it back when the view is visible. This made displaying images in a GridView much easier and made the program much more efficient.

I then went on to create an adapter that would inflate the GridView so that views can be added and removed dynamically. I created the class ImageAdapter which extends BaseAdapter. The ImageAdapter would be initialized first with every image's absolute path that will be displayed in a given GridView. When the GridView is displayed on screen it will request a View from a given position, here I create a new ImageView with the corresponding image. To display the image within the ImageView I again used Glide.

Once I had a basic gallery viewer created I wanted to add the ability to create tags for each image. I was unsure of what type of data structure I wanted to use for this. At first I had an array of tuples, the image and the associated tag. This quickly did not work as when you add another tag to an image it would create a new tuple copying the image twice in the data set. I needed a way for tags to point to images and images to point to tags. But this many to many

relationship would cause my program to get stuck in infinite loops when searching for similar photos.

I then thought a map would be the solution. Maps are key value pairs and that is exactly the kind of relationship I needed. I needed a tag to map to an image, and an image to map to a tag. Having created two maps one of tags pointing to images and another with images pointing to tags, this quickly became too complex to run any sort of similar images search. A big problem occurred when I would try to add and remove tags. When I would add a tag I would have to add the relation to both maps making for a lot of redundant code. Finally I went with using ArrayLists of custom objects: ImageObject and Tag.

The ImageObject class would hold an ArrayList of Tag objects. I wanted to name this class Image as it holds all of the information about an image, but that class is already defined in Java. This list could be altered as well as check if something is contained within it. This made for checking if a Tag has already been added very easy, and allowed for cleaner code. Clean code is always good. Following this logic I went ahead and did the same for a Tag object.

The Tag object would hold an ArrayList of ImageObject that the Tag was associated with. The implementation of these two classes allowed me to add to each others collections as well as remove easily. Now that I had a basic relationship established I wanted to allow for the user to add and remove their own Tags for images. I tried to create a layout that would display to the user all the tags currently in an image. Since I had defined all of my View elements in my Layout file I decided to create another view for tags in the Layout file.

Designing layouts was a much harder process than I first imagined. I thought writing the back end code would be the hardest part. Although it was challenging, creating layouts was very

time consuming, especially when I started to create the layouts dynamically. Creating layouts in an XML file it not that difficult. There are hundreds of different attributes that can be added and each one drastically affects how a layout will be displayed.

I decided to display the gallery on the left and the image selected on the right with all of the images tags displayed in a text box below it. This was a proof of concept idea just to see that tags were being assigned to images. All layouts at this point were created in XML files. Android studio includes a nice feature where you can view the XML file to see exactly how it will display on the screen. This works great for an application with fixed view, but when I started to create views dynamically nothing seemed to go as planed.

3.1 Creating Dynamic Layouts

The first step in creating dynamic layouts was to have all images loaded into the view at run time. Up to this point I had defined the images in the XML just to get the project up and running in the beginning. Once I started to try to create dynamic layouts Android Studios very nice Layout editor would now render useless to me. I started small at first, just load text views into a LinearLayout that displayed the current images Tags. This went well and I thought that creating dynamic layouts would be a breeze. Theres a saying in computer science; 80% of the project will take 20% of the time and that last 20% of the project will take 80% of the time. I feel, while this is accurate, that it is completely the opposite when creating layouts.

I felt as if I was spending 80% of my time on just 20% of the layout. Getting layouts created dynamically to flow with each other gets exponentially harder the more layouts you

create. To load my images dynamically into a grid pattern, as every other gallery viewer does, I created a LinearLayout of LinearLayouts. In my head it was a good idea, I would have one main LinearLayout with a vertical orientation, then keep adding a new row of elements, and repeat. This worked only for a small amount of time.

I started adding images of different sizes with different aspect rations which would cause image to overlap each other. I then thought I would scrap my idea of creating views dynamically and just have multiple Activities that would have their own layout defined in XML. When creating a new Activity in Android the current Activity is suspended and the next Activity starts. The original Activity is paused and there was no way for me to access the data from the original Activity that stored all of the ImageObject and Tag relations. Data can only be sent to a new Activity in the form of a Message.

A message has a string key and a value. This is good for sending individual values with respect to a String to the new Activity, although object cannot be sent using this method. I thought about saving to a file and having the next Activity read the data from the file, but this seemed redundant. I decided to take a break from the multiple Activity design and started to focus on finding a way to analyze photos.

3.2 TensorFlow

I found that TensorFlow has a classifier that can be built and run on the Android Device, where as other API's send a request where the image is processed. Seeing that TensorFlow can run on the device I decided that this would be the classifier I would use. TensorFlow for image classification has been pre-trained to analyze for just over a thousand different labels. I

downloaded a demo of the NeuralNetwork off of the Google Play Store, and started playing with an application that would display what the camera currently sees.

After playing with this program for a few days I decided that it would be what I use to classify images. What I overlooked is that TensorFlow only returns one label for a given image. TensorFlow was only returning what it thought, out of its database, was in the image. The accuracy for TensorFlow was also very low, compared to Google Cloud Vision; the API I used for the final project. I was overlooking TensorFlow's low scores for its ability to run without an internet connection. I valued this feature much more that I should have as its accuracy and number of returned labels were minimal. In the end I ruled out TensorFlow as a viable option and continued to search for not only an accurate API, but one that was free.

3.3 Google Cloud Vision

After doing research I have found that Google's Cloud Vision API was free to use for up to 1000 requests per month. A demo on Google Vision's website has a simple demo where you can drag and drop an image to get all labels and confidence values. TensorFlow for reference would rarely return a label with a confidence over 70%, meaning that TensorFlow was not that certain that the returned label was actually what the picture contains. After dropping a few photos into Google Vision's API demo on their website, I was getting 15+ results with very high accuracies. This led me to choose this as the classifier to be used for my project.

Implementing Cloud Vision was no easy feat for Java. All requests must be done in an AsyncTask defined within the current Activity. This is because the AsyncTask access the internet

and can not be done anywhere but inside of the current Activity. To make a request you first have to create an annotated request. This annotated request at its base is a JSON put request. First I had to create a VisionRequestInitailizer and pass this object my API key. The KEY I have was generated when I created a Google Cloud Vision account. They key is still in the project, and works fine. I then had to build a VisionAnnotationRequest object from the GsonFactory object. I then had to convert the image for the request into a bitmap, scale it down and convert the scaled Bitmap into a ByteArray. This is because the VisionAnnotationRequest takes a Vision.Image object encoded in base 64. To achieve this, I had to compress the Bitmap into the ByteArray. Now that ByteArray contains all of the bytes of the Bitmap, we can encode the ByteArray through the Vision.Image class's encodeContent function. I thought at this point I was finished with the request but I would get errors that my API key was invalid. This was a phantom error in that my KEY was correct, but the request had not been fully annotated.

I then had to add the features I wanted for the request, which is LABLE_DETECTION, and I set a max result value of 20 labels. I finished the request by adding the feature I wanted to the AnnotateImageRequest. I then had to create an ArrayList of AnnotatedImageRequest and add the annotatedImageRequest I built to the ArrayList. I was then able set a BatchAnnotationRequest with the ArrayList of my one annotation request. All calls to Cloud Vision have to be of a Vision.Images.Request object. To create this object you have to annotate the BatchAnnotationRequest using a Vision.Builder created from the GSON and httpTransport objects. Finally I put all of this into its own class as it was very long and technical. I then pass the annotated request to the AsyncTask where the request can be executed. Once I finally got this working I was able to analyze images with higher accuracies and more results. For this program,

the more data the better. For each label detected in an image I would need to create a relation between the two. To do this I created the class ImageManager, and as the name suggests it will manage all relations between Tags and ImageObjects.

The ImageManager class would be responsible for containing all ImageObjects and all Tag objects. These objects are stored within an ArrayList, one for ImageObjects and another for Tags. Relations can be added and removed between a given image and tag. When a request to make a new relationship is initiated this class will check if the given image and tag exist within its data set, and if they dont it will create a new instance of the given object and add it to its data set. Then the ImageManager will add the given ImageObject to the Tag's list of images and vice versa. Now the ImageObject contains the Tag that describes it, and the Tag contains the image it defines. I ran into a problem with my ImageManager class in that I could not retrieve Tags for a given ImageObject or vice versa.

For a given ImageObject my ImageManager class would return no Tags even though I have explicitly created multiple relations for the given ImageObject. While debugging I found that every time I would add a new relation to an ImageObject an entirely new ImageObject would be created. This was due to the contains function within the Collections object. Since an ArrayList is a Collection, the Collections.contains function checks for equality using the objects equals function. Equals will return true if two objects are identical in all aspects, including the objects reference. Therefore when creating a new ImageObject with the exact same parameters as an existing one, they would return as not equal as they are two different instances. To fix this I would adjust my ImageObject and Tag classes to Override the default equals function. ImageObjects would now be compared for equality based on their images absolute path, as the

image file for each ImageObject is unique. The Tag object would override its equals function to compare two Tags based on the Tag's name.

3.4 Face Detector

My next task was to get facial detection working. I have found a library also by Google while doing research for Cloud Vision that does facial detection. This library is very simple to use. I simply create an instance of com.google.android.gms.vision.face.FaceDetector and initialize it to set it to detect all landmarks on a face, and set it in accurate mode. Then I could just simply pass the detector a Fram object, which is set form a Bitmap of the image to be detected, and the directory will return a list of Vision.Face objects. I created a class, FacialDetection which would create a Thread where it can analyze an image for faces without halting the application. I would then parse the Vision.Face object to get the exact position of a face within an image. For some reason when creating a Bitmap to send to the detector the Bitmap would be rotated causing the detector to detect no faces. I then created my own BitmapResources class.

BitmapResources would be responsible for creating a bitmap from a given image's path. This would then check that a Bitmap created matches its original image's orientation then rotating the image by the amount of degrees difference using a matrix. This solved the problem of Bitmaps being created in the wrong orientation. I continued to add functionality to crop an image by specified boundaries.

To parse the face returned by the FaceDetector I created a FaceObject and FaceBoundaries class. A FaceObject holds all information pertaining to a face found within a given image. A FaceObject has a member FaceBoundaries, which holds all information about

where a face is located within a given image. FaceBoundaries will take a Vision.Face object and retrieve its location within the image. The Vision.Face object contains a faces location as well as the width and height of the face. Using this information I can then create a hypothetical rectangle around a portion of an image that would only contain the found face. Using this information in my FaceObject class I could then create a Bitmap cropped from the original image using my BitmapResources class. This cropped Bitmap would be saved into a designated “Faces” directory so that it can later be displayed. Now that the application could detect faces and labels for a given image I could return to my dilemma of having a single Activity application.

3.5 Fragments

Creating whole layouts dynamically from scratch was a very difficult task and resulted in a large amount of trial and error. This is was an optimal approach to creating new views as I would have to run the program for every minor change to ensure that layouts were displayed in the correct manner. Compiling and running the application would take between 1-3 minutes. This slowed down development until I found another way of displaying views by using Fragments.

A placeholder for a Fragment can be added to the MainActivity’s XML layout file. These placeholders can then be inflated from a Fragment with its own XML layout file. Having now found a way to create different views without the need to create them completely from code I could speed up the development process. I created my first Fragment, SimilarImagesFragment. This fragment would be responsible for displaying a single image with all similar images below.

This Fragment will be created when the user selects an image from the photo gallery. It contains buttons to perform the following actions; Close image, view tags and analyze the current image. Below the current image similar photos will be displayed along with photos taken

from this day and people found in this image. To display similar images and images from this day, I had to add functionality to my ImageManager class, as well as create a People class.

The ImageManager class would implement a method to determine which other photos in its data set were similar to a given image by analyzing the images tags. It will take the images top 40% of tags with the highest confidence as long as their confidence is above 60%. The top 40% of tags best define what context of the image is, and filtering out results with a confidence lower than 60% prevents from using tags that may not best describe the image. With these tags, we can then get all images that tags associates with whos confidence is greater than 90%. If we are currently analyzing a photo that contains a Dog whos confidence is 70%, we will then only get other photos with the tag Dog whos confidence is 90% or higher. This is to limit how many results are returned for a given image, as well as ensure similar images are accurate. These found images are then returned in an ArrayList.

ImageManager would then have to include a method to get all images from the same date as a given photo. For this the images in the data set would have to be sorted on their date. To do this I modified ImageObject to implement Comparable<ImageObject> class. By implementing this class I could have a collection of ImageObject compared based on the date they were taken. Back in my ImageObject class I could get all images from the same date by getting the index of the current image within the data set and looping though all images above and blow the index as long as the current image has the same date. All images found will be added to an ArrayList and returned.

Now that I can display similar images and images from the same date I wanted the ability to display all images of a person within the current image. Since I already have all faces found in

every image, I needed a way to group all faces. I tried to find free to use facial recognition API's or libraries but the only libraries that I found were neural networks that would have to be trained on a set of faces for a specific person. I could not train these neural networks dynamically since faces found belonged to an unknown person. So I decided to have the user select which faces matches a person.

3.6 Person Class

I created a Person class to contain which faces map to a given person. This class would contain all information about a Person including their name, a unique ID and all FaceObjects that contain this persons face. To create the interface for a user to create and modify a Person I created a PersonFragment class. This will display every FaceObject's cropped face under the "Faces" directory. The user can then select every image that contains that persons face. Once finished the user can select the check mark button to finalize the newly created Person. This Fragment will display all people along with their faces. The user can edit an existing Person by selecting the button at the top with their name, entering them in selectingFacesMode. Within this mode they can add a face to the selected person, or remove faces from them, when they are finished they can select the check button to finalize the changes. If the user selects the remove button they will enter removePersonMode. Within this mode they can select one of the buttons displaying a persons name to remove that person. After they select a person they will be brought out of removePersonMode. All faces associated with that person will now belong to the "Unknown Faces" collection at the bottom of the screen. These faces can then be added to any person.

3.7 Serialization

Finally I needed the ability to save and load all image data so that the application would not have to re-analyze every photo. To do this I created the Serialization class that will handle saving and loading to a comma separated text file. When loading a text file the class will check if the directory “SeniorProject” exists and if not create it. A directory “Faces” will also be created under this directory where all cropped face images are stored. Within the SeniorProject directory all image data will be saved under the file ImageDataFile.txt. This file will store all data about images, tags, faces and people. The file is broken up into 3 main parts, Images, Faces and People; which preface the type of object that was saved. Every ImageObject will be written to the file on a new line, each line will contain the path of an ImageObject, whether the image has been analyze for labels and faces, followed by all tags associated with that image and their respected confidence. Next each FaceObject will be written to the file containing the face’s original image path, the path of the cropped face which is named using the face’s unique ID and the face’s FaceBoundaries. Finally all people will be written to the file containing the persons name, their unique ID, followed by the ID’s of all the faces associated with that person. Below is an example of a small save file:

```
IMAGES
/Download/PhotoLibrary/IMG_01.jpg,true,false,function hall(0.934),ceremony(0.924),event(0.825),wedding reception(0.774)
/Download/PhotoLibrary/IMG_02.jpg,true,true,land vehicle(0.984),mountain biking(0.973),cycle sport(0.967)
/Download/PhotoLibrary/IMG_04.jpg,true,true,dog(0.968),dog like mammal(0.966),dog breed(0.942)
/Download/PhotoLibrary/IMG_05.jpg,true,true,car(0.981),motor vehicle(0.98),vehicle(0.849),antique car(0.847)
/Download/PhotoLibrary/IMG_06.jpg,false,true
/Download/PhotoLibrary/IMG_07.jpg,false,true
FACES
/Downloads/PhotoLibrary/IMG_06.jpg,/SeniorProject/Faces/1.jpg, 222, 1604,304,308
/Downloads/PhotoLibrary/IMG_07.jpg,/SeniorProject/Faces/2.jpg, 1449,965,404,506
PEOPLE
Travis,1,2
```

Here we can see that the first image saved, IMG_01.jpg, has been analyzed for tags, but not for faces. This image contains the tags with respected confidence values: “Function Hall”

with confidence .934, “Ceremony” with confidence .924, “Event” with confidence .825 and “Wedding Reception” with confidence .774. We can see that IMG_06.jpg and IMG_07.jpg have not been analyzed for tags but have been analyzed for faces. Under the FACES section we can see that these two images do in face contain faces. Their face image is stored under the file 1.jpg and 2.jpg respectfully. Finally under the PEOPLE section we can see that Travis has a unique ID of 1 and contains the face with unique ID 2.

For loading images the serializer would first need to be initialized with the ImageManager used to access the data set. The serializer would first load all images from public directories using a Cursor iterate through all all image media in the external storage directory, which is where all public files are stored. It would then get the date the image was taken from the current Cursor’s media.DATE_TAKEN column. We do not want to add faces as an image to the data set as everything added will show up in the users gallery and faces should only be shown when created a new Person object, so we check that the current images path does not contain the Faces directory. If the image is not from the Faces directory, then an ImageObject will be created from the images path and date data and add it to ImageManager’s data set. Next images will be loaded from the users private directory, this directory contains all images taken within the application. They are loaded by checking each File under the “Pictures” sub directory of the applications private directory. All images found will then be loaded into the ImageManager.

The Serializer will now check that the file “ImageDataFile.txt” existed under the directory SeniorProject. If it does not, then it will check if the file existed under the Downloads folder. If the file was not found then the serializer would not load any preexisting relations, otherwise the data will be parsed from the file.

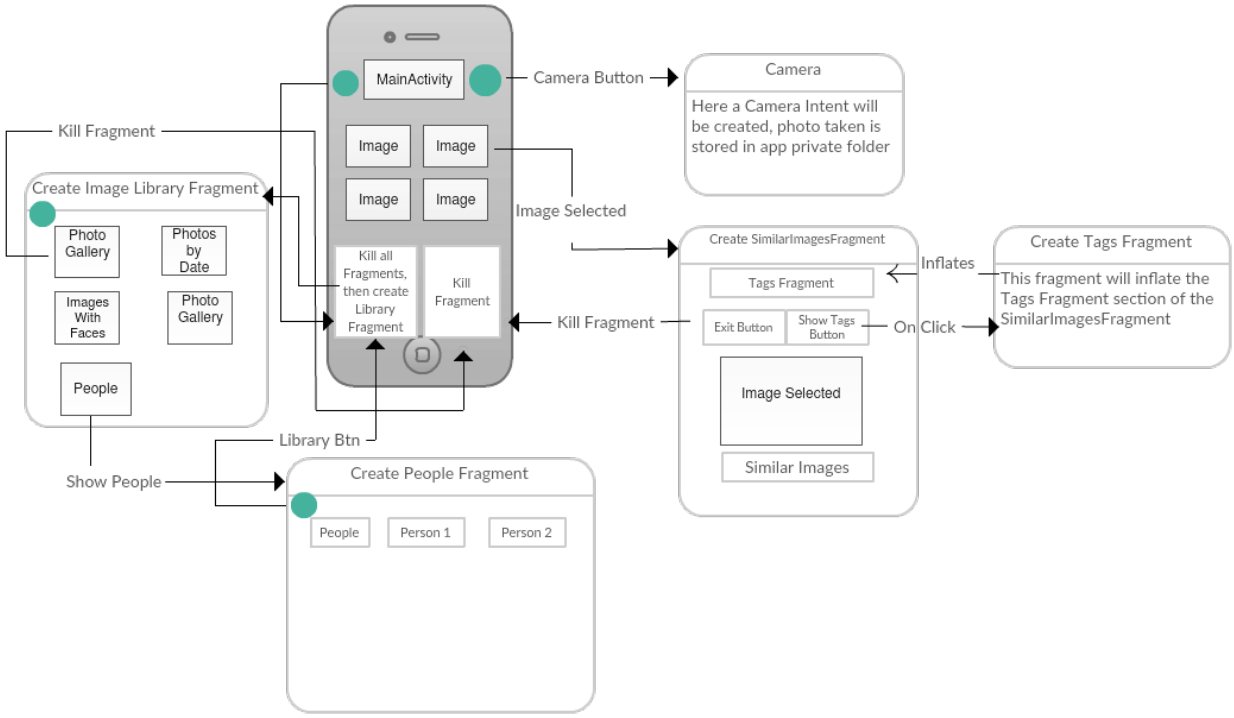
First image data will be parsed, and a new ImageObject will be created. This ImageObject will be set with the corresponding has been analyzed and has detected flags, next for every tag associated with this image the found relation will be added to the data set along with the tags confidence. If an image does not contain the correct amount of elements, then any relations for this ImageObject will not be added.

Next Face data can be parsed from the file. First the serializer will ensure that the face data contains the correct number of elements. All faces should have the same number of elements as they do not contain a dynamic list such as images do. For each face the serializer will create a new FaceObject and initialize it with the original images path, the face's cropped face image path, the face's unique ID and finally the face's boundaries. The newly created FaceObject can then be added to ImageManager's list of FaceObjects.

Finally People data can be parsed from the file. For each person the serializer will create a new Person object and initialize it with the persons name, and unique ID. Then all faces associated with this person will be added to the Person object and finally added to ImageManger's list of People.

3.8 Controlling Fragments

When a user selects a photo or the ImageLibrary button, a Fragment will be created on the current view. To prevent views from over lapping on top of the photo gallery, a View that will write over the gallery will be drawn to cover up the gallery. The bellow diagram shows how fragments are created.



3.9 API Key Out of Requests

While in the last few weeks of creating this application I ran out of requests for Google Cloud Vision. I was running the app many times as I was ensuring that the application worked correctly on my device and on an emulator. While doing this I reached my limit of 1000 requests per month. This was a concern of mine from the beginning as 1000 requests per month does not allow for much testing on an image library with about 140 photos. Having now been locked out of my KEY I decided to revert back to using TensorFlow for image classification.

Since I already had a working version of TensorFlow, implementing it back in was not that difficult. On the other hand, because my KEY was restricted the face detector would no longer work either. Thankfully Android includes its own FaceDetector. This face detector returns only returns where the center of the face is within an image, and the distance between the

persons eyes. This made cropping out ones face from an image as I no longer had the face's boundaries easily accessible. I decided to crop the image from the center of the face and create a square around that face with a length that was 2 times the width of the persons eyes. This resulted in some faces being cropped fine, while others would clip the persons face if the person was at any sort of angle. Android's FaceDetector also ran much slower, it would take about 8-10 seconds compared to Google's Vision which took just under 2 seconds. This is because the FaceDetector was running on the device and the device I am using is not particularly powerful, and I wanted this application to run smoothly on a range of devices. The detectors accuracy was also extremely low, if there was any sort of noise, or really anything else but a face in an image, it would return back garbage results. The detector would return an accuracy so I could filter out the bad results. The problem with this is that not many faces would return a confident value. So by filtering out low accuracy faces, I would be throwing out perfectly fine detected faces. I will put results of this detector in the *Test* section. After not being satisfied with the results of this detector I was ready to pay for Google's Cloud Vision services.

Luckily there is a way to authenticate the request even with a restricted API key. To do this I wold have to create a signature using a JSON credentials file created from my Cloud Vision account. After failed attempts for not being able to set the environment variable GOOGLE_APPLICATION_CREDENTIALS within my project, I found on their website a way to get a signature to use that is automatically created when the application is built. To do this I have implemented a class from their website PackageManagerUtils. This class was not written by me. There was no license or distribution agreement indicated anywhere about this class, and the lack of any Vision libraries included leads me to believe that this is a general class to get the

applications signature. I can now use my API key without any restrictions for my application so that it is not in production.

4. Test

After Getting Google Cloud Vision to work I wanted to decrease the time it takes for an image to analyzed. I first started with the below image by passing it to google and allowing for as many labels as it could give me. The first time I sent the image it was a resolution of 6000x4000 pixels. This is a very large image to be sending through the internet. The full scale image returned 37 labels. Of the labels returned, 19 had a confidence greater than or equal to 70%. Below are my findings for the same image passed at different sizes, along with all tags found on the next page.

Image Size	6000 x 4000	1000 x 666	200 x 133	50 x 33
# Labels above 70%	19	18	18	6
Total # of Labels	37	35	31	10

Image Size	6000 x 4000	1000 x 666	200 x 133	50 x 33
Time (Seconds)	8.1	2.6	2.3	2.4

6000x400 8s

Mountainous Landforms 95%
Mountain 94%
Ridge 93%
Wilderness 92%
Mountain Range 92%
Rock 87%
Sky 87%
Outdoor Recreation 85%
Valley 82%
Cliff 81%
Terrain 78%
Hill Station 78%
Adventure 77%
Geological Phenomenon 77%
National Park 72%
Alps 72%
Highland 71%
Escarpment 71%
Extreme Sport 70%
Geology 67%
Tourism 65%
Hiking 64%
Tree 63%
Mountaineering 63%
Landscape 60%
Mountaineer 59%
Mount Scenery 58%
Summit 57%
Glacial Landform 56%
Hill 56%
Arête 55%
Fell 55%
Recreation 54%
Elevation 53%
Massif 52%
Cloud 50%
Backpacking 50%

1000x666 2s

Mountainous Landforms 94%
Ridge 94%
Mountain 93%
Wilderness 92%
Mountain Range 91%
Rock 84%
Outdoor Recreation 83%
Valley 83%
Cliff 82%
Hill Station 81%
Sky 81%
Geological Phenomenon 80%
Terrain 78%
Adventure 77%
National Park 74%
Escarpment 74%
Extreme Sport 71%
Highland 71%
Geology 68%
Alps 67%
Tourism 64%
Mountaineer 63%
Mount Scenery 60%
Mountaineering 59%
Summit 58%
Hiking 58%
Massif 58%
Landscape 56%
Fell 56%
Glacial Landform 54%
Tree 53%
Recreation 51%
Arête 51%
Mountain Pass 50%
Hill 50%

200x133 2s

Mountainous Landforms 94%
Mountain 94%
Ridge 94%
Wilderness 92%
Cliff 89%
Mountain Range 89%
Hill Station 88%
Geological Phenomenon 83%
Outdoor Recreation 83%
Sky 83%
Rock 82%
Adventure 80%
Valley 80%
Terrain 79%
Extreme Sport 78%
Tourism 74%
Escarpment 73%
National Park 72%
Summit 63%
Geology 62%
Alps 61%
Glacial Landform 58%
Mountaineer 58%
Massif 56%
Recreation 56%
Tree 54%
Mountaineering 53%
Elevation 53%
Landscape 52%
Mount Scenery 52%
Hiking

50x33 2s

Sky 95%
Mode Of Transport 88%
Geological Phenomenon 84%
Atmosphere 81%
Cloud 80%
Water 73%
Meteorological Phenomenon 56%
Fog 55%
Tree 54%
Wave 52%

These findings led me to choose to send request with a maximum image size of 800x800. This allows for fast analyzation wile not compromising accuracy too much.

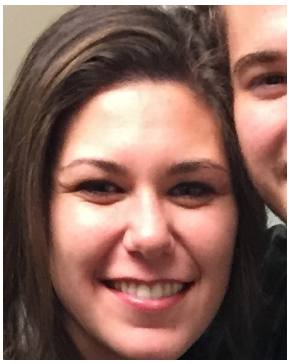
Now I wanted to check how quickly images could be analyzed. To do this I created an image library with 96 photos and ran the facial detector on each one and I came to the following results for analyzing for both faces and labels:

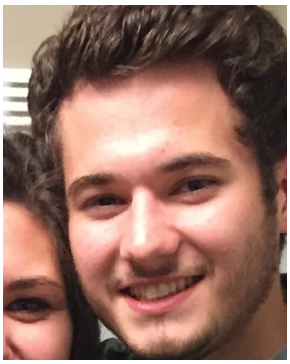
Faces: start - 19:51:04.296, End - 19:53:07.525 - Elapsed: 2.0538 minutes or 1.28s per image.

Next I ran the same text for the label detection with the same 96 images all being cropped down to a maximum size of 800x800, results as followed:

Labels Start - 20:04:02.989, End - 20:06:06.404 - Elapsed 2.0569minutes or 1.28s per image.

These results surprised me as to how close they are, after all they are using the same service.



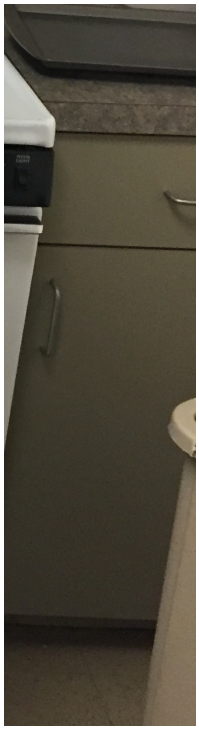


Google Cloud Vision Face Detector results vs. Android's FaceDetector for the following image.

Google Cloud Vision results:

Produced 2 highly accurate faces with accurate boundaries.

Android's FaceDetector results:



As we can see Android's FaceDetector produced many garbage results with only two images actually capturing a face. The other three do not even contain a face. This is a result of the detector having a very low tolerance to any sort of noise in an image.

5. Classes

This section defines each Class implemented within my project. It is assumed that the reader has a general knowledge of simple data structures.

5.1 AnnotationRequest

This Class will create an annotated request to access Google's Cloud Vision API. It will initialize the JSON request with proper authentication by using this apps signature. An API key is

needed for all requests used and it is set for label detection with a maximum hit rate of 20. The image to be analyzed is compressed into an image with a size of 800x800, this is to save on bandwidth as well as speed up analyzing time. An image of this size still produces accurate results.

5.2 BitmapResource

This class is used to create and modify bitmaps. It will ensure that all bitmaps created are rotated back to their original orientation. I was encountering problems creating a bitmap from a file as they would sometimes be rotated by 90 degrees. This class has functionality to:
Get a bitmap from a file: Which will ensure that its orientation matches the original orientation.
Crop to Face: will crop a face on the given FaceBoundaries object, which I define.
Scale a bitmap down: Can scale a bitmap to a given maximum dimension. This is used for scaling bitmaps down to 800 x 800 for sending label requests.

5.3 Face Boundaries

This class holds where a face is located within an image. It contains 4 variables that store the coordinates for a box to be framed around a found face. When cropping a bitmap down to just a face, the class will ensure that the boundaries of the face do not exceed the boundaries of the bitmap. To do this it will adjust the bounds of the face. This class can be serialized, returning a string containing every member element separated by a given delimiter.

5.4 Face Object

This class is created for every face that is found in an image. Each FaceObject has a unique ID that is assigned to it in the ImageManger class. This class is comparable to other FaceObject. The two objects are compared on their unique ID. A bitmap of just the face can be returned which will crop the original image down so that it only contains the face. This bitmap is then stored under the “SeniorProject” directory in its own “Faces” directory. Here all faces are stored. They will not be viewed by the gallery, unless user is creating a Person.

5.5 Facial Detection

This class implements a Google Vision Play Store face detector. It will loop take a given image’s path and then pass it to the detector in full resolution, this is because the face detector will return a Vision.Face object containing all information about the location of the found face. These locations are coordinates in pixels and if I compress the image, then creating the bitmap from the scaled down image will have extremely low quality. Maintaining the original image size also allows for faces that are farther away from the camera to be detected, resulting in less faces being un-detected. This object is then passed to FaceObject to be parsed and to get the cropped face bitmap.

5.6 Image Adapter

This class is where I create my custom Image Adapter the image adapter is used to load images into a GridView. To do this I use Glide for smooth loading. All ImageViews created have a maximum width that is the width of the screen / the number of columns to display. Glide is set to clear its cache every 5 minutes. This is to prevent Glide from loading an old image that was deleted or replaced with a new one.

5.7 ImageLibraryFragment

This class is where the user can select between different Libraries. There is the Image Gallery, which will call MainActivity to kill this process and display its own Image Gallery. It has a library for displaying photos by date. To display images by date the class uses internal functions to loop through all images in ImageManager and for every image that has a date that is different than the last, create a text view that spans all columns. This will break up the gallery into sections of same data photos.

The next library is the Faces Library. This will display every image that contains a face. It will not however display the copped face, just the original image that contains a face.

The next library is the Photos by tags library. This library will display images similar to how the dates library is displayed. It will break up a group of images when the Tag changes. These images are displayed by finding the most popular Tags and displaying a maximum of 20 imagers per Tag.

Finally there is the People Library. This library will display all people in ImageManager. This will create a new Fragment PersonFragment

5.8 PersonFragment

This class will display all people that exist. People can be created and removed. The user will select the add person button then enter the name for the new person. Then the user will then select all faces that belong the the created person. Then when all faces are selected, the user can select the finish button to finalize the Person created. If a user selects the minus button then selects a persons name from the top, the Person will be removed.

A user can edit a person by selecting one of the buttons at the top of the layout, then when the persons name is highlighted in green, they can add or remove faces for that person.

5.9 ImageManager

This class is essentiality the brain of the program. Here all relations between images and tags are added and removed. A relation between an ImageObject and Tag object can only be created once. Relations between Tags and ImageObjects can also be removed. This class contains an ArrayList of Tag objects that hold a reference to different ImageObjects, and an ArrayList of ImageObject with references to Tag's that define what is in the image. This class is also responsible for creating new People and for detecting faces within images. People created will be created with a unique ID that is created by taking the current highest ID of any given person and

increasing that value by 1. This will always create a unique ID for any Person created. The same unique ID allocation is used for FaceObjects created. This class is also responsible for detecting faces within the data set.

When analyzing for faces within a photo the ImageManager will first check that a given photo has not yet been analyzed for faces yet. This class contains a FaceDetectionThread class that will do all of the facial detection on a background thread so that the application can still function while images are being analyzed. All faces found will then be added to an ArrayList of FaceObjects contained within this class. Faces cannot be removed once created, unlike People which can be created and removed at the users will. This class has functionality to retrieve all people within a given photo as well as get all photos that match a users search.

The search function will try to match a users search string to either Tags or Person's. It will return all images associated with a Tag that partly matches the users search. People will only be matched if the search string is contained at the beginning of the persons name. This is to allow for the user to search for person by a nick name.

This class also has functionality to retrieve images that are similar to a given image or images taken on the same date. This function works by analyzing the top 40% of tags for the given image with the highest confidence scores and returning all images that those Tags contain with a confidence higher than 90%. This is so that fewer images are returned that will most-likely match the given image. To get images taken on the same date, the class will sort all images based on the date they were taken. Then it will get the index of the given image within the ArrayList and index above and below the given images index adding all images whose date is consistent with the given image. All results are then returned.

5.10 ImageObject

This class contains all information about an image. It stores all Tag's that are associated with it along with its respected image file's path. This class has flags about whether or not this image has been analyzed for faces and for tags. It also contains member variables to hold the day, month and year the image was taken. Each image can have multiple Tags, FaceObjects and Person objects associated with it. Because an image can have any number of these object, each one is stored in its own ArrayList. These ArrayLists are created dynamically and can update their size on an as needed basis. This implements Comparable<ImageObject> so that it can be compared within a Container to other ImageObjects. This is used to sort images based based on the date they were taken.

5.11 MainActivity

The main activity is where all classes and fragments stem from. This Activity will initialize a Serializer, ImageManager and an ImageAdapter. First this class will check that all permissions are granted for the application and request any permissions the application might need. The application will then add all images to the ImageAdapter and use that to inflate the main Photo Gallery. This gallery will display every image on the users device. When an image is selected it will create a new Fragment, SimilarImagesFragment. The current Photo Gallery will then be displayed as hidden behind a gray View. This view will be removed any time a Fragment

is removed from the view. There are a few buttons at the top of the app that call different functions within this class. The first is the help button which will display a menu showing what each button does and how to navigate the application. Within this menu there are options to set the application into dark mode, which will change the background from white, to a dark gray. The user can also turn off auto analyze photos in background from this menu. By default images will be automatically analyzed in the background. Another button at the top of this layout is the camera button. When this button is pressed the camera will be opened and the user can take a new photo to be analyzed. The next button is the Image Library button. When this button is selected a ImageLibraryFragment will be created where the user can select between different libraries to view. Finally there is a search bar at the top of the screen where a user can search for a key word which will then call ImageManager to get all images that pertain to the given search string.

Images analyzed within this class will call a function to analyze all photos within ImageManager's data set. This is done in a AnalyzePhotos thread that is put in the background so the user retains full application function. This thread will call the Label Detection Task where an AsyncTask is created to send an annotated request to Google Cloud Vision for analyzation. The result from the request will then be parsed and any labels found be added to the ImageManger as a new relation between the image being analyzed and all labels found. This class will ensure that all errors are caught and logged.

When a fragment is created MainActivity will ensure that no other Fragments are currently in the display. This is to prevent more than one fragment being viewed at a given time. Fragments can also call their respected RemoveFragment method which will remove them from

the view. This is useful for when the user is finished looking at an image, or simply wants to go back to view the Photo Gallery.

5.12 PackageManagerUtils

This class was not written by me, all credit goes to google. This class will get the current applications signature to use in the request to Cloud Vision so that a restricted API key can be used. This class saved me as I ran out of requests for my API key. The class has not been modified in any way. There were no license agreements or notions to acknowledge Google in being the creator of this function. That in part with the class not including any of Google Vision's libraries and the classes being simple and short, makes me believe that this is a general way of retrieving an applications signature.

5.13 Person

This class implements a Person. A person has a name, a unique ID and an ArrayList of faces associated with it. A person can be created or removed from the data set. This class is fairly simple as it is mainly getters and setters. FaceObject enter have to be entered by the user at this time. My goal for the future is to implement Facial Recognition, but free versions of this are hard to come by. This class has functionality to retrieve a persons unique ID, add and get FaceObjects, and get every image that this person is contained in. It does this by looping through all the FaceObjects associated with it and returning each FaceObjects original image path.

5.14 PersonFragment

This class will display the person fragment. The person fragment will display all faces found within every photo, along with pre existing people. The faces will be displayed in a grid broken up by individual people with the persons name prefacing all the faces associated with that person. Within this fragment the user can create a new person. The user will be prompted to enter the new persons name in a text box and then select all faces that belong to that person. When a user selects a face for the person the face will be highlighted in green so the user can see what faces they have already selected. Once the user has selected all faces for a given person they can then select the check mark button on the top right, this will finalize the request to make a new person.

People can be removed or edited. A user can remove a person from the data set by selecting the minus button. This will cause the button to turn red indicating that they are now in removal mode. Once in this mode they can select the button at the top that displays the persons name. Once they select the person, that person will be removed and its faces will be stored under the “Unknown Faces” collection at the bottom of the screen.

A user can edit faces for a specific person. When the user selects one of the buttons containing a persons name at the top of the screen the the button they select will change color indicating that they are modifying that person. The user can now click any existing faces under that person to remove them, or select faces from the unknown people collection. When the user is finished they can select the check mark button again to finalize the process.

5.15 Serializer

The Serializer class is responsible for loading and saving all data to the ImageDataFile.txt. This class will load all images from the public directory including images taken with the camera, images downloaded and images added to the public Pictures directory. Once all photos are loaded into the ImageManager, the Serializer will try to load all image data from the file. A directory will be created for saving Faces and the ImageDataFile.txt under the public directory SeniorProject. If the save file cannot be found within the SeniorProject directory then the Serializer will look for the file under the downloads directory. If no file is found then no tags will be added to the data set.

The Serializer will read the file line by line parsing each line depending on what type of data the current line is. There are three main sections that are saved to the file. The first is the Images data. This contains all data about an image including its image files path, whether or the the image has been analyzed for tags, whether or not the image has been analyzed for faces and then finally all tags associated with the image along with the tags confidence value. This data is separated by a comma making parsing the data much easier. All image data will be parsed into a new ImageObject that will then be added to ImageManagers data set.

The Serializer will next parse all FaceObjects that were created. Each FaceObject saves its original images absolute path, the path of the cropped image containing the face, the FaceObjects unique ID and finally the FaceObjects FaceBoundaries. This data is all comma separated and used to create a FaceObject to be added to the data set.

The las Section to be parsed is the People section. This section contains all People Objects that were created. Each line within the file will contain the persons name, followed by their unique ID, followed by the ID of every FaceObject that that person is associated with.

When writing to the file, the Serializer will save the data back in the same order that it was read. It will first save all image data, then all FaceObject dada and finally all People data. It does this by using each objects GetSerializedString function with a comma as the delimiter to divide individual elements.

5.15 SimilarImagesFragment

This class will display a selected image large with all similar images found below. Within this Fragment the user can view other phots taken from the same day as the selected image. They can view all similar images related to the current selected image. And also all people within this image will be displayed. There are a few action buttons at the top of this fragment. The three dots button will show all of the tags that were created for this image. These tags can be removed as well as create new Tags for this image. If the image has not yet been analyzed then the user can selected the analyze button to analyze the current image for faces and tags. When the user is finished viewing a selected image they can select the red x in the top left to remove the fragment from the current view.

5.16 Tag

The Tag class holds all data about a Tag including the tag's name and images that the tag references. This class has an ArrayList of ImageObjects that can be added and removed from. A Tag object can reference many different photos. Each ImageObjet that the tag points to also has an associated confidence value. This value is either determined when the classifier detects labels within an image, or when a user inputs a custom tag, the confidence will be 1. This is because the users Tags should have higher priority over auto generated Tags. It is assumed that the user will better understand the context of the image compared to Cloud Vision.

5.17 TagFragment

This Fragment will be inflated within the SimilarImagesFragment. It simply displays all Tag objects associated with the current selected image. Tags can be added or removed within this fragment. The user can select the plus button on the left to add a new tag to the list of tags. The minus button can be selected to remove tags. When the minus button is selected all tags will be displayed red and then the user can select as many tags to remove as they like. When the user is finished they can select the minus button again to get out of removingTagsMode.

6. Summary

This project was much larger than what I thought it would be. This is the largest project I have worked on and I realized the importance of not only commenting code, but taking notes to where I want to go next for the next time I work on the project. I feel that I have improved my skills not only at programming, but general problem solving overall. I also feel more confident in my ability to read and understand library documentation. This was an area that I severely lacked in as I thought these documentations where hard to understand and confusing. As it turns out they are presented in a logical manner, a basic understanding of the language and libraries was enough for me to use these documents to my benefit. I am also glad that I now have experience

implementing API's into a project as APIs are very common in the work place. I have also learned how to have better time management. A part of the program that I struggled with was time management. It is very easy to say "Tomorrow" when you have a large time frame. Another difficult part was working alone. Working alone is a challenge in its self because you have no to bounce ideas off. I found that in talking with my friends about my project I would hear my thoughts out loud and would realize that some of the problems I had with my program were easily avoidable.

I plan to continue development on this application to grow it to be a more powerful Gallery application. Although this application is not on the Google Play store currently, I would like to eventually make the application public, but in order to do this I would have to pay for the Cloud Vision services. In the future I plan to incorporate social media into the application. Social media is very popular for photo sharing and I believe that this application can benefit from it. I would also like to use the backend Java code to create a desktop application version for both Windows and OSX operating systems as Java is portable.

In the end this project was challenging and feel that I have grown my self as a software developer. The Android platform is something that I enjoy developing on. I have grown my Java development skills greatly during the course of this project which will help me in my career. Java is currently one of the most popular languages and as it is a high level language, it is very powerful. I look forward to continuing to grow as a developer as I make my way into the field.

7. Bibliography

Android Developer Documentation for App Development, https://developer.android.com/docs/

- Main reference used. Contains all information about Androids View, dependencies, libraries, Activities, Fragments, AsyncTasks and Threads. All of these were used within my project.

Cloud Vision API Client Libraries, https://cloud.google.com/vision/docs/libraries

- Information on authenticating a request to Cloud Vision and dependencies included in Gradle.

Cloud Vision API v1, https://developers.google.com/resources/api-libraries/documentation/vision/v1/java/latest/overview-tree.html

- Cloud Vision API v1packages, used to create the Annotated request for Cloud Vision label detection. This is a very extensive package with references to all classes needed to create aVision Request.

Google Play Services, https://developers.google.com/android/guides/overview

- Google Play Services used for Facial Detection within images.

Glide documentation, https://bumptech.github.io/glide/

- Glide library is used to load images into a View dynamically for faster performance and reducing memory foot print.

Java on Google Cloud Platform, https://cloud.google.com/java/docs/

- Google Cloud Vision documentation for Java. This helped me get a general understanding of how to make requests to Cloud Vision as well as learn about all of Vision's features.

Package Manager Utils, https://github.com/GoogleCloudPlatform/cloud-vision/blob/master/android/CloudVision/app/src/main/java/com/google/sample/cloudvision/PermissionUtils.java

- Class used to create signature need for Annotated request to Cloud Vision.