The University of Hong Kong

Faculty of Engineering

Department of Computer Science

COMP7705

Project Report

Lipstick Finder: A Mobile Application for Lipstick Recognition, Makeup, and Recommendation

Submitted in partial fulfillment of the requirements for the admission to the degree of Master of Science in Computer Science

By

FENG Yizhou 3035679628

JIANG Ling 3035030066

QIAO Shunjie 3035669233

ZHAO Zixuan 3035656315

Supervisor: Dr. T.W. Chim

Date of submission: 01/08/2020

Abstract

The demand for beauty guide applications is surging currently, especially for lipsticks. In this project we built a mobile application, Lipstick Finder, to recognize the lipstick from images, do lip digital makeup and recommend lipsticks to users.

The lipstick recognition module incorporates a novel lipstick RGB color calculation algorithm (RMBD) and a close-to-real-world lipstick color database. The quantitative evaluation result shows that our approach has a plausible recognition accuracy.

The lip makeup module overlays target lipstick color on the profile image in the HSV color space. The makeup algorithm is capable to synthesize convincing lip makeup images.

To recommend personalized lipsticks, the recommendation module conducts database filtering mainly for new users based on users' preferences towards lipsticks, and collaborative filtering for existing users based on users' interactions with lipsticks.

Lipstick Finder is constructed to support aforementioned modules using React Native framework, wishing to provide an intelligent lipstick assistant for everyone.

Keywords: React Native, lipstick, recognition, makeup, recommendation

Declaration

We (Jiang Ling, Zhao Zixuan, Qiao Shunjie, Feng Yizhou) declare that this project, **Lipstick Finder: A Mobile Application for Lipstick Recognition, Makeup, and Recommendation**, is our original work. We have not copied from any other's work or any other sources except where due reference or acknowledgement is made explicitly, nor has any part been authored by any other person.

Acknowledgements

We want to express our special thanks of gratitude to our supervisor Dr. T.W. Chim, not only for his constructive suggestions on our application development but also for his patient guidance and gentle encouragement. Without his help, we would not have been able to finish this project to our own satisfaction.

We are also thankful that Dr. Vivien Chan provides us with effective suggestions in the weekly presentation. Thank the University of Hong Kong for providing us with a comfortable learning environment, and thank the MSc (CS) faculty for still working hard in the emergency of the COVID-19.

Lastly, thank all team members for endeavoring themselves in the project. We share a good time together.

Table of Contents

1. Introduction

Lipstick has eternally occupied a prominent share in the beauty market. There are countless brands, colors with various liquidity, not to mention dazzling textures shining or matte. Due to overabundant advertisements and social media, choosing a suitable lipstick is even harder for ordinary people now, especially those who are not familiar with lipsticks. Many people refer to other's profile photos with pretty lip colors that they think might be suitable for them as well. However, the lipsticks used in profile photos are unknown to viewers. Life will be much easier if there is a tool to help recognize lipstick directly from the photo, and it would be more pleasant if users can try the lipstick color on their face.

Unfortunately, applications in the current market are unsatisfactory. First, none of them have both lipstick recognition and lip makeup functions. Second, problems exist in applications being able to deliver lip makeup service. Some of them only provide filters but not real lipstick products, such as Meitu and Qingyan. Some of them only offer the makeup service for lipsticks of one brand, which is an obvious problem in some cosmetic brands' official websites.

To meet users' requirements, we construct an application, Lipstick Finder, including both lipstick recognition and lip makeup functions. Meanwhile, to further realize our goal of providing an intelligent lipstick assistant for everyone, we incorporate the recommendation system in our application to give personalized lipstick recommendations.

Our report is organized as follows:

Section 1: Briefly introduce our project.

Section 2: We analyze users' pain points in current society and investigate current beauty applications in the market. Based on the analysis above, we define the scope of our project.

Section 3: We introduce the methods we used to implement the three main functions of our application: lipstick recognition, lip makeup and lipstick recommendation.

Section 4: The design and implementation details for each module of our application are illustrated in this section.

Section 5: This section evaluates our application from two aspects: module functionality and algorithm performance.

Section 6: We discuss the future work and potential improvements of our application.

Section 7: We summarize all work we have done and point out our greatest contribution.

2. Problem Analysis

2.1 Background

There is always a problem for many people: “What kind of lipstick does the character use in the movie?” Another problem usually comes subsequently: “Does the lipstick of this color suit me/someone?” (e.g. Figure 2-1)

Figure 2-1: People's desire for lipstick recognition products.

For the potential lipstick buyers, the only way to find the target lipstick using one image is to ask questions in the relevant forums. It may take a long time to get others' responses because probably no one knows the answer. Besides, even though someone gives an answer, they are not sure whether the lipstick recommended by others is the correct color.

For cosmetics firms, they promote their lipsticks in order to earn the profit. Advertising has always been a big part of their spending. They widely use TV advertising, online advertising and advertising boards to promote their new series. But it is hard to turn potential customers into actual buyers unless they have tried the lipstick and find that this color suits them well.

If an application can recognize the most possible lipstick for a user-uploaded image, give users digital trials and link them to purchase links, the gap between

potential lipstick customers and cosmetic firms will be filled. That’s a great commercial value.

2.2 Market analysis

We researched many products on the market but did not find any product that can provide a complete lipstick recognition function and put it into use on a large scale.

For lip makeup products, they can be roughly divided into two categories. One is photo editing applications such as Meitu and Qingyan, and the other one is the official websites of cosmetic brands that provide lip makeup functions.

Meitu is a popular photo editing application. As shown in Figure 2-2, users can do the makeup process directly on the person in the profile by choosing different color swatches at the bottom of the view. The color/light of the lips can be adjusted with a slider. What's more, Meitu can identify key points on the face, allowing users to manually adjust key points in detail. Same as Meitu, Qingyan is also a retouching application, which provides lip makeup service. The advantage of Qingyan is that it points out the specific color, so it is more convenient for users to find lipsticks of specific colors. However, the color name setting remains a common shortcoming for both products. Since their main function is to beautify photos, the lipstick color palette they provide does not match lipstick products in reality. These colors are actually a set of photo filters that are used to beautify the lips in the photo. They cannot additionally give export information of these lipstick colors, such as brand color number, etc. Although users already select a suitable color, they still need to search more

lipstick trail reports to check whether the on-lip color really suits them before purchasing.

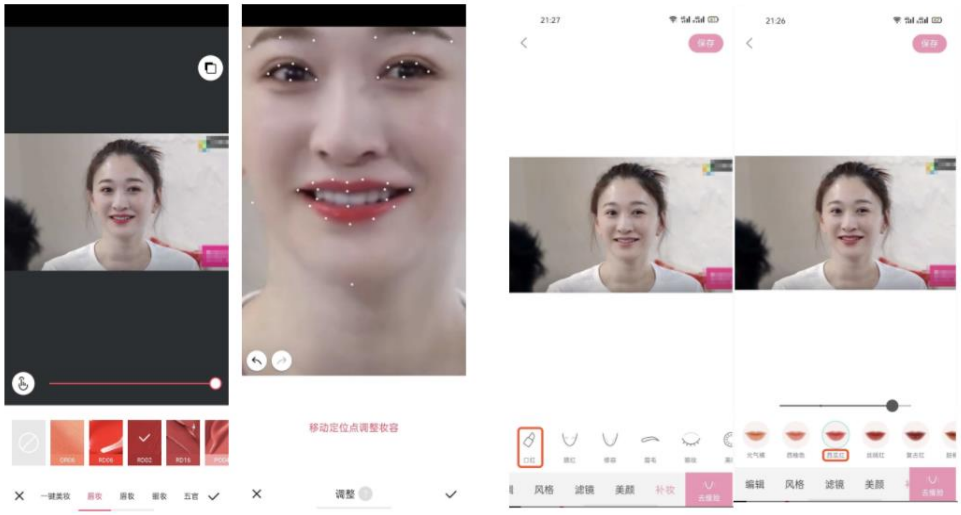

Figure 2-2: Left: Screenshots of Meitu. Right: Screenshot of Qingyan.

Some cosmetic brands' official websites provide lipstick try-on functions, such as Lancome shown as Figure 2-3. However this function is not used for lipsticks in the entire market, because the brand's database only supports its own products' digital makeup. In general, this kind of official websites only support PC access, and the mobile interface is not user-friendly.

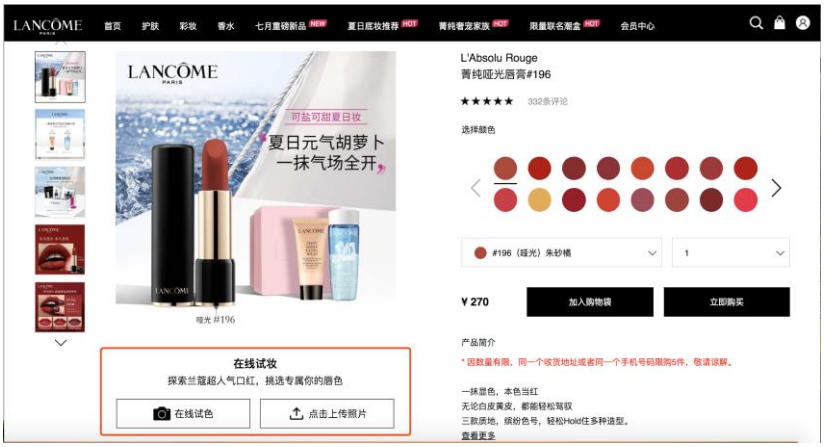

Figure 2-3: Website screenshots of Lancome

Some developers try to incorporate the lip makeup test function into the mobile application, but these products basically neither provide comprehensive lipstick products or a convenient makeup test service. Lipstick Swatches shown in Figure 2-4 is the most downloaded related product in the app store. But we have not been able to experience it because the makeup function is very unstable and keeps on crashing. According to the introduction of the application, if the user selects a lipstick and then gives a profile image by taking a picture through the camera or selecting a photo from the album, Lipstick Swatches will render the lips in the photo according to the lipstick selected by the user. However, there is only one lipstick brand displayed in this application, and there are only a few lipstick textures and color choices.

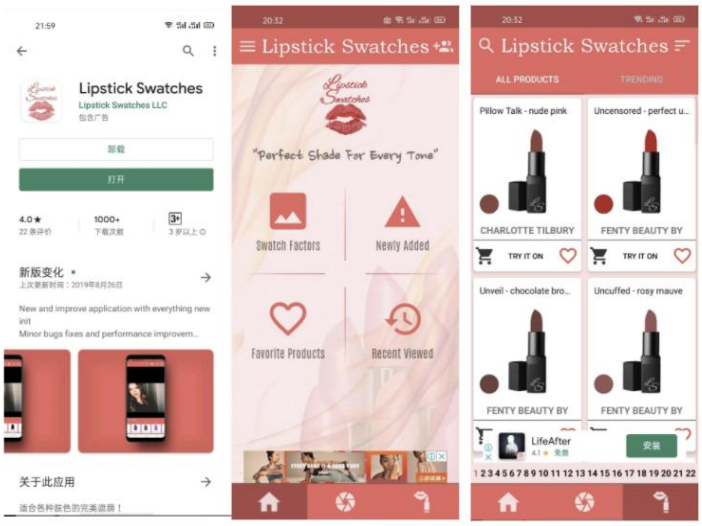

Figure 2-4: Screenshots of Lipstick Swatches

2.3 Project Scope

The core function of this project focuses on lipstick recognition. The target users are those who want to know what lipstick a person in the photo used. Such target users usually have a sequent question: “Does this lipstick suit me/someone?”

To have a complete pain point solving solution for the expected target users, we extend the project's function to include lip makeup and recommendation system.

This project is going to build a system in the form of a mobile application that can achieve the below functions:

- It can extract lipstick color from the user-uploaded photo and return the most similar lipstick brand and corresponding color number.
- Given a profile image and a lipstick color selected by the user, it can generate convincing lip makeup results.
- A lipstick recommendation system will recommend users the lipsticks they potentially like on a daily basis. The detail page of a lipstick contains purchasing links, which endows dual value to the Lipstick Finder.

3. Methodology

3.1 Recognition Algorithm

Given a human face image, we aim to recognize the lipstick color number of this human and give the most similar lipsticks in our database. We expect the recognition results to have similar color with the user-uploaded image. The recognition process falls into three parts: lips region detection, lips color detection and similar lipsticks inference.

3.1.1 Lips Region Detection

To achieve the lips region detection, we use the face-parsing model [1] to get the lips region of the human face. This model trained CelebAMask-HQ dataset [2] on BiSeNet [3], which can handle high-resolution human face features segmentation tasks at a real-time inference speed.

We take the 512×512 resolution image as the model input, the output will be a 512×512 matrix whose elements represent the human face feature code. Using upper lip and lower lip's feature codes, we can retrieve the corresponding lips region from the original input image as shown in Figure 3-1.

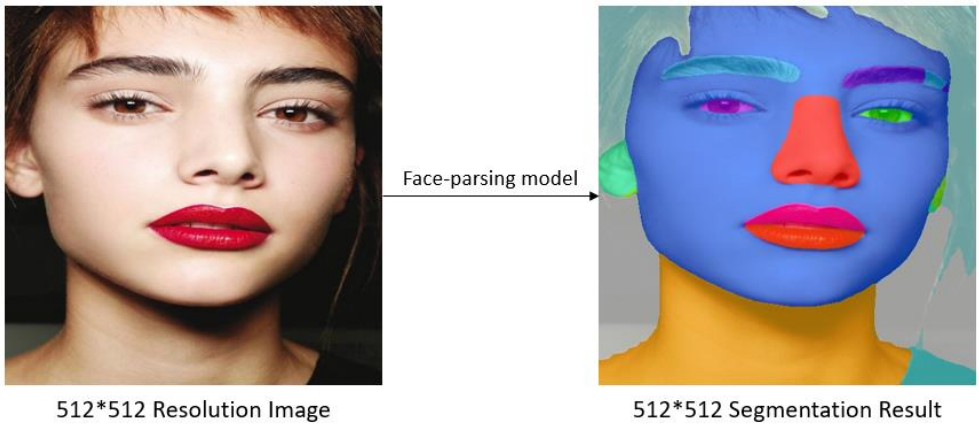

Figure 3-1: The visualization result of human face segmentation using face-parsing model, the upper lip and the lower lip are segmented

3.1.2 Lips Color Detection

We propose an over-bright and over-dark region removement with the average RGB color calculation algorithm (RMBD) to detect the lips region color. The RMBD removes the over-bright and over-dark points on the lips. This method largely avoids the abnormal color detect result caused by light imbalance. The details are elaborated in the nested two sections 3.1.2.1 and 3.1.2.2.

3.1.2.1 Remove Anomalous Illumination Region

Color space represents the color information of one image using three color components. Retrieving one image's RGB color space form is easy, but the RGB form of one image is not preferred in unrestricted illumination condition detection because it mixes the color and light intensity. To separate the over-bright region, over-dark region and normal light region, we convert the RGB color space of the user-uploaded image into the HSV color space. H, S and V represent an image's hue, saturation and value correspondingly. The color space conversion is processed using (1), (2) and (3).

$$H = \arccos \frac{\frac{1}{2}(2R-G-B)}{\sqrt{(R-G)^2-(R-B)(G-B)}} \quad (1)$$

$$S = \frac{\max(R,G,B)-\min(R,G,B)}{\max(R,G,B)} \quad (2)$$

$$V = \max(R,G,B) \quad (3)$$

Once all pixels are transformed into the HSV color space, we exploit a filter to filter out the over-bright points and over-dark points. Specifically, the filter takes input as a 512×512 matrix I_c whose elements are feature codes of the user-uploaded image. Feature code 12 represents the upper lip, feature code 13 represents the lower lip and feature code 1 represents the face. Corresponding to the 512×512×3 HSV color matrix I_{hsv}, the filter simply sets $I_c(i, j)$ to 1 if $I_{hsv}(i, j, v)$ is smaller than 5 or larger than 250. The mathematical expression of the filter result is shown in (4) and the sample visualization result is shown in Figure 3-2.

$$I_c(i,j) = \begin{cases} 1, & \text{if } I_{hsv}(i,j,v) \in [0,5) \cup (250,255] \\ I_c(i,j), & \text{others} \end{cases} \quad (4)$$

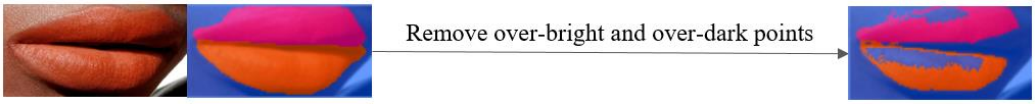

Figure 3-2: The visualization result after removing over-bright and over-dark points of upper and lower lips

3.1.2.2 Lips Average RGB Values Calculation

Using the over-bright and over-dark region removement method we propose in 3.1.2.1, a filtered 512×512 feature code matrix I_{fc} is generated. We therefore calculate the average R, G and B values of the filtered lips region using I_{fc} as shown in Figure 3-3. More specifically, the image in 512×512×3 RGB color space is represented by I_{rgb}, those pixels whose corresponding feature code $I_{fc}(i, j)$ equals 12 or 13 will be used to calculate R_{avg}, G_{avg} and B_{avg} using (5).

$$X_{avg} = \overline{I_{rgb}}_{(i,j)\in I_{fc}: I_{fc}(i,j)\in(12,13)}(i,j,X), X \in (R,G,B) \qquad (5)$$

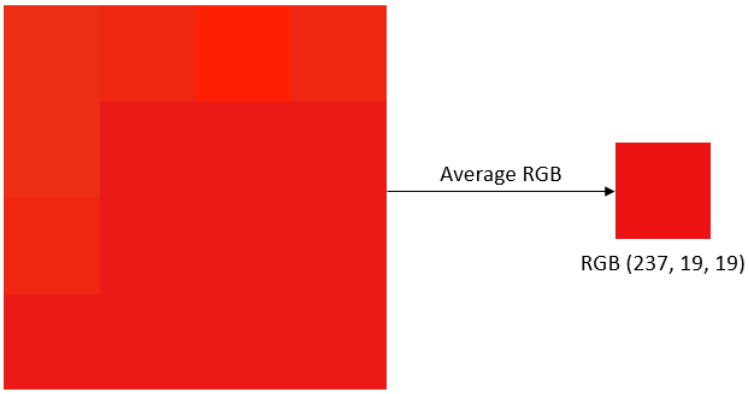

Figure 3-3: Sample average RGB values calculation

3.1.3 Similar Lipsticks Inference

The top-3 similar lipsticks will be given as recognition results after getting lips region average RGB values. The weighted Euclidean color distance [4] between the average RGB values and one lipstick color in our database is chosen as the

similarity sorting criterion. The color distance calculation formula between RGB color C_1 and RGB color C_2 is given in (6).

$$\bar{r} = \frac{C_{1,R}+C_{2,R}}{2}$$

$$\Delta R = C_{1,R} + C_{2,R}$$

$$\Delta G = C_{1,G} + C_{2,G}$$

$$\Delta B = C_{1,B} + C_{2,B}$$

$$\Delta C = \sqrt{\left(2 + \frac{\bar{r}}{256}\right) \times \Delta R^2 + 4 \times \Delta G^2 + (2 + \frac{255-\bar{r}}{256}) \times \Delta B^2} \qquad (6)$$

Once all color distances are calculated, we sort the distances in ascending order and return the first three lipsticks as recognition results.

3.2 Makeup Algorithm

The input of our makeup algorithm contains a 512×512 resolution user-uploaded image and a target lipstick's color. Our objective is to generate the lip makeup image using the target color. The process consists of two parts: lips region detection and lip makeup. The lips region detection algorithm is introduced in section 3.1.1. However, how to overlay the target color on the lips region remains a problem. A typical strategy is to overlay color in the HSV color space [5]. The details are introduced in the following subsections.

3.2.1 HSV Color Space Conversion

The process of how we convert an image from the RGB color space into the HSV color space is elaborated in section 3.1.2.1. Apart from converting the user-uploaded image, we also transform the target lipstick's color into the HSV color space, which is represented by C_{hsv}.

3.2.2 Color Overlay

The 512×512×3 color overlay image in the HSV color space I'_{hsv} is generated using (7). Figure 3-4 shows the process of how I_{hsv}'s hue channel and saturation channel are replaced by C_{hsv}'s corresponding channels.

$$I'_{hsv}(i,j) = [C_h, C_s, I_v(i,j)] \qquad (7)$$

Figure 3-4: The visualization result of color overlaid image

3.2.3 Nonlip Region Compositing

Using inverse transformation from the HSV color space to the RGB color space, we can get I'_{rgb}. Note that the lips region can be detected using the method introduced in section 3.1.1. The lips region is represented as a foreground mask M_f, in which $M_f(i, j)$ is 1 if it includes in the lips region and 0 elsewhere. Thus, together with the original image I_o, the final target makeup image I_t can be composed as (8) where **1** is an image of all ones and · refers to element-wise product operator. The lip replacing process is shown in Figure 3-5.

$$I_t = M_f \cdot I'_{rgb} + (\mathbf{1} - M_f) \cdot I_o \qquad (8)$$

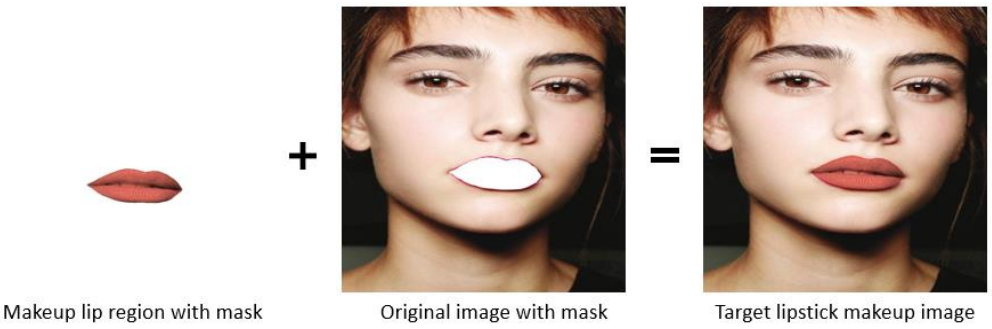

Figure 3-5: The visualization result of target lip makeup image

3.3 Recommendation Algorithm

We design our recommendation system based on the consideration of two conditions. First, given the basic features of lipsticks that new users like, we expect the system to offer a certain number of lipsticks with higher possibility that they would be interested in. Meanwhile, existing users' behavior may reflect their preferences of certain kinds of lipsticks, which allows the system to give more personalized recommendations to existing users of our app. Due to these considerations, we constructed two kinds of recommending strategies: one is database filtering mainly designed for new users who just register or seldom use Lipstick Finder, and the other one is collaborative filtering mainly serving existing users who use Lipstick Finder frequently.

3.3.1 Database Filtering

The Lipstick Finder requires new users to answer questions about their preferences of some basic features of lipsticks, such as the kind, the texture and the color during the process of registration in our app. These features are consistent with what we used to label our *lipsticks* collection in MongoDB except the color. In addition, we collected the ranking of some popular lipsticks

from a famous social media Xiaohongshu. We also stored this information as a collection, *topLipsticks*, in our database.

We first execute the range query operation as shown in Figure 3-6 on *topLipsticks* collection based on the features that the specific user likes except the feature, color.

```
#Get the lipsticks from Toplipsticks meet the requirement of kind and texture
filteredresult = client.db.topLipsticks.find({'liquid':{'$in':liquid},'texture':{'$in':texture}},
                                             {'_id': 0, 'rank': 1,'brand': 1, 'series': 1,
                                              'liquid': 1, 'texture': 1, 'color': 1,
                                              'price': 1, 'name': 1, 'lipstick_id': 1})
```

Figure 3-6: Range Query in topLipsticks Collection using "liquid" and "texture"

For the lipsticks achieved from the previous operation, because users' color preferences are recorded as ambiguous labels (e.g., "Orange", "Pink", etc.), while the values of color in the *topLipsticks* collection are RGB values, we can't execute the range query on this feature.

To solve this problem, we first define one specific color value for each color scheme used in users' preference in Table 3-1:

Color Scheme	Default Hex	Color Swatch
Red	#FF0000	
Pink	#FFC0CB	
Orange	#FF7300	
Brown	#A16B47	
Purple	#7400A1	

Table 3-1: Color scheme definition

For each color scheme liked by the user, we sort the RGB distance between the remaining lipsticks and this color scheme and return a list of closest lipsticks.

The formula used to calculate the RGB distance between two RGB values is the same as the formula (6) that has been illustrated in section 3.1.3.

After doing this for each color scheme, we aggregate the results and sort them by the “ranking” feature in ascending order. Finally, we collect the required number of lipsticks from the top of the ranking list and recommend them to new users.

3.3.2 Collaborative Filtering

The interactions between an existing user and the Lipstick Finder including “liking a lipstick”, “browsing a lipstick”, etc., are recorded in the database. Based on these interactions, the rating of lipsticks for each user can be calculated and stored. The rating information will be used to execute collaborative filtering in the following steps, which are adapted from the “Recommendation Systems: User-based Collaborative Filtering using N Nearest Neighbors” [6], to give more personalized recommendation for these users:

First, we normalize the rating by the mean rating value of lipsticks each user gives since different users may have different standards to rate the lipsticks and we don’t want this subjective matter affecting the result of calculating the similarity between users. As shown in Figure 3-7, `adj_rating` is achieved through original value minus mean value.

	userID	lipstickID	rating_x	rating_y	adj_rating
0	0	00002	10	5.75	4.25
1	0	00000	10	5.75	4.25
2	0	00334	10	5.75	4.25
3	0	00222	10	5.75	4.25
4	0	00893	10	5.75	4.25

rating_x : original value
rating_y : mean value
adj_rating : adjusted value

Figure 3-7: Samples of Adjusted Rating Table

Using the adjusted rating table, we can construct the users-and-lipsticks matrix. The row is `userID` and the column is `lipstickID`. Figure 3-8 shows the interpolation of missing values is done with the average of the adjusted rating value of each lipstick.

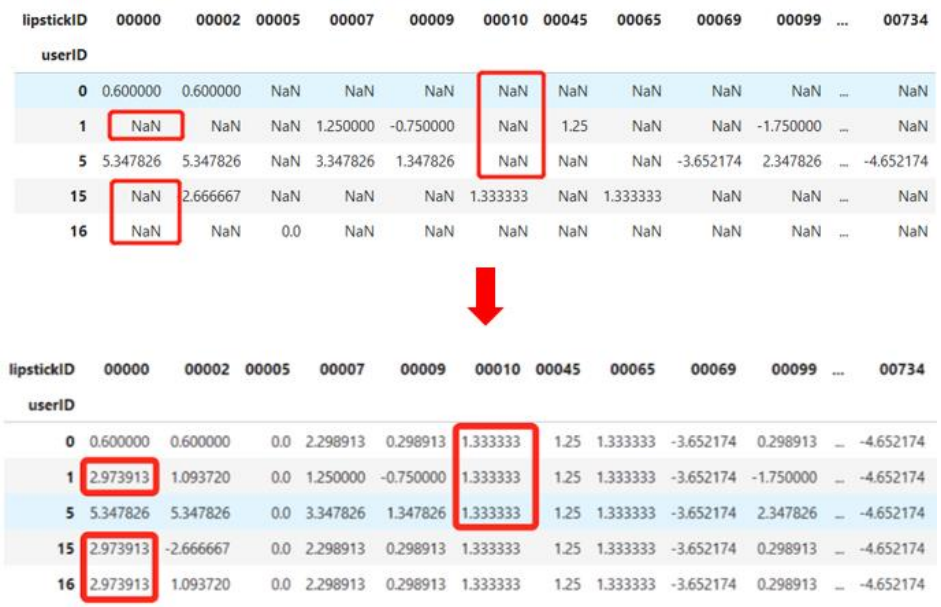

Figure 3-8: Example of filling "NaN" with average rating of lipsticks

With this matrix, we can calculate the similarity between users according to the formula (9) of cosine similarity. U_i represents the vector for user i and $r_{i,k}$ represents the adjusted rating of lipstick k given by user i. As is shown in Figure 3-9, value of similarity closer to 1 means these two users are more similar to each other.

$$sim(U_i, U_j) = \cos(U_i, U_j) = \frac{U_i \cdot U_j}{\|U_i\| \, \|U_j\|} = \frac{\sum_k r_{i,k} r_{j,k}}{\sqrt{\sum_k r_{i,k}^2} \sqrt{\sum_k r_{j,k}^2}} \quad (9)$$

userID	0	1	5	15	16
userID					
0	0.000000	0.942450	0.818504	0.937185	0.952424
1	0.942450	0.000000	0.918228	0.969656	0.990451
5	0.818504	0.918228	0.000000	0.883732	0.942582
15	0.937185	0.969656	0.883732	0.000000	0.979127
16	0.952424	0.990451	0.942582	0.979127	0.000000

Figure 3-9 Samples of Users Similarity Matrix

Next, for each user, we will select a few users who are the most similar ones to it. We generate a list of lipsticks which haven't been rated by this user, but have been rated by similar users. For these lipsticks, we use formula (10) to predict the rating this user may give based on the rating given by similar users. The result is shown as Figure 3-10. For example, for user 0 who hasn't rated the lipstick 00099, by referring to the rating given by similar users, user 0 is predicted to give 8.6798 to this lipstick.

$$r_{u,i} = \bar{r}_u + \frac{\sum_{v \in N(u)} sim(u,v)(r_{v,i} - \bar{r}_v)}{\sum_{v \in N(u)} sim(u,v)} \quad (10)$$

```
userID: 0
  lipstickID      score
0      00099   8.679849
1      00045  10.650000
2      00009   9.177217
```

Figure 3-10. One example of collaborative recommendation result for user 0.

Finally, sorting the result by the calculated rating and storing results into the database for next day recommendation.

4. System Design and Implementation

The Architecture of our application “Lipstick Finder” is shown as Figure 4-1.

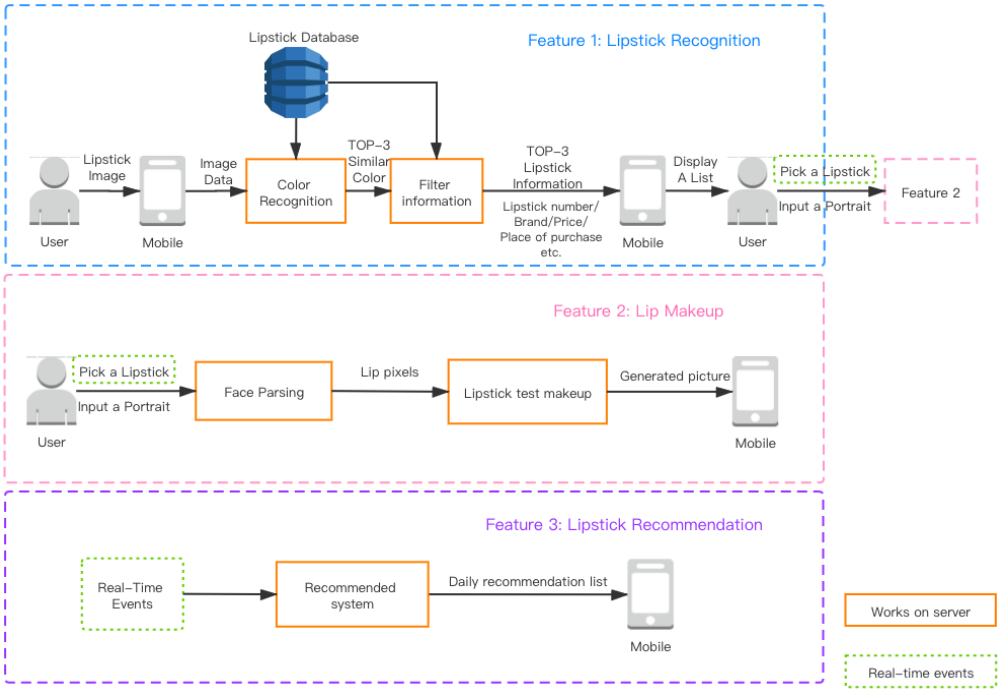

Figure 4-1: System architecture

We propose three main features to achieve our project’s objectives:

- Lipstick Recognition: after getting a profile image from the user, the backend will do face parsing and extract the top 3 possible lipsticks, then pass it to the user.
- Lip Makeup: users can provide a profile image and choose a lipstick (or the lipstick recognized previously) to do lip makeup and see whether that lipstick suits them.
- Lipstick Recommendation: it will recommend a list of lipsticks everyday based on users’ behavior.

In this section, we introduce every component of our system, including tools being used, database, design and implementation in both frontend and backend.

4.1 Tools

4.1.1 React Native

React Native is a development tool enabling developers to build cross-platform mobile applications using Javascript. It provides a core set of platform agnostic native components like View, Text, and Image that map directly to the platform's native UI building blocks. These components wrap existing native code and interact with native APIs via React's declarative UI paradigm and JavaScript. This feature enables teams to use it in existing Android or IOS applications, or just start a new one from scratch without lowering users' experience [7].

4.1.2 Flask

Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications [8]. Flask is part of the categories of the micro-framework, so it has little to no dependencies to external libraries [9]. We use Flask as our backend server framework to handle HTTP requests.

4.1.3 PyTorch

PyTorch is a scientific computing library based on Python, providing people with a flexible and fast deep learning research platform [10]. PyTorch is developed by Facebook's artificial intelligence group. It supports GPU acceleration and implements dynamic neural networks which is not supported by mainstream frameworks such as Caffe and TensorFlow. In this project, we

used a pre-trained face segmentation model that was trained and tested on PyTorch to find the lip of profile images.

4.1.4 MongoDB

MongoDB is a NoSQL database which stores data in documents without a predefined schema. Compared to relational databases (e.g., MySQL, PostgreSQL, etc.), the scalability and flexibility of MongoDB can help us quickly build a database with multiple collections to support our backend server without frequently changing schemas as development goes on.

4.2 Database

4.2.1 Lipstick Dataset

For lipsticks data, we started with Olivia's lipsticks dataset [11], but this dataset is outdated and too small (Figure 4-2 shows it contains only 287 lipsticks).

Figure 4-2: Left: Olivia's lipsticks dataset in JSON format. Right: total brands and number of lipsticks in this dataset.

The limitation of the old dataset also created unnecessary difficulties for later evaluation of our recognition algorithm, because we were not able to generate a

suitable test set. Therefore, we decided to build our own lipstick dataset with python scraping the latest lipsticks data from official cosmetic websites. Although using a program is more efficient than manually collecting data, it was still time-consuming. In addition to color, we also collected other features: texture, liquidity and price.

After cleaning the dataset and dropping duplications, we collected 1550 lipsticks from 13 brands in total as shown in Figure 4-3, which cover 8 common textures in Figure 4-4 and 2 liquidity types in Figure 4-5.

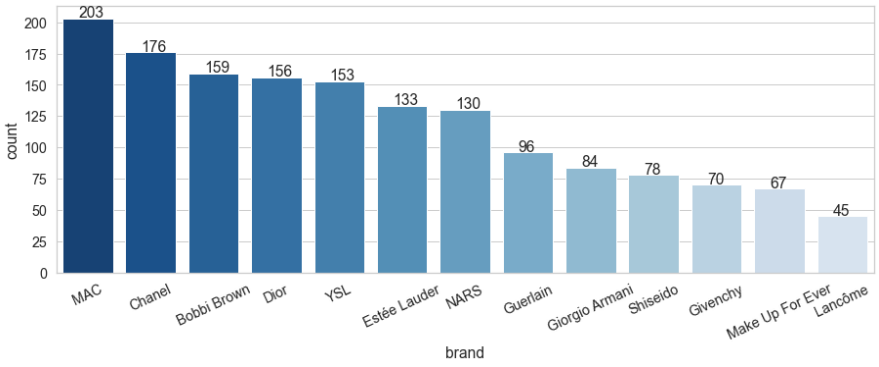

Figure 4-3: Brand distribution of current lipstick dataset

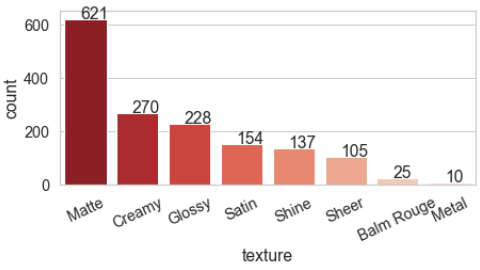

Figure 4-4: Texture distribution of current lipstick dataset

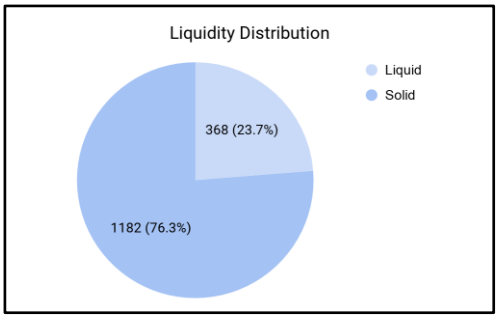

Figure 4-5: Liquidity distribution of current lipstick dataset

We do not directly use the extracted color as our lipsticks' color. Instead of color swatches crawled from the official website, we replace them with our recognition algorithm's recognized results. We hypothesize that the official

color swatches do not always show the real color of lipstick on the lip. The real on-lip color is actually an encoded result. We suppose that the color swatch's color has one to one mapping relationship with its encoded on-lip color. So we replace lipstick colors using our recognition algorithm's result to create a real on-lip color dataset. The process of real on-lip color recognition is shown like Figure 4-6.

Figure 4-6: The process of real on-lip color recognition: First, get the standard lipstick's trial image; Second, do color recognition using our recognition algorithm; Third, store it in the dataset.

4.2.2 Database

4.2.2.1 User

ID	email	password	name	gender	color	kind	texture	path
5	zzx@163.com	04ef1a034647ef32	Thanks	male	[Pink]	[Lip glaze]	[Matte]	/home/ubuntu/web/profiles/5.jpeg
8	QSJ@gmail.com	c401fb758089944da3ce	Q	others	[Red, Brown]	[Lip glaze]	[Glossy]	/home/ubuntu/web/profiles/8.jpeg

Table 4-1: Users collection in MongoDB

Table 4-1 is the *Users* collection in our database. The collection contains user information with these features: user ID, email, password, name, gender, lipstick color system the user preferred, lip makeup products the user preferred,

lip makeup products' texture the user preferred, and the address where the user avatar is stored on the server. The `email` and `password` are collected from the register module. The `gender`, `color`, `kind` and `texture` are collected from the questionnaire module. The `name`, `gender`, and `path` can be modified in the account module. The usage of `ID` makes it convenient to find the personal information of a specific user. What's more, `ID` is the unique identifier of the user, it has a linkage effect with the information in the liked lipsticks and personal lipstick rating collections.

4.2.2.2 Lipstick

brand	series	series_en	name	color	liquid	texture	price	brand_id	series_id	texture_id	lipstick_id
Bobbi Brown 芭比波朗	丰润莹彩唇釉	Rich Color Gloss	4号 Dusty Rose 玫瑰紫	#A76369	TRUE	Glossy	280	0	9	2	0
Bobbi Brown 芭比波朗	丰润莹彩唇釉	Rich Color Gloss	9号 Pink Sorbet 粉水晶	#C85F5D	TRUE	Glossy	280	0	9	2	1

Table 4-2: lipsticks collection in MongoDB

Table 4-2 is the *lipsticks* collection in our database. This collection contains 1550 lipsticks with these features: brand, series, lipstick name, color, liquidity, texture, price, etc. The usage of `<feature>_id` makes the query faster and more efficient. For example, given `brand_id` and `series_id`, we can retrieve a list of lipsticks from our database with a query similar to Figure 4-7.

```
pd.DataFrame(db.lipsticks.find({'brand_id': 6, 'series_id': 5}, {'_id': 0, 'link': 0}))
```

	brand	series	series_en	name	liquid	texture	color	price	brand_id	series_id	texture_id	lipstick_id
0	Guerlain 娇兰	丝绒唇釉	None	M06	True	Matte	#9C4E4A	290	6	5	3	00778
1	Guerlain 娇兰	丝绒唇釉	None	M25	True	Matte	#BA182F	290	6	5	3	00779

Figure 4-7: Querying database with specific brand and series. This query is used in the lip makeup screen in Lipstick Finder.

4.2.2.3 Top Lipsticks

```
pd.DataFrame(list(db.topLipsticks.find({},{'_id': 0, 'link': 0})))
```

	rank	brand	series	series_en	name	liquid	texture	color	price	brand_id	series_id	texture_id	lipstick_id
0	1	Estée Lauder 雅诗兰黛	倾慕哑光唇膏丝绒系列	Envy Velvet	333 Persuasive 干枫叶色	False	Matte	#BD4B43	270	3	15	3	00493
1	2	Guerlain 娇兰	亲亲唇膏	None	330	False	Sheer	#B33332	320	6	11	6	00815

Table 4-3: topLipsticks collection in MongoDB

Table 4-3 is the *topLipsticks* collection in our database. The collection contains information about the 30 most popular lipsticks and lip glazes with ranks that we have collected from beauty platforms and apps. These features (brand, series name, lipstick name, color, liquidity, texture and price) are consistent with the *lipsticks* collection, especially `<feature>_id`.

4.2.2.4 Liked lipsticks

userID	lipstickID
1	[00002, 00000, 00334, 00222, 00893]
2	[00007, 00009, 00099, 00045]

Table 4-4: likes collection in MongoDB

Table 4-4 is the *likes* (users' liked lipsticks) collection in our database. Each document of this collection consists of the specific user ID `(userID)` and the list of his/her favorite lipsticks `(lipstickID)`. The values of `userID` are consistent with the `ID` in the *Users* collection, and the values of `lipstickID` are consistent with the `lipstick_id` in the *lipsticks* collection.

4.2.2.5 Personal lipstick rating

	userID	lipstickID	rating
0	0	00000	10
1	0	00002	10

Table 4-5: rating collection in MongoDB

Table 4-5 is *rating* (users' rating of lipsticks) collection in our database. Each document contains a user's identification, a lipstick's identification and a rating. The `userID` and `lipstickID` are consistent with what we used in the other collections of the database. The `rating` feature is achieved according to the users' behavior in our app, such as browsing lipsticks and liking lipsticks. This collection is used in collaborative recommendation in our system.

4.2.2.6 Collaborative Filtering Recommendation

	userID	lipsticksID
0	0	[00007, 00045, 00009, 00005, 00685]
1	1	[00010, 00065, 00005, 00685, 01530]
2	5	[00045, 00005, 00685, 01530]
3	15	[00007, 00045, 00009, 00005, 00685]
4	16	[00007, 00065, 00010, 00045, 00685]
5	8	[00007, 00065, 00010, 00045, 00005]

Table 4-6: cfRecommendation collection in MongoDB

Table 4-6 is the *cfRecommendation* collection storing the collaborative filtering result in our database. Each row contains a user's identification and an array of lipsticks' identification. The `lipsticksID` is the collaborative recommendation results generated by the algorithm for a specific user. The `userID` and `lipstickID` used in this collection is consistent with that of other collections in our database.

4.3 Application Modules

All modules and functions of Lipstick Finder are shown in Table 4-7.

Modules	Functions
Registration	New user register
Access Control	Log in
	Log out
Lipstick Recommendation	Daily lipstick recommendation
Lipstick Detail Information	Display lipstick details and purchasing links
Lipstick Recognition	Upload image to detect lipstick
	Provide photo editing tools
Lipstick Makeup	Select lipstick from drop-down lists
	Upload image with selected lipstick to do lip makeup
	Save image to local album
Account	Display user's lipstick like list
	Edit name, gender or profile image
	Reset password
	Display version number and our project team

Table 4-7: Modules overview

In this section, both functionality and implementation of the frontend and backend of all Lipstick Finder components will be introduced.

4.3.1 Registration

4.3.1.1 Interface Introduction

The account creation is mandatory before using the Lipstick Finder. To register, users must follow below steps:

1. Open the Lipstick Finder app, there is a “sign up now” link below the “Login” button as is shown in Figure 4-8. Click it to enter the screen of registration, where users must input the email address, password and confirm-password to continue the registration process. The email must be in valid email format and the password must not be less than 8 characters. The server will check whether the email has been used before. Click the “SIGN UP” button, a questionnaire will be popped out.

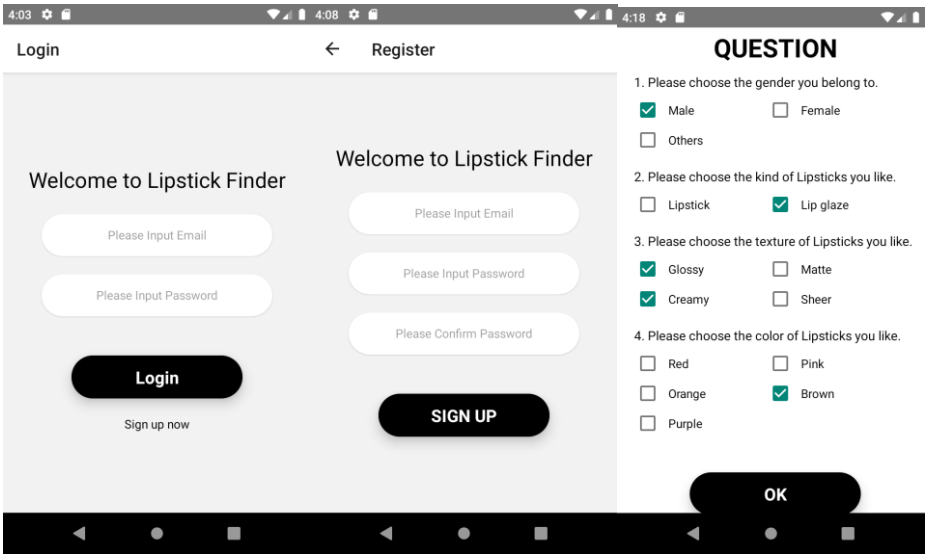

Figure 4-8: Navigate from Login Screen, to Register and Questionnaire Screen

2. All questions on the questionnaire are mandatory, because the answers will play an important role during the cold boot of recommendation for

this user. Select the answers appealing to true conditions and submit them by pressing the “OK” button.

4.3.1.2 Frontend Code

The main components we used to enable interaction between users and our app in Register Screen are `TextInput`, `TouchableOpacity` and `Alert`. We use three `TextInput` components to let users enter the email, password and confirm-password information. `TouchableOpacity` is used to render the button and `Alert` is used to pop up a reminder.

Before posting the “signup” request to our server, the password will be encrypted with SHA256 for security. The function shown in Code 4-1 is implemented in ./LipstickFinder/res/utils/security.js.

```
import { sha256, sha224 } from 'js-sha256';

let aes = require('aes-js');

export const encrypt= (key, text) => {

    let aes256key: ?Array<number>;

    let hash = sha256([key, "LipstickFinder"].join(''));
    aes256key = Array.from(
            Array.from(hash)
            .map((c, i) => (i % 2 ? c : null))
            .filter(Boolean)
            .map(c => c.charCodeAt(0))
        );

    const textBytes = aes.utils.utf8.toBytes(text);
    const aesCtr = new aes.ModeOfOperation.ctr(aes256key);
    const encryptedBytes = aesCtr.encrypt(textBytes);
    const encryptedHex = aes.utils.hex.fromBytes(encryptedBytes);
    return encryptedHex;
};
```

Code 4-1: Encryption of password

Registration will send a POST request using the Fetch API. If the server returns the “`True`” message in response, it means that the account has been

successfully stored into the database and we will navigate the screen to "Questionnaire", otherwise the user will be reminded with a corresponding error message. The fetch request code is shown as follows:

```
fetch('http://124.156.143.125:5000/signUp', {
  method: 'POST',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    email: email,
    password: uploadPassword,
  }),
})
  .then((response) => response.json())
  .then((responseJson) => {
    if (responseJson.result == 'True') {
      storeData('email', email);
      storeData('password', uploadPassword);
      storeData('uid', responseJson.uid);
      this.props.navigation.navigate('Questionnaire', {
        email: email,
        password: password,
      });
    } else if (responseJson.result == 'False') {
      LayoutAnimation.easeInEaseOut();
      Alert.alert('Account Already Exists');
    }
  })
  .catch((error) => {
    console.error(error);
  });
```

Code 4-2: Network request of "Register"

The main components we used to render the questions and choices in Questionnaire Screen are `Text`, `FlatList` and `CheckBox`. For each question, `Text` is used to show the question, and `FlatList` is used to pack a list of `CheckBox` components, each corresponding to one choice.

The POST request of the questionnaire is similar to Code 4-2. If the server responds with "success", the account has been successfully updated and the user will be navigated to "Login", otherwise the user will be warned with an error message and remains in this screen.

4.3.1.3 Backend Code

Upon receiving the request for signing up, the system first checks whether the email exists in our database or not. If not, the system will store the user's email and encrypted password into the database, generate an user ID for this new user and return “True”, otherwise return “False” in the response.

Upon receiving the request for upload answers of the questionnaire, we need to check the validity of the account first, then store the user's answers if it's valid.

4.3.2 Access Control

4.3.2.1 Interface Introduction

When users open our app, they will see the “Login Screen” as Figure 4-9. Type in correct email and password to sign in by clicking the “Login” button.

The function of “Logout” is hidden in the left side of the Account Screen. Draw the left side of the “Account Screen” and click the “Logout” to log out the app.

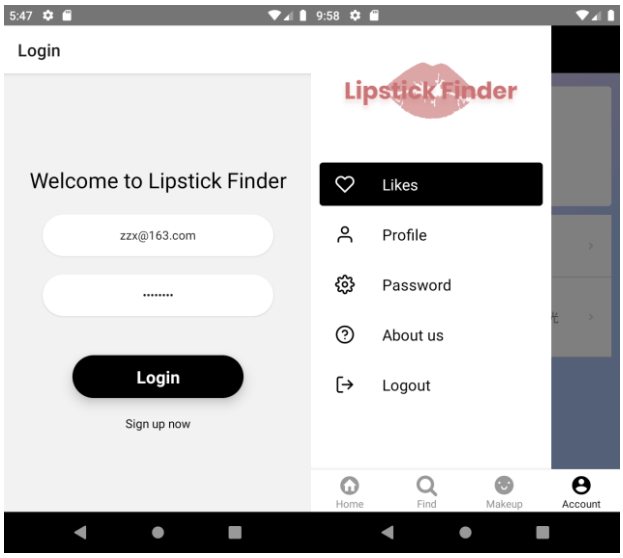

Figure 4-9: Screens rendering the function of Access Control

4.3.2.2 Frontend Code

The “Login Screen” is constructed mainly by `Text`, `TextInput`, `TouchOpacity` and `Alert` components in React Native. The `Text` is used to show the title. The `TextInput` is offered to users to type into their account information. The `TouchOpacity` is used to render the “Login” and “Sign up Now”. The `Alert` is used to pop up reminders to users.

After checking the format validity of input account information, the password will be encrypted directly (Code 4-3), which is handled similar to the registration process (Code 4-1). With the email and password, the front end will send a POST request to the server using Fetch API. If the response shows that the inputted information is valid, then the front end will navigate the screen to screens of functions, otherwise it will remind the user about errors and stay in this screen.

```
let uploadPassword = encrypt(email, password);
```

Code 4-3: Encryption of Password

The “Logout” is just a custom drawer item in the drawer navigation. Upon being clicked, it just set the stack of navigation to zero and route the screen to “Login”.

4.3.2.3 Backend Code

Upon receiving the “Login” request, the backend will check whether the email exists in the database first. If the query shows that the email exists in the database, the backend will check the validity of the password. If the password is also correct, the backend will set the value in `result` field of the response to “True”. If the email and password fail any one of the previous tests, the

backend will set the value in the `result` field of the response to “False”. Codes dealing with the function of “Login” are shown in Code 4-4.

```
def login(client, email, password):
    myquery = {"email":email}
    reply = client.db.Users.find(myquery)
    if (reply.count() == 1) and (password == reply[0]["password"]):
        return reply[0]["ID"], True
    else:
        print("login here")
        return "", False
```

Code 4-4: Function dealing with “Login”

4.3.3 Lipsticks Recommendation

4.3.3.1 Interface Introduction

Based on users' behavior, Lipstick Finder recommends a set of lipsticks for each user every day in “Home Screen”, shown as Figure 4-10.

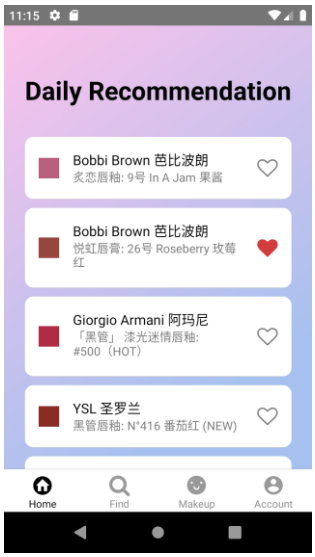

Figure 4-10: Screenshot of Home Screen

The “Home Screen” displays several lipsticks in a list on a scrollable page. Each `ListItem` represents a particular lipstick with its brand, series and name in text. The color of this lipstick will be presented in a square on the left side of

the `ListItem`. The state of the heart on the right side of the `ListItem` indicates whether the user marks this lipstick as a favorite lipstick: the red heart is marked as like, and the white heart is not marked as like, which can be shifted by short process on it. Long press this `ListItem` will jump to another interface, Lipstick detail information in 4.3.4.

4.3.3.2 Frontend Code

The whole component renders the `ListItem` to show the lipstick recommendation list in a `ScrollView` as shown in Code 4-5. In order to achieve the effects mentioned above, we use the state mechanism to monitor the lipstick information, then fill the lipstick information in the state into the `ListItem`.

```
{this.state.lipstickInfos.map((l, index) => (
  <ListItem
    containerStyle={{borderRadius: 10}}
    underlayColor='transparent'
    style={styles.row}
    key={index}
    leftIcon={{name: 'square-full', type: 'font-awesome-5', color: l.color}}
    rightIcon={{name: l.like ? 'heart' : 'heart-o', type: 'font-awesome', color: l.like ? COLORS.HEART : 'grey'}}
    title={l.brand}
    subtitle={l.series + ': ' + l.name}
    bottomDivider
    onPress={() => this.updateLike(index)}
    onLongPress={() => this.lipstickInfoPress(index)}
  />
))}
```

Code 4-5: Code of lipstick recommendation list

The lipstick information (such as name, series) is obtained through the Fetch API using the POST method to make a network request to the server. For security reasons, the password is encrypted on the frontend before sending to the server for authentication. This request will be triggered in `componentDidMount()` when the interface is mounted.

Moreover, in order to ensure the real-time performance, the heart label of lipstick needs to be refreshed. In Code 4-6, a `navigation.`

`addListener()` is added in `componentDidMount()` to call `refresh()` function with an HTTP GET request to retrieve the latest "like", so that every time the user focuses on this interface, everything is fresh.

```
componentDidMount = async() =>{

  const userID = await getData("uid");
  const password = await getData("password");

  fetch('http://124.156.143.125:5000/getRecommendLipstickInfo', {
      method: 'POST',
      headers: {
          'Accept': 'application/json',
          'Content-Type': 'application/json',
      },
      body: JSON.stringify({
          userID: userID,
          password: password,
      })
  }).then(response => response.json()).then(responseJson => {
      this.setState({lipstickInfos: responseJson.recommendLipstickInfoVos});
  }).catch((error) => {
      console.error(error);
  });

  this.props.navigation.addListener('focus', () => {
      this.refresh();
      // alert('Screen was focused')
  })
};
```

Code 4-6: Code of componentDidMount()

4.3.3.3 Backend Code

When the server receives the request to get the recommendation of lipsticks, it will first check the validity of the account information and call the `getRecommendation()` in the lipstickRecommendation.py to get the recommendation results (Code 4-7).

```
def getRecommendation(client, userID, designated_num=5):
    cfResult = getRecommendationFromCfRecommendation(client, userID)
    cfcount = len(cfResult)
    print(cfcount)

    if cfcount == designated_num:
        return cfResult

    else:
        result = cfResult + getRecommendationFromTopLipsticks(client, userID, designated_num-cfcount, cfResult)
        return result
```

Code 4-7: Rendering Recommendation List from results of Collaborative Filtering and Database Filtering

The function `getRecommendationFromCfRecommednation()` read the recommendation result from the *cfRecommendation* collection, which is generated by the process of collaborative filtering every day.

The collaborative filtering algorithm in the userBasedCF.py is executed on the server every day. It first reads the data from the *rating* collection in MongoDB and then constructs the DataFrame using Pandas (Code 4-8).

```
#Read Data from MongoDB
data = db.rating.find({},{'_id':0, "userID":1,"lipstickID":1,"rating":1})
data = list(data)
df = pd.DataFrame(data)
```

Code 4-8: Read the rating information from MongoDB

Next, the rating will be adjusted by original value minus mean value. Two rating matrices - `check` and `final` - are constructed using the original and adjusted value. `check` will be used to calculate the predicted rating and `final` will be used to calculate user similarity (Code 4-9).

```
#Adjust the rating by mean value
Mean = df.groupby(by="userID",as_index=False)['rating'].mean()
Rating_avg = pd.merge(df,Mean,on='userID')
Rating_avg['adj_rating']=Rating_avg['rating_x']-Rating_avg['rating_y']

#Construct the rating matrix
check = pd.pivot_table(Rating_avg,values='rating_x',index='userID',columns='lipstickID')
final = pd.pivot_table(Rating_avg,values='adj_rating',index='userID',columns='lipstickID')
```

Code 4-9: Adjust the rating value and construct the rating matrix

The missing values NaN in the final matrix are interpolated with the mean value of lipsticks. After interpolation, the similarity between users can be calculated by using the formula of cosine similarity (Code 4-10).

```
final_lipstick = final.fillna(final.mean(axis=0))

cosine = cosine_similarity(final_lipstick)
np.fill_diagonal(cosine, 0 )
similarity_with_lipstick = pd.DataFrame(cosine,index=final_lipstick.index)
similarity_with_lipstick.columns=final_user.index
similarity_with_lipstick.head()
```

Code 4-10: Interpolation and calculate the similarity matrix

For one specific user, we select the required number of the most similar users according to the value of similarity. Then, we first find the lipsticks which have been rated by these similar users, but not been rated by this user. For each of them, we predict the rating value this user may give based on the original rating value that users similar to him have given. Finally, we sort these lipsticks according to the rating predicted and retrieve the required number of results (Code 4-11).

```
def getRecommendation(user,sim_top_n,numOfRecommendation):
    Lipstick_rated_by_user = check.columns[check[check.index==user].notna().any()].tolist()
    a = sim_top_n[sim_top_n.index==user].values
    b = a.squeeze().tolist()
    d = Lipstick_user[Lipstick_user.index.isin(b)]
    l = ','.join(d.values)
    Lipstick_rated_by_similar_users = l.split(',')
    Lipstick_under_consideration = list(set(Lipstick_rated_by_similar_users)-set(Lipstick_rated_by_user))
    Lipstick_under_consideration = list(Lipstick_under_consideration)
    score = []
    for item in Lipstick_under_consideration:
        c = final_lipstick.loc[:,item]
        d = c[c.index.isin(b)]
        f = d[d.notnull()]
        avg_user = Mean.loc[Mean['userID'] == user,'rating'].values[0]
        index = f.index.values.squeeze().tolist()
        corr = similarity_with_lipstick.loc[user,index]
        fin = pd.concat([f, corr], axis=1)
        fin.columns = ['adg_score','correlation']
        fin['score']=fin.apply(lambda x:x['adg_score'] * x['correlation'],axis=1)
        nume = fin['score'].sum()
        deno = fin['correlation'].sum()
        final_score = avg_user + (nume/deno)
        score.append(final_score)
    data = pd.DataFrame({'lipstickID':Lipstick_under_consideration,'score':score})
    recommendation = data.sort_values(by='score',ascending=False).head(numOfRecommendation)
    recommendation = recommendation.lipstickID.values.tolist()
    return recommendation
```

Code 4-11: Calculate the possible rating and return the results.

The function `getRecommendationFromTopLipsticks()` first reads the user's preference from the *Users* collection. For preferences except color,

range query on *topLipsticks* collection in MongoDB is used to collect the lipsticks appealing to users preferences (Code 4-12).

```
#Get the lipsticks from Toplipsticks meet the requirement of kind and texture
filteredresult = client.db.topLipsticks.find({'liquid':{'$in':liquid},'texture':{'$in':texture}},
                                             {'_id': 0, 'rank': 1,'brand': 1, 'series': 1,
                                              'liquid': 1, 'texture': 1, 'color': 1,
                                              'price': 1, 'name': 1, 'lipstick_id': 1})
```

Code 4-12: Range query in MongoDB to collect possible results.

After range query operation, we first filter out the lipsticks which also exist in *likes* and *cfRecommendation* collection for this user. Then, for each color this user likes, we calculate the RGB distance between this color and the color of the remaining lipsticks. Then, we select the required number of the closest lipsticks for each color and aggregate them (Code 4-13).

```
#Get Top num lipsticks closet to different color
topLipsticksResult = pd.DataFrame(columns=['rank','brand', 'series', 'liquid', 'texture',
                                           'color', 'price', 'name', 'lipstick_id'])
for item in color:
    itemResult = getLipstickWithShortestNDistance(df, item, num)
    topLipsticksResult = pd.concat([itemResult, topLipsticksResult])
```

Code 4-13: Select possible results by RGB distance and aggregate results.

Finally, we drop the duplicates in the aggregation, sort the remaining lipsticks according to the `rank` feature in ascendant order, and return the required number of results on the top of the sorted list (Code 4-14).

```
topLipsticksResult = topLipsticksResult.drop_duplicates()
topLipsticksResult = topLipsticksResult.sort_values(by='rank',ascending=False).head(num)
```

Code 4-14: Drop duplicates, sort the result according to the rank feature and select the required number of lipsticks.

4.3.4 Lipstick Detail Information

4.3.4.1 Interface Introduction

Users can check the detailed information and access purchase links in the "Lipstick Detail Information Screen". The color of the lipstick, lipstick name, brand, series number, texture and price will be displayed in detail. Users can also click the "LIKE" button to like or unlike the lipstick (Figure 4-10). The purchase links will direct users to this lipstick's links in the Taobao, Amazon or Ebay by opening the browser.

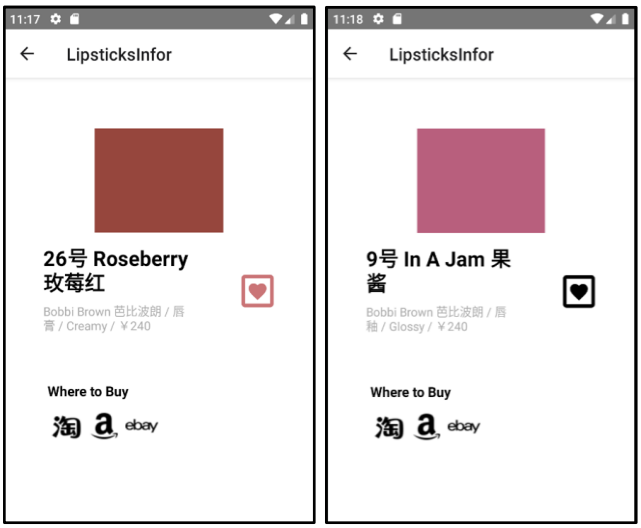

Figure 4-10: Left: Lipstick labeled as like. Right: Lipstick didn't label as like.

4.3.4.2 Frontend Code

The structure of this screen is straightforward. The first row is a square filled with the lipstick color. The second row is the name, brand, series, texture and price of this lipstick, and a heart icon within `TouchableOpacity`. The bottom row display purchase links.

The state of the like label is consistent with the “Home Screen” through navigation parameters transmission. Every time a user clicks the heart icon to switch like state, it will be reset in the state, and also be sent to the server with the fetch POST method.

To assist the recommendation system, we need to record the count that the user has visited the interface. We add a `navigation.addListener` in `componentDidMount()` to record the focus times of this lipstick for this user, and send it to the server by fetch POST method.

4.3.4.3 Backend Code

The `markIfLike()` is used to update the `lipstickID` of the *likes* collection and the rating value of the *rating* collection according to the state of the `like` label (whether the user marks the specific lipstick as like or not) in the frontend. Once the frontend posts the `userID`, `password`, the specific `lipstickID` and the `like` label to the server, the server will validate this user’s authentication before further processing:

- If the `like` label is true, which means the specific lipstick is the user liked, then append this `lipstickID` to the like list of this user in the *likes* collection (if `lipstickID` is not in existing like list), and update the rating value of this `lipstickID` to 10 in the *rating* collection.
- If the `like` label is false, remove this `lipstickID` from the existing like list, and update the rating value of this `lipstickID` to 0 in the *rating* collection.

The `focusOnLipstick()` method is used to calculate the rating of specific lipstick for the user. We do password authentication first for security, then update the `rating` feature in the *rating* collection by increasing one for each visiting (if the value of rating is below 10).

4.3.5 Lipstick Recognition

4.3.5.1 Interface Introduction

The lipstick recognition is achieved in the "Find Screen". To recognize the lipstick in a profile image, user need to follow below steps:

Figure 4-11: "Find Screen" in Lipstick Finder.

Figure 4-12: Select profile image in "Find Screen".

1. Click the circle button in the center of "Find Screen" as shown in Figure 4-11 to select a profile image that the user wants to detect the lipstick (Figure 4-12).

Figure 4-13: User can rotate and crop the image before confirming.

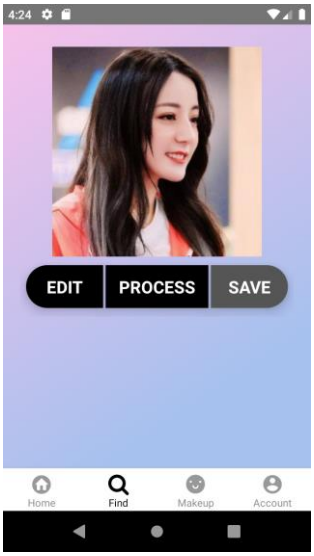

Figure 4-14: Profile image appears in "Find Screen".

2. Users can rotate and resize the photo as shown in Figure 4-13 before uploading. Once confirmed, the photo will replace the previous circle button and displayed together with a button groups: EDIT, PROCESS, and SAVE.
3. Press "EDIT" button: four image adjusting sliders will appear for the user to adjust color tone. If the user thinks the color tone of the original photo is too warm, adjusting "Temperature" can make the photo appear in a colder tone (see the demo from Figure 4-15 to Figure 4-18). The recognition result will be affected by the user's adjustment. For example, the colors of lipsticks returned in Figure 4-16 are much colder than those returned in Figure 4-18.
4. Press "PROCESS" button: the server will analysis and extract the most representative color of the target person's lips from the photo. After scanning through the whole lipstick database, the server will transfer back the top three lipsticks with the most similar color.

5. Press “SAVE” button: if the user adjusts the photo, this button will be activated so that the user can store the adjusted image to the local album.

Figure 4-15: User can press “EDIT” to adjust color tone.

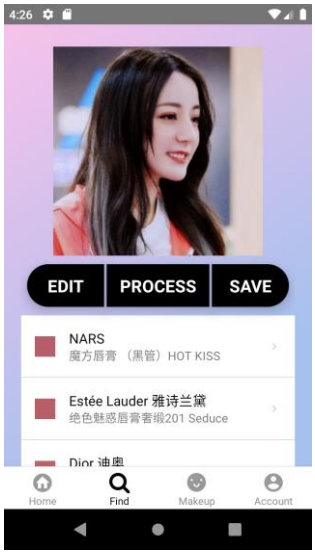

Figure 4-16: After pressing “PROCESS”, the user will get three lipsticks recognized.

Figure 4-17: Adjust the photo to have a cold color tone.

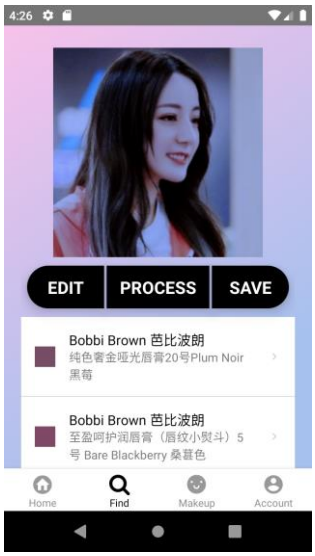

Figure 4-18: For the same image with colder color tone, the lipsticks recognized will also be affected.

4.3.5.2 Frontend Code

There are four Javascript files being called in “Find Screen”:

- LipstickFinder/res/views/Findscreen.js: it contains all functions that need to render this view and support lipstick recognition.
- LipstickFinder/res/style:
    - Filter.js: a customized slider component to edit images.
    - Styles.js: the style sheet for components.
    - Colors.js: the color text or hex codes for components.

```
import ImagePicker from 'react-native-image-crop-picker';
    chooseImage = () => {
      ImagePicker.openPicker({
        width: 512,
        height: 512,
        cropping: true,
        mediaType: 'photo',
        includeBase64: true,
        compressImageMaxWidth: 512,
        compressImageMaxHeight: 512,
        compressImageQuality: 1,
        cropperCircleOverlay: true,
        avoidEmptySpaceAroundImage: true, // ios
      }).then(image => {
        this.setState({
          photoPath: image.path,
          photoMime: image.mime,
        });
      });
    };
```

Code 4-15: Function chooseImage() in Findscreen.js.

The ImagePicker in function `chooseImage()` in Code 4-15 supports the user to choose an image from the device's local album. Setting `cropping` to true allows the user to rotate, scale and crop the image before confirming. The image selected will be cropped in size 512×512, which is exactly the size of the image being processed in the face-parsing model in the backend. After confirming the image, the image's path and mime (e.g., image/jpeg) will be stored in state and are waiting to be further processed: either do extra photo editing, or send to the

server.

```
renderImage = () => {
  console.log(this.state.photoPath);
  if (this.state.photoPath) {
    const width = styles.imgWindow.width;
    return (
      <View style={styles.upperContainer}>
        <TouchableOpacity onPress={this.chooseImage} style={{alignItems: 'center'}}>
          <Surface style={styles.imgWindow} ref={ref => (this.image = ref)}>
            <ImageFilters {...this.state} width={width} height={width}>
              {{uri: this.state.photoPath}}
            </ImageFilters>
          </Surface>
        </TouchableOpacity>
```

A

```
        <View style={{flexDirection: 'row'}}>
          <TouchableOpacity onPress={() => this.setState({collapsed: !this.state.collapsed})}
                            style={[styles.btnProcess, {width: 90, borderTopRightRadius: 0, borderBottomRightRadius: 0}]}>
            <Text style={styles.btnText}>EDIT</Text>
          </TouchableOpacity>
          <TouchableOpacity onPress={this.processImage}
                            style={[styles.btnProcess, {width: 120, borderRadius: 0, marginHorizontal: 3}]}>
            {this.state.loading
              ? <ActivityIndicator size='large' color='white'/>
              : <Text style={styles.btnText}>PROCESS</Text>}
          </TouchableOpacity>
          <TouchableOpacity onPress={this.saveImage}
                            disabled={!this.image}
                            style={[styles.btnProcess, {width: 90, borderTopLeftRadius: 0, borderBottomLeftRadius: 0, backg
            <Text style={styles.btnText}>SAVE</Text>
          </TouchableOpacity>
        </View>
      </View>
    );
  } else {
    return (
```

B

```
      <View style={styles.centerContainer}>
        <TouchableOpacity onPress={this.chooseImage} style={styles.btnCircle}>
          <Icon type={'font-awesome-5'} name={'camera-retro'} size={100} color={'white'}/>
        </TouchableOpacity>
      </View>
    );
  }
};
```

C

Code 4-16: Function renderImage() in Findscreen.js. It has 3 parts: A displays the profile image chosen by the user in "Find Screen"; B is a button group row with 3 buttons (EDIT, PROCESS and SAVE); C is a circle button.

The function `renderImage()` in Code 4-16 supports the transition from Figure 4-11 to Figure 4-14. The code logic is: a circle button (Code 4-16-C) will be rendered if there is no image chosen, otherwise the profile image (Code 4-16-A) and a button group (Code 4-16-B) will be shown.

```
11  import ImageFilters from 'react-native-gl-image-filters';
12  import Filter from '../style/Filter';
13  import Toast from 'react-native-simple-toast';
14
15  const FILTER_SETTINGS = [
16    {key: 'temperature', name: 'Temperature 色温', initValue: 6500.0, minValue: 2000.0, maxValue: 20000.0},
17    {key: 'brightness', name: 'Brightness 亮度', initValue: 1.0, maxValue: 2.0},
18    {key: 'contrast', name: 'Contrast 对比度', initValue: 1.0, maxValue: 2.0},
19    {key: 'saturation', name: 'Saturation 饱和度', initValue: 1.0, maxValue: 2.0},
20  ];
165             {this.renderImage()}
166             <Collapsible collapsed={this.state.collapsed} align="center" style={styles.slider}>
167               {FILTER_SETTINGS.map(filter => (
168                 <Filter key={filter.key} name={filter.name} minimum={filter.minValue} maximum={filter.maxValue}
169                         initValue={filter.initValue} onChange={value => this.setState({[filter.key]: value})}/>
170               ))}
171             </Collapsible>
```

Code 4-17: Photo editing sliders in Findscreen.js. There are four sliders: color temperature, brightness, contrast and saturation. The editor is collapsible.

Before sending the image to the server to recognize the lipstick, the user can modify the image to correct the color of lips they actually would like to know as shown in Figure 4-15 and Figure 4-17. This process is supported by `ImageFilters` and `Filter` from package react-native-gl-image-filters [12]. Changing the value of a slider will update the corresponding visual effect in Code 4-16-A.

```
const PREDICT_URL = 'http://124.156.143.125:5000/predict';
processImage = async () => {
  const editedPath = await this.getEditedImage();
  const photoPath = editedPath ? editedPath : this.state.photoPath;
  const photoMime = this.state.photoMime;
  RNFetchBlob.fetch(
    'POST',
    PREDICT_URL,
    {
      Authorization: 'Bearer access-token',
      'Content-Type': 'application/octet-stream',
    }, [{
      name: 'file',
      filename: 'filename',
      type: photoMime,
      data: RNFetchBlob.wrap(photoPath),
    }]).then(res => {
    this.setState({
      lipsticks: res.json(),
    });
  }).catch(err => {
    alert(err);
  });
};
```

Code 4-18: Function processImage() in Findscreen.js. RNFetchBlob will send images to the server with POST request, and store lipsticks returned from the server to state for further rendering.

After pressing the "PROCESS" button, the function `processImage()` in Code 4-18 will be called. It will first check whether there is any modification to the image, then use the corresponding image path. Here we use `RNFetchBlob` [13] to handle POST requests, which will send an image file to the server and parse server's response to a list of lipsticks. The lipsticks obtained will be rendered as soon as the state is updated as shown in Figure 4-16 and Figure 4-18.

```
getEditedImage = async () => {
  if (this.image) {
    const result = await this.image.glView.capture();
    return result.uri;
  }
  return null;
};

saveImage = async () => {
  const editedPath = await this.getEditedImage();
  if (editedPath) {
    this.setState({photoPath: editedPath});
    const d = new Date();
    const path = RNFetchBlob.fs.dirs.DCIMDir + `/Images/LipstickFinder${d.getHours()}${d.getMinutes()}${d.getSeconds()}.jpg`;
    RNFetchBlob.fs.writeFile(path, editedPath, 'uri').then(res => {
      if (res > 0) {
        Toast.showWithGravity('Saved', Toast.SHORT, Toast.CENTER);
      } else {
        Toast.showWithGravity('Error', Toast.SHORT, Toast.CENTER);
      }
    });
  }
};
```

Code 4-19: Function saveImage() in Findscreen.js. RNFetchBlob will save the adjusted image to the device's local album.

After pressing the “SAVE” button, the function saveImage() in Code 4-19 will be called. It will capture the adjusted image and save it to the local album using `RNFetchBlob`.

4.3.5.3 Backend Code

BiSeNet is utilized in our recognition algorithm for image segmentation. We use the pre-trained model which is provided by zllrunning [1]. To avoid repeated loading during function calls, we load the pre-trained model into memory in advance (Code 4-20).

```
# Load into memory
# Load model
n_classes = 19
net = BiSeNet(n_classes=n_classes)
save_pth = '/home/ubuntu/face-parsing.PyTorch/res/cp/79999_iter.pth'
net.load_state_dict(torch.load(save_pth, map_location=torch.device('cpu')))
net.eval()
```

Code 4-20: Load the pre-trained model into the memory

The Flask framework handles the POST request from the frontend. The recognition API will first save the request image on the server (Code 4-21).

```python
# Recognition
@app.route('/predict', methods=['POST'])
def predict():
    if request.method == 'POST':
        #Save image
        image = request.files['file']
        fileName = str(time.time())
        filePath = predictDataDir+fileName+'.jpg'
        image.save(filePath)
```

Code 4-21: Save the request image on the server

Now use the loaded model to do the image segmentation (Code 4-22). `parsing` is the 512×512 human face feature code matrix, in which `12` represents the upper lip and `13` represents the lower lip. `upperLipPos` and `lowerLipPos` mark the upper lip region and the lower lip region in two lists.

```python
with torch.no_grad():
    image = Image.open(filePath).convert('RGB')
    imgArray = np.array(image)
    image = to_tensor(image)
    image = torch.unsqueeze(image, 0)
    out = net(image)[0]
    parsing = out.squeeze(0).cpu().numpy().argmax(0)
    upperLipPos = np.where(parsing == 12)
    lowerLipPos = np.where(parsing == 13)
```

Code 4-22: Get human face feature code matrix

After checking the existence of the lips region, the average RGB value of the lips region will be calculated and the top three similar lipsticks are returned to the frontend (Code 4-23).

```
if len(upperLipPos[0]) > 0 or len(lowerLipPos[0]) > 0:
    pointsCount = len(upperLipPos[0])+len(lowerLipPos[0])
    rgb = [0, 0, 0]
    if len(upperLipPos) > 0:
        rgbUpper = np.sum(imgArray[upperLipPos[0], upperLipPos[1], :], axis=0)
        rgb = np.sum([rgb, rgbUpper], axis=0)
    if len(lowerLipPos) > 0:
        rgbLower = np.sum(imgArray[lowerLipPos[0], lowerLipPos[1], :], axis=0)
        rgb = np.sum([rgb, rgbLower], axis=0)
    res = [math.floor(i/pointsCount) for i in rgb]
    # calculate the distance between extracted RGB and all lipsticks' RGB
    for lipstick in lipsticks_db:
        lipstick['distance'] = ColourDistance(res, lipstick['RGB'])
    result = sorted(lipsticks_db, key=lambda x: x['distance'])[:3]
    return jsonify(result)
else: return jsonify([])
```

Code 4-23: Calculate the average RGB value and return top 3 similar lipsticks

4.3.6 Lip Makeup

4.3.6.1 Interface Introduction

After knowing the lipstick, users can try the color in their or someone else's face in the "Makeup Screen" following below steps:

1. Click the circle button in the "Makeup Screen", and choose a target person's profile photo from the local album (Figure 4-19).

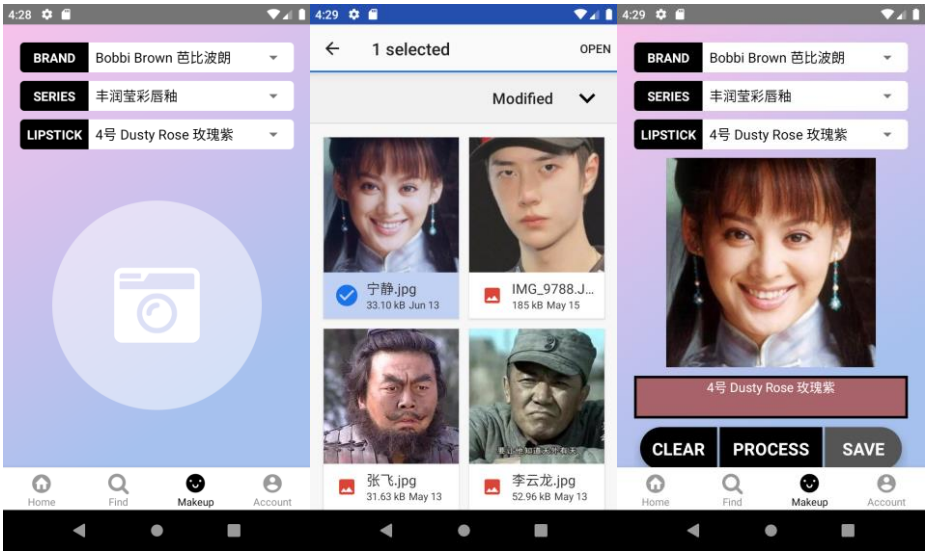

Figure 4-19: "Makeup Screen" in Lipstick Finder. Select a profile image.

2. Pick a lipstick by brand, series and name from the drop-down lists. The rectangle tile below the image shows the color of this specific lipstick. After pressing “PROCESS”, the image together with the lipstick color will be sent to the server, which will color the target person’s lips and return the updated image back to the user (Figure 4-20).

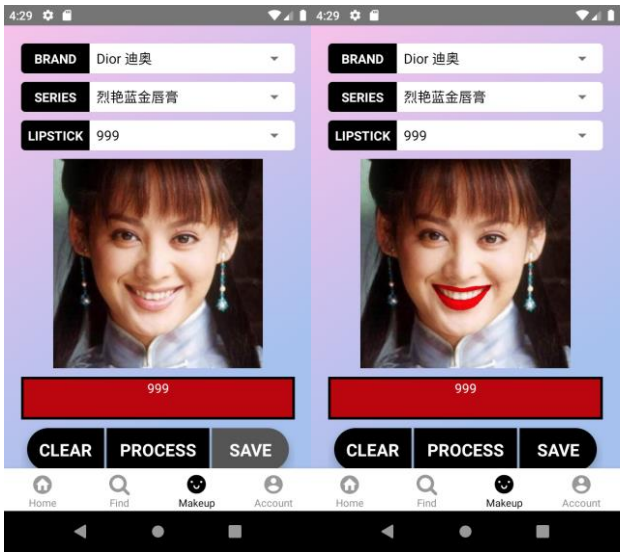

Figure 4-20: Pick a lipstick by brand, series and lipstick’s name. After pressing “PROCESS”, the lip of the target person will be colored with the lipstick

3. Clicking a lipstick from the list in “Find Screen”, the user will be navigated to “Makeup Screen”, and color tile is split into 2 pieces with left one holding the color of the lipstick from “Find Screen” and the right one holding the color of the lipstick picked in “Makeup Screen”. User can select one color tile and press the “PROCESS” button, the target person’s lips will be colored with corresponding lipstick as shown in Figure 4-21. Users can evaluate whether the color of this lipstick suits the person.

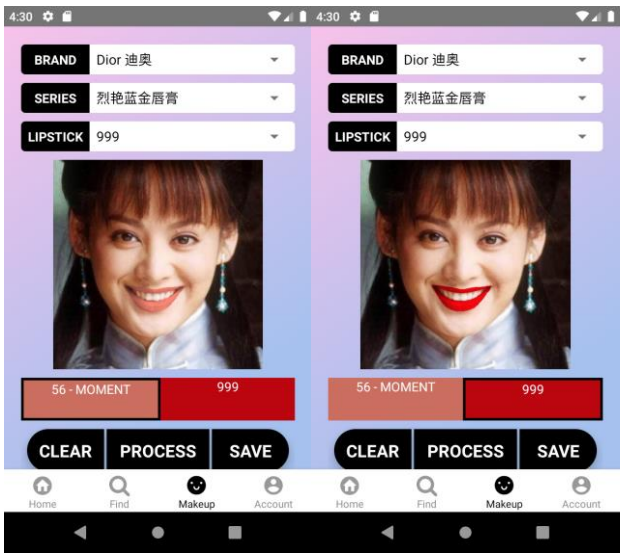

Figure 4-21: Click a lipstick found in "Find Screen", the color tile will be split into 2 pieces: the one on left is the lipstick found in "Find Screen", the one on the right is the lipstick picked in "Makeup Screen".

4. Once the image is updated in lip color, users can press the "SAVE" button in "Makeup Screen" to save the image to the local album (Figure 4-22). Additionally, users can click the image area and try the makeup with another profile image. Figure 4-23 shows the results of three males' lip makeup. Although the lip regions in later two persons are hard to determine (their original lip colors are similar to their skin), our system still performed well and quickly.

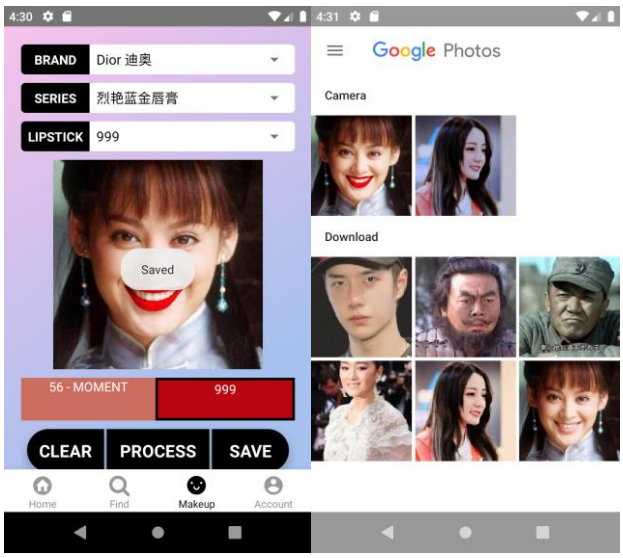

Figure 4-22: Press the "SAVE" button, the updated image will be saved to the local album.

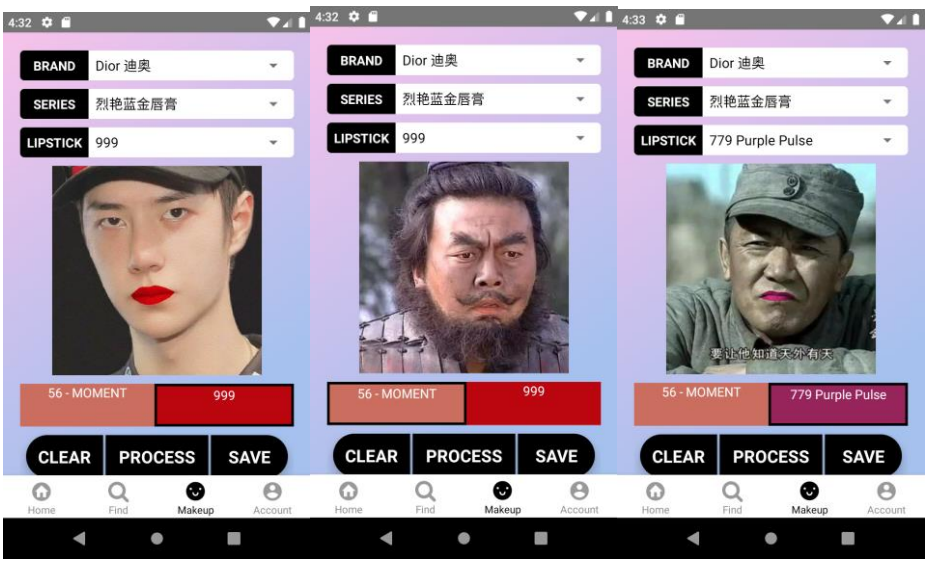

Figure 4-23: the lip makeup results of three males.

4.3.6.2 Frontend Code

There are four Javascript files being called in "Makeup Screen":

- LipstickFinder/res/views/Makeupscreen.js: it contains all functions that need to render this view and support lip makeup requests.

- LipstickFinder/res/style:
    - Styles.js: the style sheet for components.
    - Colors.js: the color text or hex codes for components.
- LipstickFinder/res/utils/asyncstorage.js: it contains util functions to store, get and remove data from `AsyncStorage` [14].

A

```
260           <View style={styles.PickerContainer}>
261             <View style={styles.PickerLabel}>
262               <Text style={styles.PickerLabelText}>BRAND</Text>
263             </View>
264             <Picker mode={PICKER_MODE} selectedValue={this.state.selectedBrandID} style={styles.Picker}
265                     onValueChange={(itemValue, itemIndex) =>
266                       this.onBrandSelected(itemValue)
267                     }>
268               {this.state.brands.map((brand, index) => (
269                 <Picker.Item key={index} label={brand.name} value={brand.id}/>
270               ))}
271             </Picker>
272           </View>
```

```
183   componentDidMount = async () => {
184     console.debug('componentDidMount');
185     this.setState({
186       photoData: await getData('photoData'),
187       photoPath: await getData('photoPath'),
188       photoMime: await getData('photoMime'),
189       // lipsticksDB: await getData('lipsticksDB'),
190     });
191     fetch(URLS.BRANDS).then((response) => response.json()).then(responseJson => {
192       this.setState({brands: responseJson});
193     }).catch(error => {
194       alert(error);
195     });
```

```
213   onBrandSelected = (brand_id) => {
214     console.debug('onBrandSelected:', brand_id);
215     this.setState({selectedBrandID: brand_id});
216     fetch(
217       URLS.SERIES + `brand_id=${brand_id}`,
218     ).then((response) => response.json()).then(responseJson => {
219       this.setState({series: responseJson});
220     }).catch(error => {
221       console.error(error);
222     });
223   };
```

B

```
274           <View style={styles.PickerContainer}>
275             <View style={styles.PickerLabel}>
276               <Text style={styles.PickerLabelText}>SERIES</Text>
277             </View>
278             <Picker mode={PICKER_MODE} selectedValue={this.state.selectedSeriesID} style={styles.Picker}
279                     onValueChange={(itemValue, itemIndex) =>
280                       this.onSeriesSelected(itemValue)
281                     }>
282               {this.state.series.map((series, index) => (
283                 <Picker.Item key={index} label={series.name} value={series.id}/>
284               ))}
285             </Picker>
286           </View>
```

```
288          <View style={styles.PickerContainer}>
289            <View style={styles.PickerLabel}>
290              <Text style={styles.PickerLabelText}>LIPSTICK</Text>
291            </View>
292            <Picker mode={PICKER_MODE} selectedValue={this.state.selectedLipstickID} style={styles.Picker}
293                    onValueChange={(itemValue, itemIndex) =>
294                      this.onLipstickSelected(itemValue, itemIndex)
295                    }>
296              {this.state.lipsticks.map((lipstick, index) => (
297                <Picker.Item key={index} label={lipstick.name} value={lipstick.lipstick_id}/>
298              ))}
299            </Picker>
300          </View>
```

C

Code 4-24: Three drop-down pickers to filter and select lipstick in Makeupscreen.js. A is a brand picker, B is a series picker, C is lipstick filter.

The three drop-down pickers at the top of the "Makeup Screen" are supported by the code in Code 4-24. Once a component is loaded, `componentDidMount()` will be invoked immediately. It sends a GET request to pull a list of brands, and the brand picker will be then loaded with the values with the first item as default value. The change of value in brand picker invokes `onBrandSelected()`, which sends a GET request to the server to retrieve all series under the selected brand. Similarly, the update of series picker invokes `onSeriesSelected()`, which requests the server to return all lipsticks given the brand and series chosen, then the lipstick picker invokes `onLipstickSelected()`. Once all of these are done, the screen stops rendering and is presented as Figure 4-19. If the user changes any value in the picker, the value updating and function calls will behave similarly.

```
processImage = () => {
  const color = this.state.useSelection ? this.state.selectedColor : this.props.route.params.color;
  this.setState({loading: true});
  const photoData = this.state.photoData;
  const photoMime = this.state.photoMime;
  RNFetchBlob.fetch('POST', URLS.MAKEUP, {
    Authorization: 'Bearer access-token',
    'Content-Type': 'application/octet-stream',
  }, [
    {name: 'file', type: photoMime, filename: 'filename.jpg', data: photoData},
    {name: 'color', data: color},
  ]).then(res => {
    this.setState({
      photoUpdateData: res['data'],
      photoUpdateMime: res['Content-Type'],
      changePhoto: false,
      loading: false,
    });
  }).catch(err => {
    alert(err);
  });
};
```

Code 4-25: Function processImage() in Makeupscreen.js. It sends a POST request to the server with the image file and color, and stores the returned photo.

After pressing the "PROCESS" button in the "Makeup Screen", the function `processImage()` in Code 4-25 will be invoked. The process is similar to the one in "Find Screen". It still uses `RNFetchBlob` to send a POST request with the image file together with the color code to the server. The server will do face-parsing to the target person and update the lips with the color, then returns the updated image, which is shown as Figure 4-21.

4.3.6.3 Backend Code

Using the same approach we introduce in the lipstick recognition part, we store the request image on the server and go through the pre-trained model to get `parsing`, `upperLipPos` and `lowerLipPos`. `LipMakeUp()` function is invoked to do the makeup which is based on the method we discuss in the section 3.2. Input image's hue channel and saturation channel are replaced by

the target color's corresponding channels. The rendered lips region will be used to replace the original image's lips region.

```
def LipMakeUp(image, parsing, color):
    b, g, r = color
    tar_color = np.zeros_like(image)
    tar_color[:, :, 0] = b
    tar_color[:, :, 1] = g
    tar_color[:, :, 2] = r

    image_hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    tar_hsv = cv2.cvtColor(tar_color, cv2.COLOR_BGR2HSV)

    image_hsv[:, :, 0:2] = tar_hsv[:, :, 0:2]

    changed = cv2.cvtColor(image_hsv, cv2.COLOR_HSV2BGR)
    not12and13 = (parsing != 12) & (parsing != 13)
    changed[not12and13] = image[not12and13]
    return changed
```

Code 4-26: Lip makeup function

4.3.7 Account

In this section, the user can visit his/her liked lipstick list, edit the personal information (i.e., name, gender and profile image), reset the password and know about the basic information about the application and our project team. Each function corresponds to an interface, we use drawer navigation to switch between interfaces. The left drawer is shown in Figure 4-24.

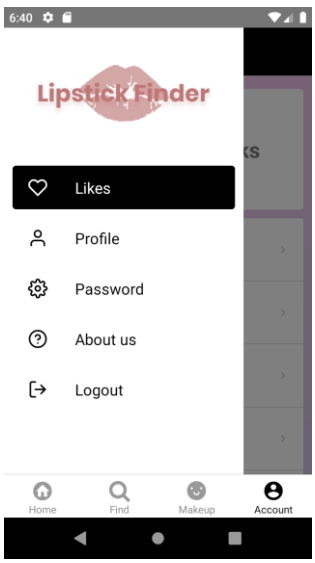

Figure 4-24: Screenshot of open drawer

4.3.7.1 Likes Screen

The “Likes Screen” shows the user name, user avatar and his/her liked lipstick list (Figure 4-25). The user can click a lipstick to enter the “Lipstick Information Detail Screen” and view its details.

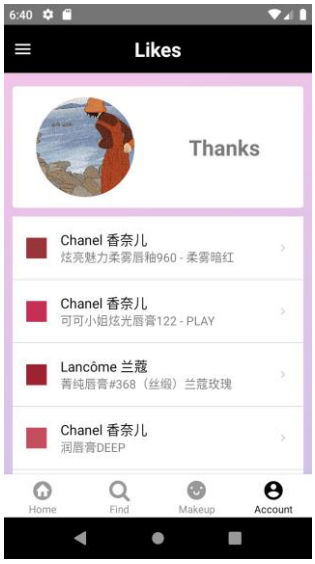

Figure 4-25: Screenshot of like screen

4.3.7.1.1 Frontend Code

In the frontend, we equip two views in a `ScrollView`. The top one contains an `Avatar` to display profile image and `Text` to display the name. We use `RNFetchBlob.fetch` to download the user's profile avatar (Code 4-27), and use the Fetch API to get the personal information. In particular, if the user has not set an avatar and name, we will show the default one. The bottom one contains a `ListItem` similar to the lipstick list in “Find Screen”. The like list of the user is also obtained with a GET request from fetch. All these requests will be used with the encrypted password for authentication.

```
RNFetchBlob.fetch('GET', URLS.PROFILEIMAGE + `userID=${userID}&pwd=${pwd}`, {
  Authorization: 'Bearer access-token',
  'Content-Type': 'application/octet-stream',
}).then(response => {
  this.setState({
    profileData: response['data'],
    profileMime: response['Content-Type'],
  });
  // console.log(response.text());
  if (response.text() == 'No image') {
    this.setState({profilePath: PROFILE_PATH});
  } else { //display the image in DB
    this.setState({profilePath: `data:${this.state.profileMime};base64,${this.state.profileData}`});
  }

}).catch(err => {
  alert(err);
});
```

Code 4-27: Profile image fetch

In order to keep the information in this interface consistent with our database, we set `navigation.addListener()` in `componentDidMount()`. Every time when the screen is focused, it will invoke `refresh()` to send these requests.

4.3.7.1.2 Backend Code

All of the requests are handled by Flask. In `getUserProfileImage()`, the server first verifies the user's identification, then reads the avatar file path recorded in the *Users* collection, and uses `send_file()` to return the image. If no avatar has been found, it responds with an error message "No image".

In the `getUserInfo()` function, we still do the user authentication firstly, and return serialized user information in JSON format.

In the `getLipstickLike()` function, user authentication should be done at first, then it will confirm whether the user exists in the *likes* collection. If the user exists, it means that it has clicked on the “like” label of lipsticks somewhere. Get all of the `lipstickID` the user labeled as `like` from the *likes* collection, then retrieve detailed information from the *lipsticks* collection for each liked

lipstick, and return them in JSON format. If the user hasn't liked any lipstick, we will return a fake data for example information.

```
@app.route('/getLipstickLike', methods=['GET'])
def getLipstickLike():
    if request.method == 'GET':
        userID = int(request.args.get("userID"))
        pwd = request.args.get("pwd")
        user = mongo.db.Users.find({'ID': userID, 'password':pwd})
        if user is None:
            return Response("Wrong Password", status=500)
        else:
            if mongo.db.likes.count_documents({'userID': userID})==1:
                reply = mongo.db.likes.find({'userID': userID}, {'_id': 0})[0]
                lipsticks = []
#                 print(reply['lipstickID'])
                for lipstickID in reply['lipstickID']:
                    lipstickInfo = list(mongo.db.lipsticks.find({'lipstick_id': lipstickID},
{'_id': 0, 'brand': 1, 'series': 1, 'liquid': 1, 'texture': 1, 'color': 1, 'price': 1, 'name': 1,
'lipstick_id': 1}))[0]
                    lipsticks.append(lipstickInfo)
#                 print(lipsticks)
                return jsonify(lipsticks)
            else:
                lipsticks_example = [{'brand': 'Here is a example', 'series': 'Storge', 'name': '
Lipstick You Like', 'liquid': True, 'texture': 'Texture', 'color': '#CA7476', 'price': 280,
'lipstick_id': '00000'},]
                return jsonify(lipsticks_example)
```

Code 4-28: Code in red box is the similar password authentication

4.3.7.2 Profile Screen

In the "Profile Screen", users can edit their name, gender and avatar. Figure 4-26 shows the modification process and results.

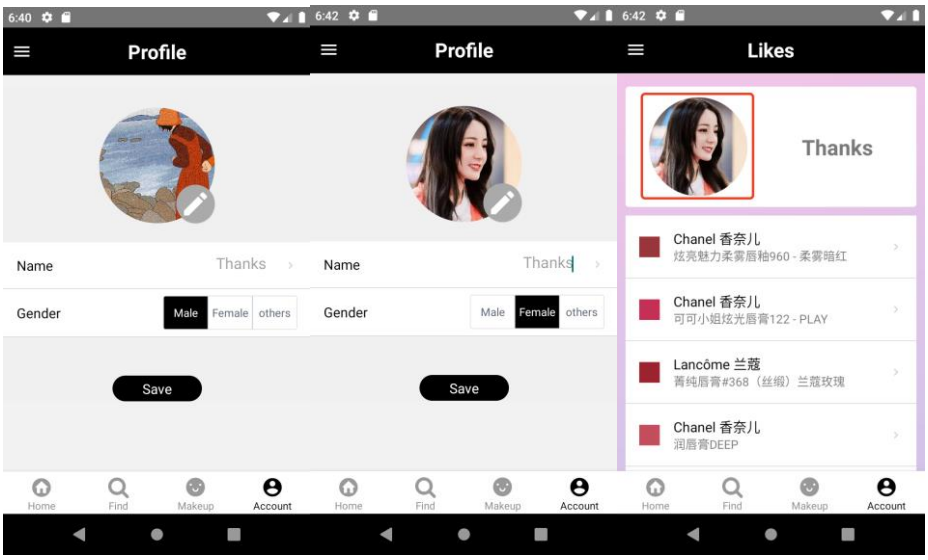

Figure 4-26: Left: Original data. Middle: Edit avatar and gender. Right: Edit result.

4.3.7.2.1 Frontend Code

We render a function `renderAvatar()` to show the profile image and two `ListItems` for name and gender.

In `renderAvatar()`, Avatar is rendered in the same way as the likes screen. The difference is that there is an accessory in the right-bottom corner. If the user clicks the accessory, `ImagePicker.openPicker()` will be invoked which supports picking one image from the album. The chosen image will be stored in state, and rendered in `renderAvatar()`.

For name modification, we use the `ListItem` with `input` and `onChangeText`, which will store the user input text as the new name in the state.

For gender modification, we use the `ListItem` with the `buttonGroup`. It will update the selected index when we switch a different button. We set a new function `updateIndex()` to change the index to different gender in the state.

There is a "SAVE" button at the bottom. Every time the user presses this button, the information in this interface will be uploaded to the server. To save network bandwidth, it is used to upload the profile picture only when the user has modified the profile picture. If the user keeps the default profile picture or does not modify the profile picture, no request is sent. We also use the Fetch API with the POST method to upload the user's name and gender.

4.3.7.2.2 Backend Code

In the `updateProfileImage()` function, after user authentication, we save the uploaded image in the server and rename it as userID.jpg, and save this path for this user in the *Users* collection.

In the `updateProfileInfo()` function, after user authentication, we update the data for this user in the *Users* collection with the uploaded name and gender.

4.3.7.3 Password Screen

In this screen, a user can change his/her password by correctly entering the current password, the new valid password and the confirmed new password consistent with the previous new password (Figure 4-27).

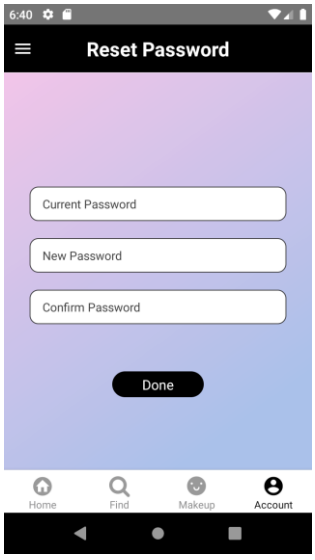

Figure 4-27: Reset password screen

4.3.7.3.1 Frontend Code

This screen contains three `TextInputs` to input the three kinds of password, which will be stored in the state, and a button for saving. When the user presses

this button, it will check whether the password of three `TextInputs` are not empty and whether the `newPassword` and `confirmPassword` are matched. If everything goes well, it will call the POST request to upload the new password, otherwise the user will be alerted with an error message.

4.3.7.3.2 Backend Code

The `resetPwd()` function is responsible for this screen. After user authentication, it will update the password with the new one in *Users* collection.

4.3.7.4 About Us Screen

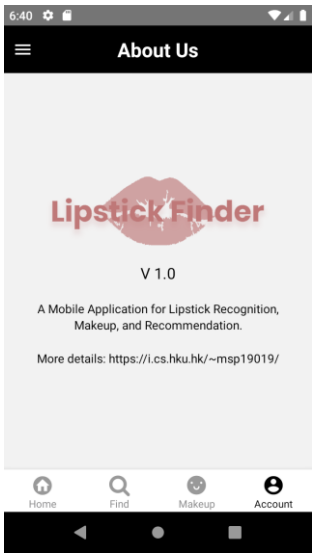

Figure 4-28: About Us screen

This screen in Figure 4-28 shows the logo, version and name of Lipstick Finder, as well as the website of this project.

5. Application Performance

5.1 Application Testing

The main purpose of this chapter is to test the integrity, logic and continuity of all functional modules of the application. There is a complete set of test cases to simulate user operations in reality. In each test case, we will explain the purpose of testing, display testing results, and pay attention to the user experience.

Since the current Lipstick Finder is a development version, the developing environment of React Native needs to be installed before installing and running the application in the Android Studio simulator. The settings of the simulator and the installation commands are shown in Appendix.

5.1.1 Test Access Control

In this section, we test the process of registration, login and logout.

5.1.1.1 Test Registration

- Route to “Registration Screen”

  Click “Sign up now” text in the “Login screen”, users can see the result shown in Figure 5-1. The interface is navigated from “Login Screen” to “Register Screen” successfully.

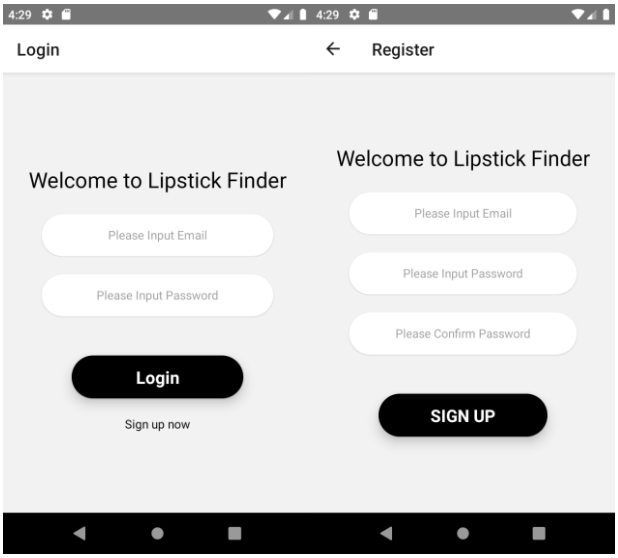

Figure 5-1: The Interface navigating from login to registration

- Sign up with illegal email or password

    1. Enter “test” as an email, and enter “12345678” as “Input Password” and “Confirm Password”, then click the “SIGN UP” button. The test result is Figure 5-2-a.
    2. Enter “test@gmail.com” as an email, and enter “12345” as “Input Password” and “Confirm Password”, then click the “SIGN UP” button. The test result is Figure 5-2-b.
    3. Enter “test@gmail.com” as an email, and enter “12345678” as “Input Password” and “123456789” as “Confirm Password”, then click the “SIGN UP” button. The test result is Figure 5-2-b.

Test results show that the application can give users a corresponding alert to instruct users to type into valid account information.

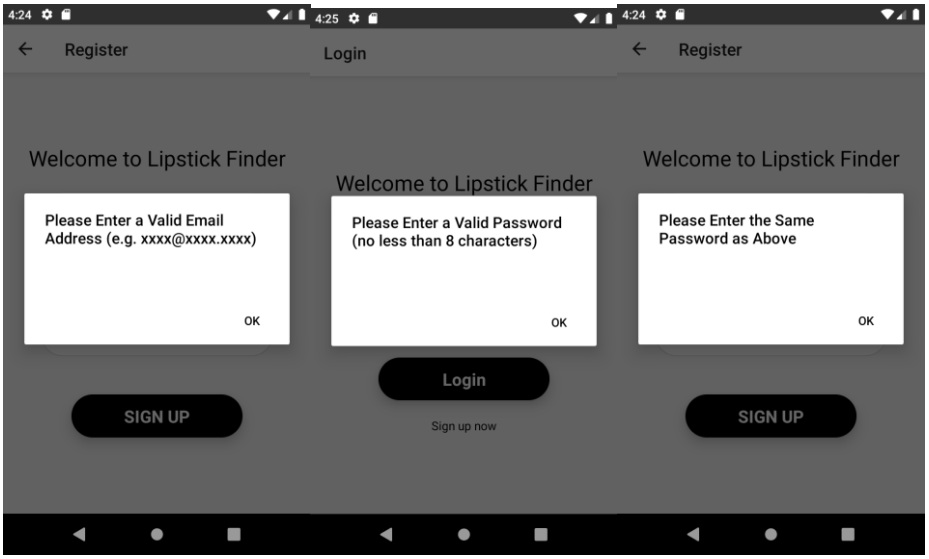

Figure 5-2: Screenshots of alert. a: illegal email, b: illegal password, c: two password not match.

- Sign up with valid email and password

  Enter "test@gmail.com" as an email, and enter "12345" as "Input Password" and "Confirm Password", then click the "SIGN UP" button. The test results are shown in Figure 5-3. The interface is successfully navigated from "Register Screen" to "Questionnaire Screen". The "SIGN UP" button works well.

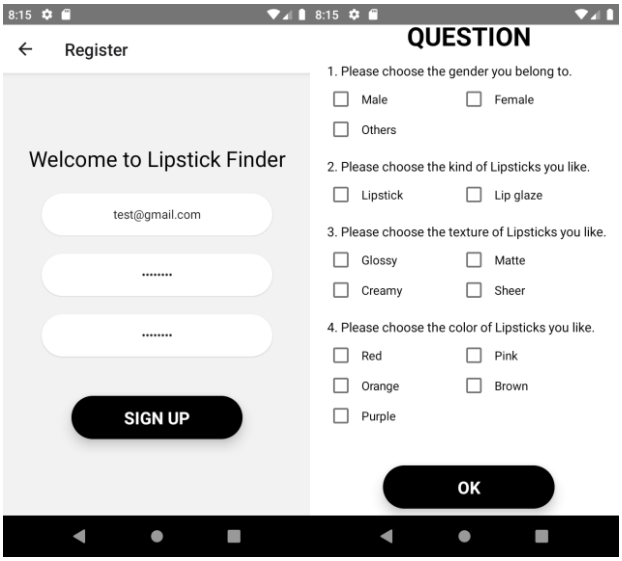

Figure 5-3: Interface jump from registration to questionnaire

- Answer question

  Click the choices for each question, and then click the “OK” button. Corresponding test results are like Figure 5-4, Figure 5-5 and Figure 5-6. The application gives different alerts according to different situations. After finishing the first single choice question and the following three multiple choices questions, clicking the “OK” button will route the screen to “Login Screen”. This “Questionnaire Screen” works well.

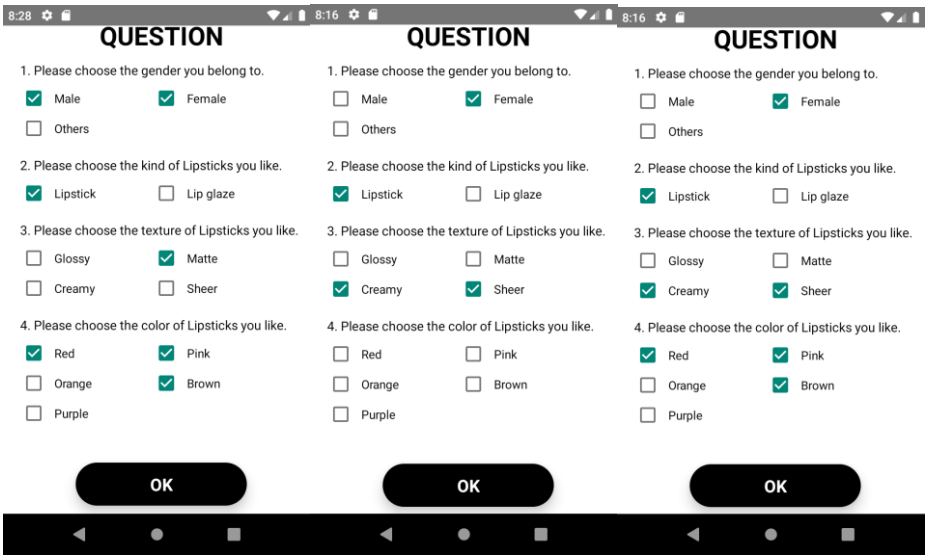

Figure 5-4: Click more than one choice in gender and its result

Figure 5-5: Missing one question and its result

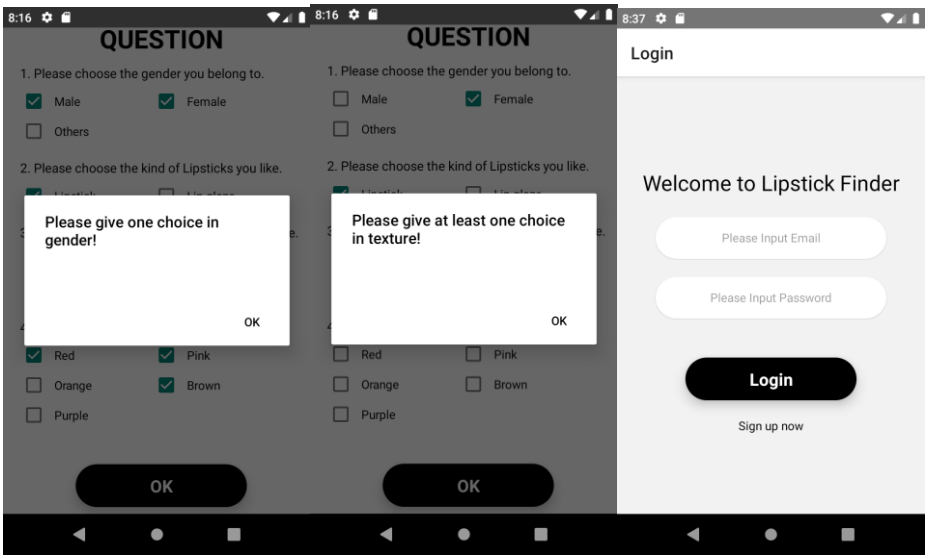

Figure 5-6: Answer all questions and its result

5.1.1.2 Test Log In

Enter “te@gmail.com” as username and “12345678” as password. Then click the “Login” button. The test result is the Figure 5-7. Enter “test@gmail.com” as username and “12345678” as password. Then click the “Login” button. The test result is Figure 5-8.

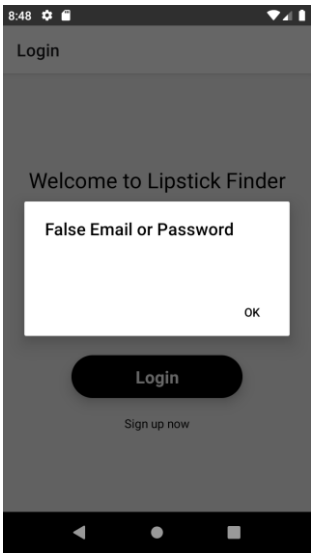

Figure 5-7: The result of wrong input

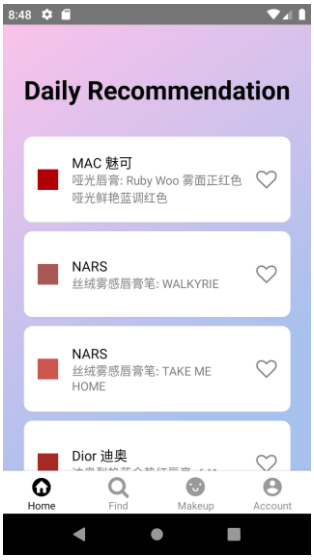

Figure 5-8: The result of correct input

In this section, Lipstick Finder can check the correctness of the email and password. If the account information is correct, the screen will be routed to "Home Screen". The login function works well.

5.1.1.2 Test Log Out

Click the "Logout" item in the drawer. The test result is shown as Figure 5-9. Users can go to the "Login Screen" directly, and cannot go back to the original screen through the "back" button, so that the logout function works well.

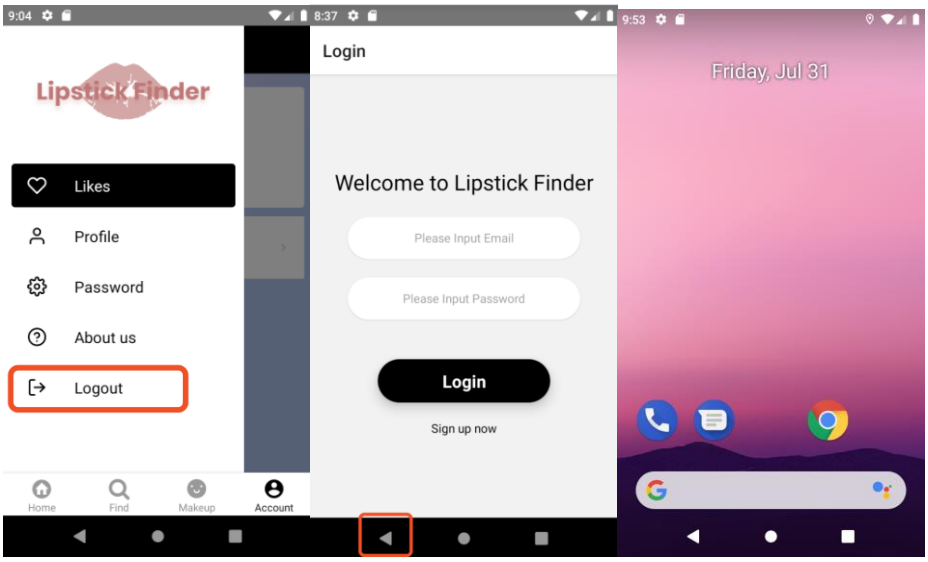

Figure 5-9: Screenshot of logout

5.1.2 Test Recommendation List, Like List and Lipstick Detail Screen

"Home Screen" is used to display the personalized recommendation for the specific user. "Lipstick Information Detail Screen" is used to display the detailed information of lipsticks. Users can label the lipstick as "like" by clicking the white heart. All lipsticks labeled as "like" will appear in the "like list" in "Account Likes Screen". In this section, we test the consistency of the information in these three interfaces.

5.1.2.1 Test Recommendation List of Home Screen

Once users successfully log into our app, the "Home Screen" will be displayed first, the test result is shown in Figure 5-10. The "like" label for these lipsticks shown in the recommendation list of "Home Screen" can be added and canceled by pressing the corresponding heart button (list item) for a short time. The test results of short-time press on "MAC Ruby Woo" and "NARS WALKYRIE"

are shown in Figure 5-10. Pressing the list item for a long time will route the screen to the “Lipstick Detail Information Screen”. The test results of long-time press on “MAC Ruby Woo” and “NARS TAKE ME HOME” are shown in Figure 5-10.

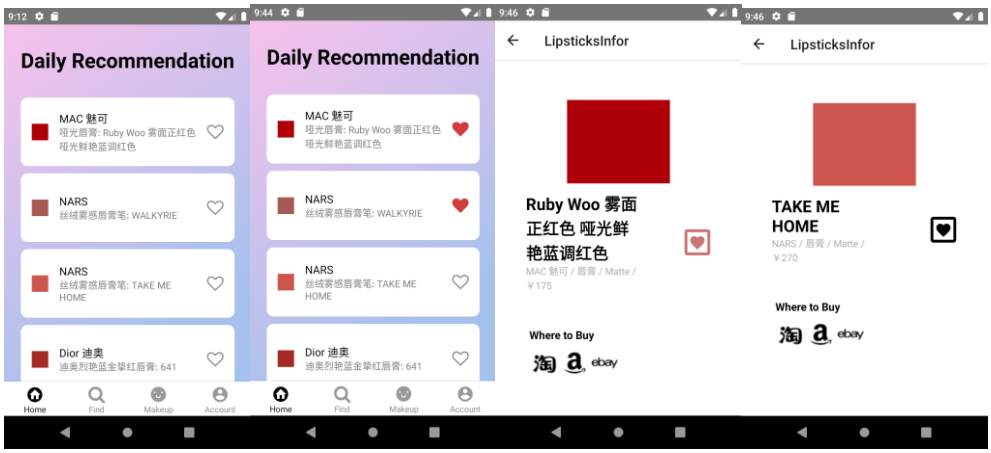

Figure 5-10: Default Home Screen, Short-time press on “MAC Ruby Woo” and “NARS WALKYRIE”, Long-time press on “MAC Ruby Woo”, and Long-time press on “NARS TAKE ME HOME”

The screen can scroll smoothly, each item can react correctly to different time intervals of pressing, and the “like” labels in these screenshots are consistent, all of which indicate that the recommendation list function works well.

5.1.2.2 Test the Like List of Account “Likes Screen”

Once we switch to the “Account Screen”, the “Likes Screen” will be displayed by default. Figure 5-11-a is the test result before pressing “MAC Ruby Woo” and “NARS WALKYRIE”. After we pressed them in “Home Screen”, the “Likes Screen” displayed these two lipsticks (Figure 5-11-b). Press the lipstick “NARS WALKYRIE” in “Likes Screen”, the test result is shown as Figure 5-11-c.

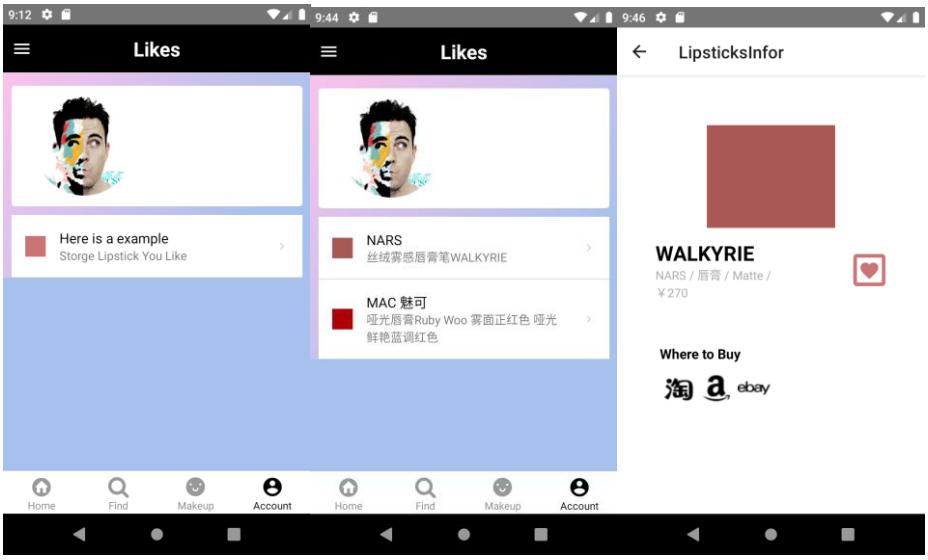

Figure 5-11: a: Default Account "Likes Screen", b: Account "Likes Screen" after short press on "MAC Ruby Woo" and "NARS WALKYRIE" in "Home Screen", c: Press on "NARS WALKYRIE"

Click the red heart in "Lipstick Details Information Screen" of "NARS WALKYRIE", then click the "back" button, the test results are shown in Figure 5-12. The results shows that the "Likes Screen" didn't show "NARS WALKYRIE" after the red heart is clicked to cancel the "like" Label.

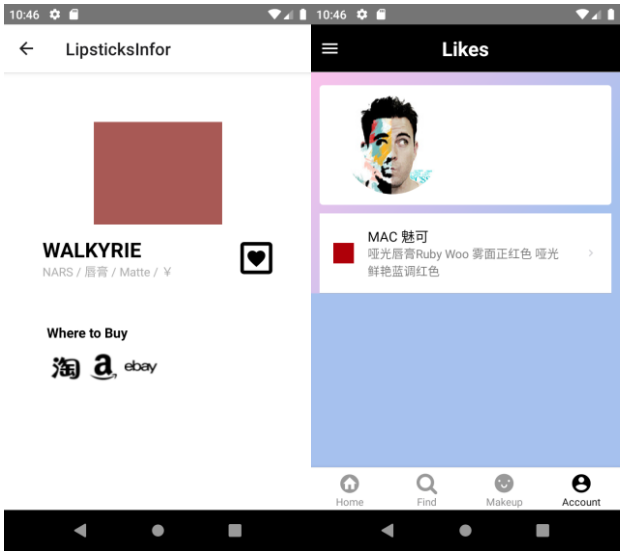

Figure 5-12: Click heart and go back

The test results above indicate that the lipstick information (such as name, brand, color) shown in the “Likes Screen”, the “Home Screen” and “Lipstick Details Information Screen” is consistent with true data, including lipsticks information and like list information.

5.1.2.3 Test “Lipstick Detail Information Screen”

Long-press “NARS WALKTRIE” in “Home Screen” to check this lipstick’s details. Click the Taobao icon to check the purchasing link in “Lipstick Details Information Screen”. Test results are shown in Figure 5-13.

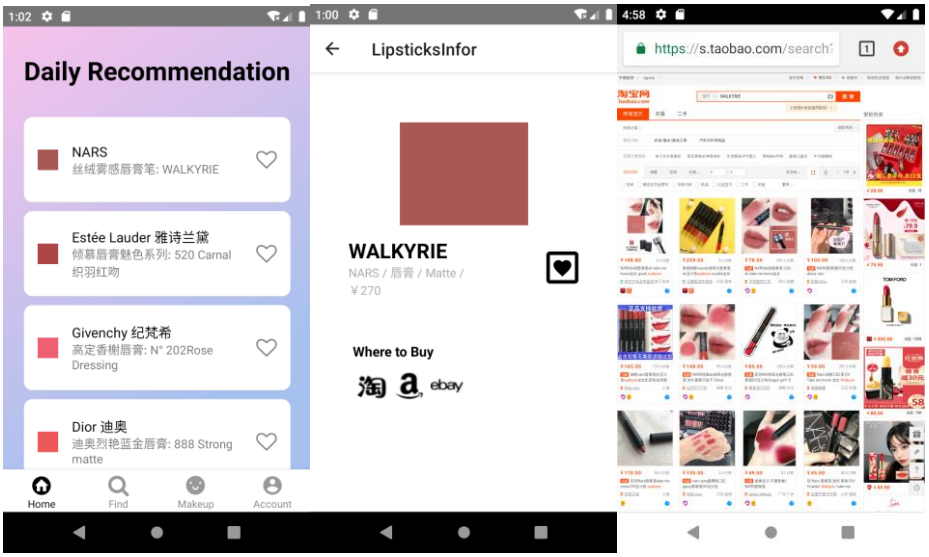

Figure 5-13: Click “NARS WALKTRIE” and click the Taobao link

The lipstick information displayed for “NARS WALKTRIE” is correct, and it can jump to the correct browser link, which indicates the “Lipstick Detail Information Screen” works well.

5.1.3 Test Account Profile Reset

In this section, we will test whether we can smoothly modify user’s information including username, gender and avatar.

Figure 5-14 shows the test result of clicking “Profile” in the account drawer. We can see the default settings of new users with a default profile image, null in name and null in gender (recognized as “Others”).

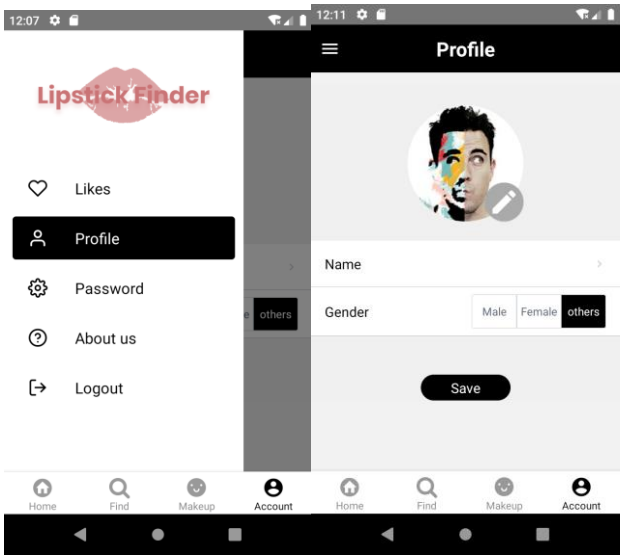

Figure 5-14: Click “Profile” item

Click the accessory icon in avatar and choose a new image from the local album. Input “test” as user name. Switch gender to “Female”. The test results are shown in Figure 5-15.

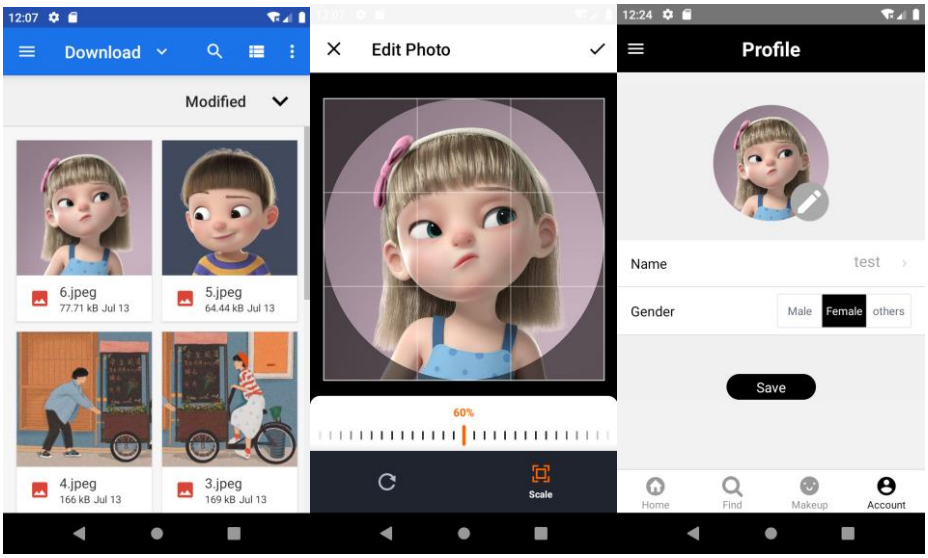

Figure 5-15: Choose an image, change name and gender

Finally click the “SAVE” button, then check whether the profile information in “Likes Screen” is updated accordingly (“Likes Screen” contains the profile image and user name). The test result shown in Figure 5-16.

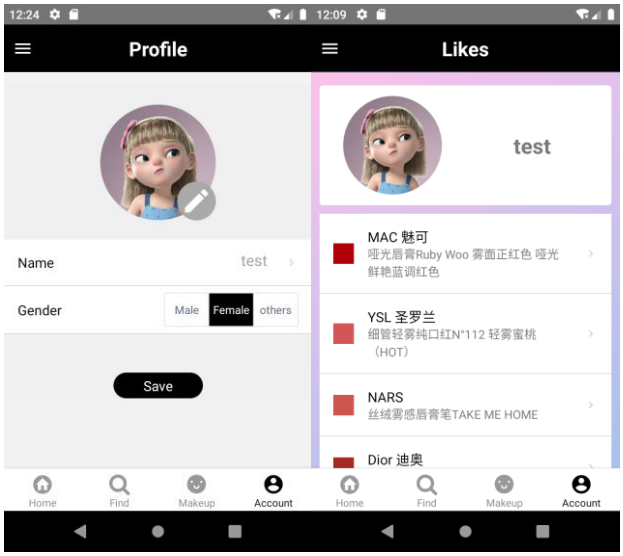

Figure 5-16: Click save button

We found that users’ information between these two screens are the same, hence the profile resetting function works well.

5.1.4 Test Password Reset

Click “Password” in the account drawer, the test result is shown in Figure 5-17.

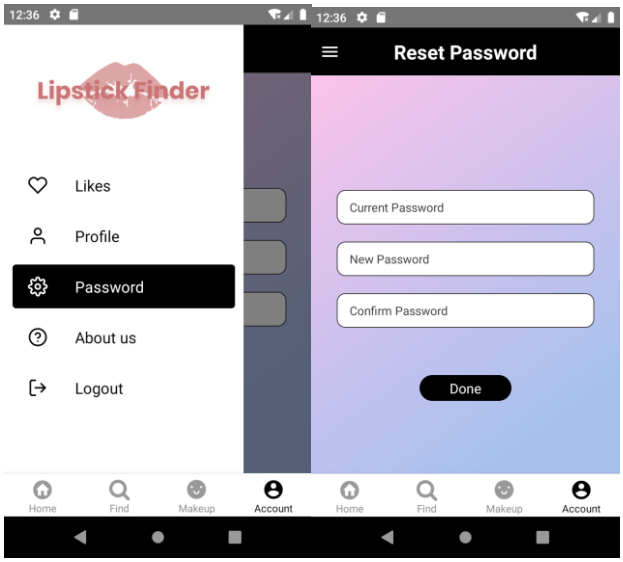

Figure 5-17: Click "Password" item

Enter the current password "12345678" and new password "qwertyuiop" for the test user. Enter "qwertyuiop" as the confirmed password in the third line, then click the "Done" button. Figure 5-18 shows the test result.

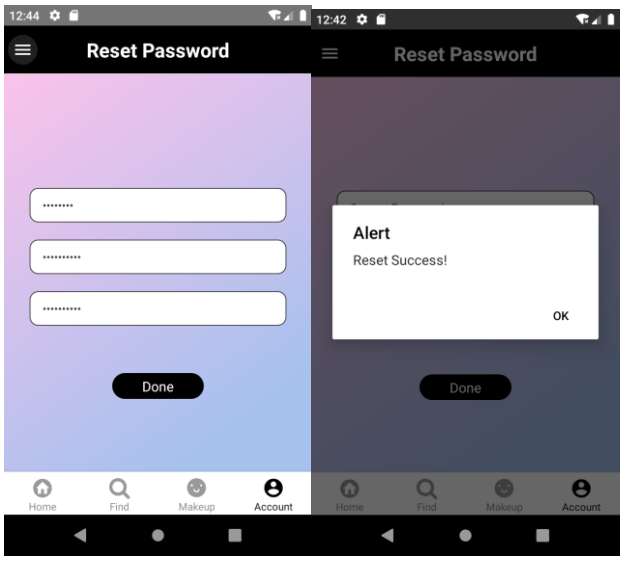

Figure 5-18: Enter passwords and save

Log out this account, and log in the same account with the new password. Test results are shown in Figure 5-19.

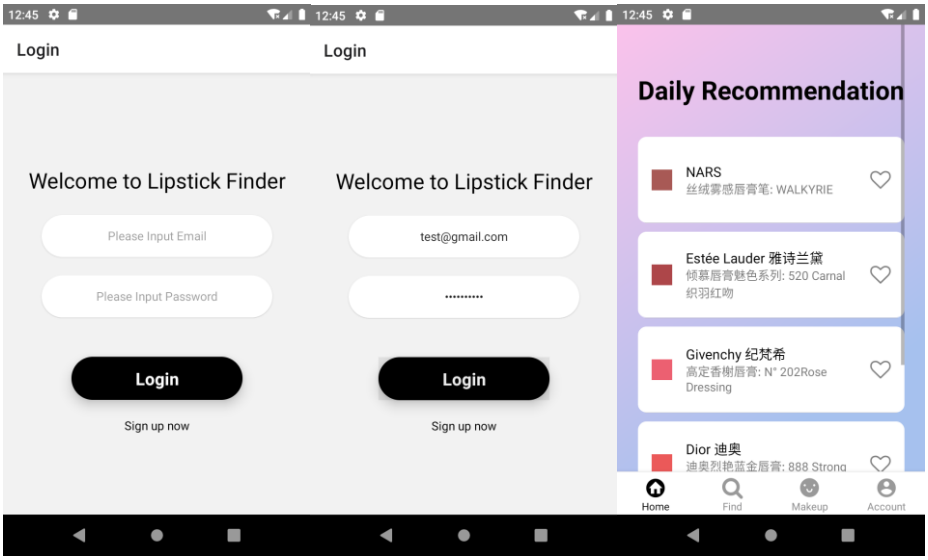

Figure 5-19: Log in with new password

The successful login with updated password indicates that the reset password function works well.

5.1.5 Test Lipstick Recognition

In this section, we will test the process of lipstick recognition with a photo of famous actress Dilraba Dilmurat.

Switch to “Find Screen”, click the circle button, and then pick the photo of Dilraba as shown in Figure 5-20. Successfully selecting the picture shows that the picker works well.

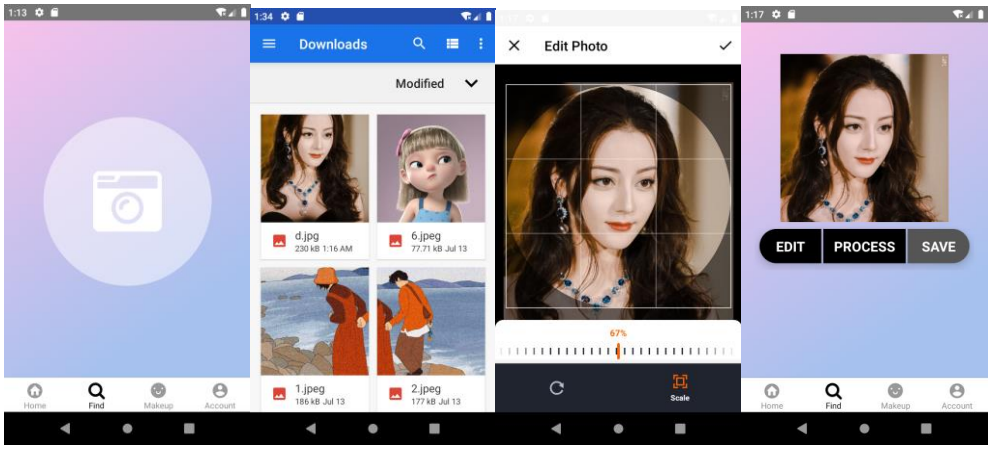

Figure 5-20: Pick Dilraba's photo

Figure 5-21 shows the test results of clicking the "PROCESS" button directly. We get three lipsticks. It seems that the color of these lipsticks is very similar to the color on Dilraba's lips, hence the "PROCESS" button works well.

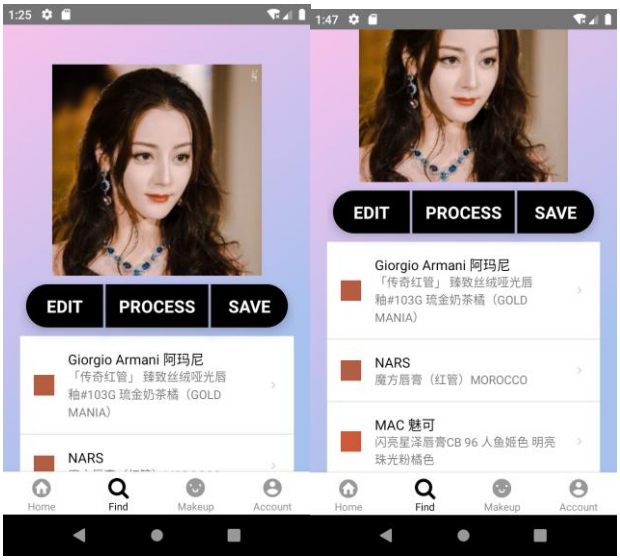

Figure 5-21 : Click the "PROCESS" button directly

Click the "EDIT" button, then reduce the temperature, improve the brightness, the test result shows like Figure 5-22. The color tone becomes bluer, it shows the "EDIT" button works well. Click the "PROCESS" button again, these colors

of recognition results are bluer than the results of the original image without edit shown in *Figure* 5-21. It indicates the recognition module works well.

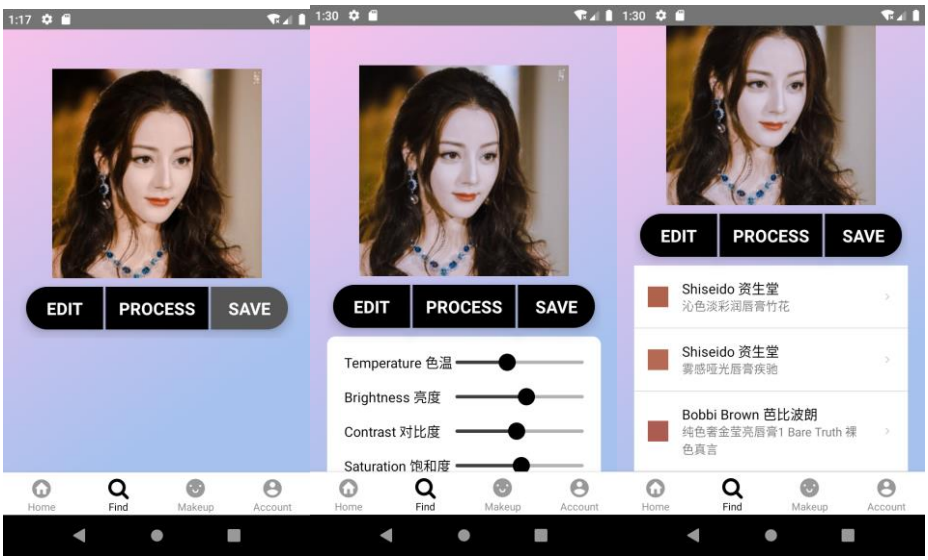

Figure 5-22: Edit, then click the "PROCESS" button

5.1.6 Test Lip Makeup

In this section, we will test the process of lip makeup with images of famous actress Yang Zi.

Switch to "Makeup Screen", the screen is default shown in Figure 5-23-a. Click the circle button, then pick a photo of Yang Zi, the picker result is shown in Figure 5-23-c. Click the "PROCESS" button, the render result shows in Figure 5-23-d.

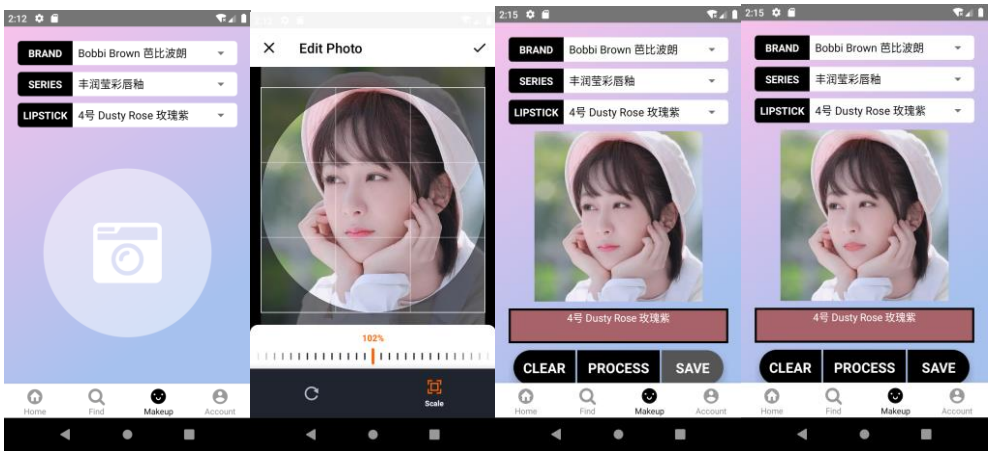

Figure 5-23: a: Default screen. b&c: Pick image d: Click "PROCESS" button

We can see the makeup results shown on Yang Zi's lip from Figure 5-23-d. The contour of the rendered part is roughly the same as the edge of her lips. The color of her lip makeup is similar to the color card of "4 号 Dusty Rose 玫瑰紫" shown below the image, which is the standard color of this lipstick "4 号 Dusty Rose 玫瑰紫". So the "PROCESS" button works well.

Change the lipstick by clicking the drop down box of "BRAND", "SERIES" and "LIPSTICK", choose "YSL 圣罗兰", "细管纯口红", and "N°12 干枯玫瑰（HOT）" respectively, then click the "PROCESS" button. The test result is shown in Figure 5-24-a. We can see that the color card changed, which means the drop down box works well. Click the "SAVE" button, we got a "Saved" alert in Figure 5-24-b, and we can find this makeup photo in the album, which means the "SAVE" button works well.

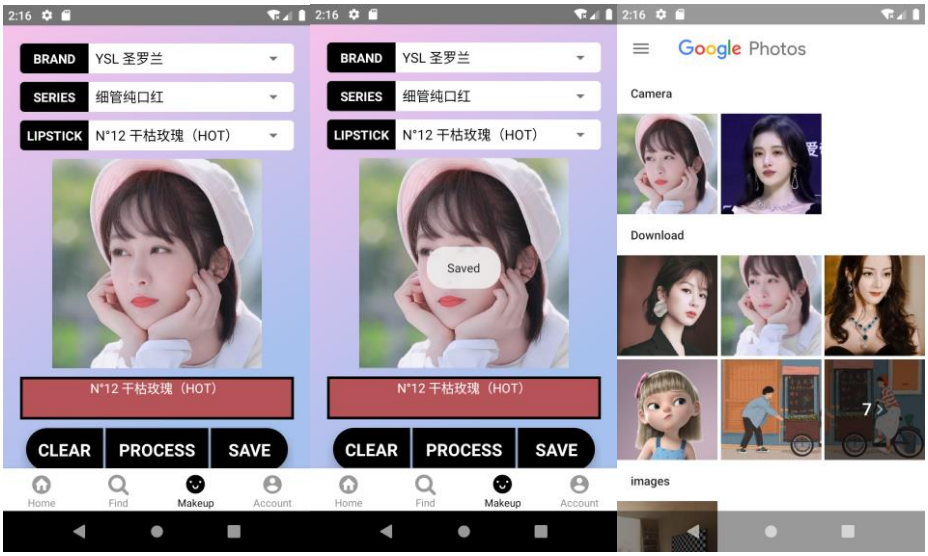

Figure 5-24: a: Change another lipstick, click the "PROCESS" button. b: Click the "SAVE" button. C: Check in album

Click the "CLEAR" button, the test result shows in Figure 5-25-a. We can see that the photo in the center is reset back to the camera icon, and lipstick remaining. We can re-pick a new photo (Figure 5-25-b) indicating the "CLEAR" button works well.

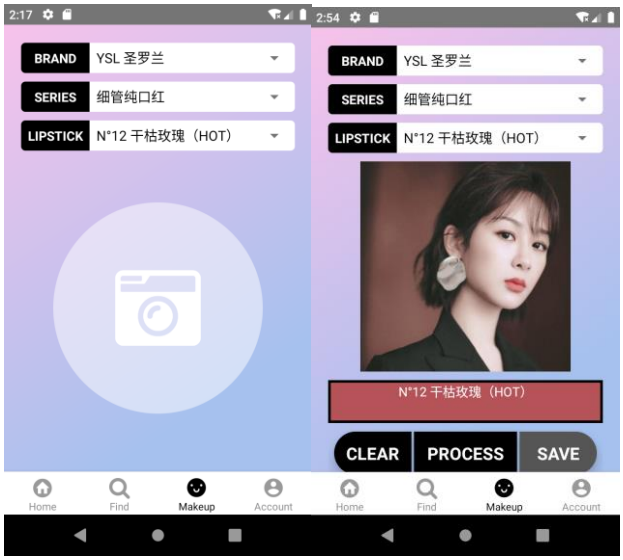

Figure 5-25: a: Click the "CLEAR" button. b: Pick a new image

Switch to previous “Find Screen” and click the lipstick “Giorgio Armami 阿玛尼 #103G 琉金奶茶橘”, then back to “Makeup Screen”, the test result is shown in Figure 5-26-b. There is a new color card of the lipstick just selected. Choose the new color card and click the “PROCESS” button, the test result shows in Figure 5-26-c. It renders with the new color, so the makeup function with selected lipstick in the previous “Find Screen” works well.

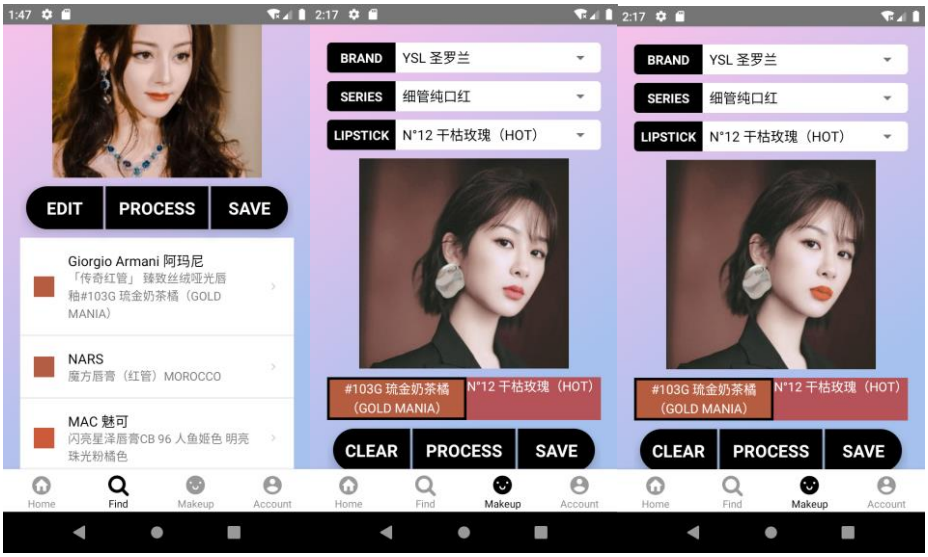

Figure 5-26: Render with previous selected lipstick (a, b, c)

5.2 Algorithm Evaluation

5.2.1 Test Dataset

In order to test the accuracy of our recognition algorithm, we collect 56 labeled lipsticks’ trial images as our test dataset. In order to let the face-parsing model [1] to recognize lip parts, we resize and paste each lipstick trial image to a standard 512×512 resolution human face. Therefore, our final lipstick recognition test dataset consists of 56 standard human faces with labeled lipsticks on their faces (e.g. Figure 5-27).

Figure 5-27: Sample images of the test dataset

5.2.2 Evaluation Result

We adopt removing over-bright and over-dark points with an average RGB color calculation algorithm (RMBD) on encoded color database as our ultimate employed recognition algorithm. Extensive experiments reveal that our recognition algorithm can give plausible recognition results. We compare RMBD with average RGB color calculation algorithm on color swatches database (Baseline) and RMBD without removing over-bright and over-dark points preprocess step. Table 5-1 shows the quantitative comparison results. It demonstrates that RMBD outperforms the baseline method quantitatively and has higher recognition accuracy in a wide range by adopting remove over-bright and over-dark points preprocess steps.

Method	Top-3	Top-5	Top-10	Top-20
Baseline	1.80%	1.80%	1.80%	1.80%
RMBD without preprocess	**39.30%**	**50.00%**	**67.90%**	71.40%
RMBD	**39.30%**	**50.00%**	**67.90%**	**73.20%**

Table 5-1: Quantitative comparison (Top-3 recognized lipstick hit rate, Top-5 recognized lipstick hit rate, Top-10 recognized lipstick hit rate, Top-20 recognized lipstick hit rate. Boldface means better.) RMBD surpasses other methods.

6. Future Work

6.1 Recognition Algorithm

Our work demonstrates the one to one mapping relationship between lipstick swatch color and lipstick actual on-lip color. By updating the color database with our recognized colors, a plausible lipstick recognition accuracy is achieved on our test dataset. However, the encoding rule between the lipstick swatch color and lipstick actual on-lip color remains a mystery. Due to the time constraint, we do not explore this hidden rule. Some machine learning algorithms (even simple regression) can be applied to explore this rule using lipstick swatch dataset and our updated lipstick color dataset, which may be useful to further improve the recognition accuracy.

6.2 Makeup Algorithm

We are capable of generating convincing lip makeup images in a real-time speed. However, the drawback of our makeup algorithm is that we can only generate an even color lips region. In real life, some lipsticks' on-lip effects have shining points, the color is not evenly distributed under this circumstance.

Figure 6-1: The sample images of lipsticks which have shining effect

The possible solution is that we can apply a predesigned shining points mask on this kind of lipsticks. Once the frontend receives the shining lip makeup request, our backend will apply the shining points mask on the makeup result to produce a similar shining effect.

6.3 Recommendation System

As for the database filtering of our recommendation system, the features utilized now are just the kind, texture and color of lipsticks and the popular lipsticks only have 30 lipsticks from popular websites and apps. However, features including `brand` and `price` can also be incorporated into the database filtering system in the future. If we have more time, we can collect more popular lipsticks. Meanwhile, if our application has enough users in the future, we can also give the popular *lipsticks* collection based on the history of users' interaction with lipsticks, which is recorded in our database. The ranking list of lipsticks generated by our application could be more suitable for our users.

User-to-User collaborative filtering algorithm is used in our database. Since the main data used in this algorithm is the rating value and rating value is calculated on the basis of users' behavior now, different mechanisms of calculating the rating, even enabling users to give the rating value directly, could be adopted. However, it still needs enough users giving us feedback to confirm the most efficient one in the future.

7. Conclusion

In this project, we used React Native framework to build a system in the form of a mobile application called Lipstick Finder to recognize the most possible lipsticks for the given profile image efficiently. Additionally, the Lipstick Finder provides lip makeup and daily recommendation service based on the user's preference and behavior.

The lipstick recognition process consists of two parts: extracting a representative color from the lips region and finding lipsticks with the most similar color. The lips region recognition is achieved with a pre-trained face-parsing model using the BiSeNet in PyTorch. We propose a novel RGB color calculation algorithm RMBD which eliminates the influence of uneven illumination on the lips region. To find lipsticks with the closest colors to the extracted color, we sort the weighted Euclidean color distance between lipsticks' colors and target color.

The lip makeup also relies on the face-parsing model. By taking the extracted lips region as a mask, the system overlays target color to the lips region in the HSV color space and replaces the original image's lips with corresponding pixels in the overlaid image.

The lipstick recommendation handles differently for new users and existing users. The system does database filtering for new users based on users' preferences in lipsticks' features (e.g., color scheme, texture and liquidity kind), while providing more delicate recommendations for existing users using collaborative filtering based on users' behavior.

The main contribution of our project is offering the lipstick recognition function because no applications in the market can solve this pain point for users. To raise recognition accuracy, we have made great efforts in four aspects: 1. Select an appropriate model to extract the lips region in real-time speed; 2. Propose a novel algorithm to generate a representative color from lips; 3. Provide photo editing tools in the application for users to adjust the color tone to satisfy users' subjective preferences; 4. Update lipstick swatch colors with the recognized colors to construct a close-to-real-world lipstick color database. Experiments show that these designs give plausible lipstick recognition accuracy.

8. References

[1] L. Zhang, face-parsing.PyTorch, University of Oxford, May 18, 2019. Accessed on: July 31, 2020. [Online]. Available: https://github.com/zllrunning/face-parsing.PyTorch

[2] C. H. Lee, Z. Liu, L. Wu and P. Luo. MaskGAN: Towards Diverse and Interactive Facial Image Manipulation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages. 5549-5558, 2020.

[3] C. Yu, J. Wang, C. Peng, C. Gao, G. Yu and N. Sang. BiSeNet: Bilateral Segmentation Network for Real-time Semantic Segmentation. In *Proceedings of the European conference on computer vision (ECCV)*, pages. 325-341, 2018.

[4] CompuPhase, Colour metric: A low-cost approximation, Informatie-Technologisch Bureau CompuPhase, May 23, 2019. Accessed on: July 31, 2020. [Online]. Available: https://www.compuphase.com/cmetric.htm

[5] V. W. Stefan, Applying a coloured overlay to an image in either PIL or Imagemagik, Berkeley Institute for Data Science, Feb 9, 2012. Accessed on: July 31, 2020. [Online]. Available: https://stackoverflow.com/questions/9193603/applying-a-coloured-overlay-to-an-image-in-either-pil-or-imagemagik/9204506

[6] A. Pathak, C. Mandava and R. Patel, Recommendation Systems: User-based Collaborative Filtering using N Nearest Neighbors, Simon Fraser University, Feb 26, 2019. Accessed on: July 31, 2020. [Online]. Available: https://medium.com/sfu-cspmp/recommendation-systems-user-based-collaborative-filtering-using-n-nearest-neighbors-bf7361dc24e0

[7] React Native Community, React Native: A framework for building native apps using React, Facebook, July 23, 2020. Accessed on: July 31, 2020. [Online]. Available: https://reactnative.dev/

[8] R. Armin, Flask, The Pallets Projects, July 31, 2020. Accessed on: July 31, 2020. [Online]. Available: https://palletsprojects.com/p/flask/

[9] D. Kushal, Introduction to Flask, Oct 10, 2019. Accessed on: July 31, 2020. [Online]. Available: https://pymbook.readthedocs.io/en/latest/flask.html

[10] PyTorch Community, PyTorch, Facebook, July 31, 2020. Accessed on: July 31, 2020. [Online]. Available: https://pytorch.org/

[11] W. L. Zhang, Lipstick color visualization, Apache Echarts, Mar 8, 2018. Accessed on: July 31, 2020. [Online]. Available: https://github.com/Ovilia/lipstick

[12] G. Galushka, react-native-gl-image-filters, Github Community, July 14, 2020. Accessed on: July 31, 2020. [Online]. Available: https://github.com/GregoryNative/ react-native-gl-image-filters

[13] Jultup, rn-fetch-blob, Jultup, Mar 13, 2020. Accessed on: July 31, 2020. [Online]. Available: https://www.npmjs.com/package/rn-fetch-blob

[14] React Native Community, async-storage, Facebook, July 20, 2020. Accessed on: July 31, 2020. [Online]. Available: https://github.com/react-native-community/async- storage

9. Appendix

Installation Process

First, prepare a new AVD, the settings of the simulator are shown as below.

Figure A-1: AVD Settings

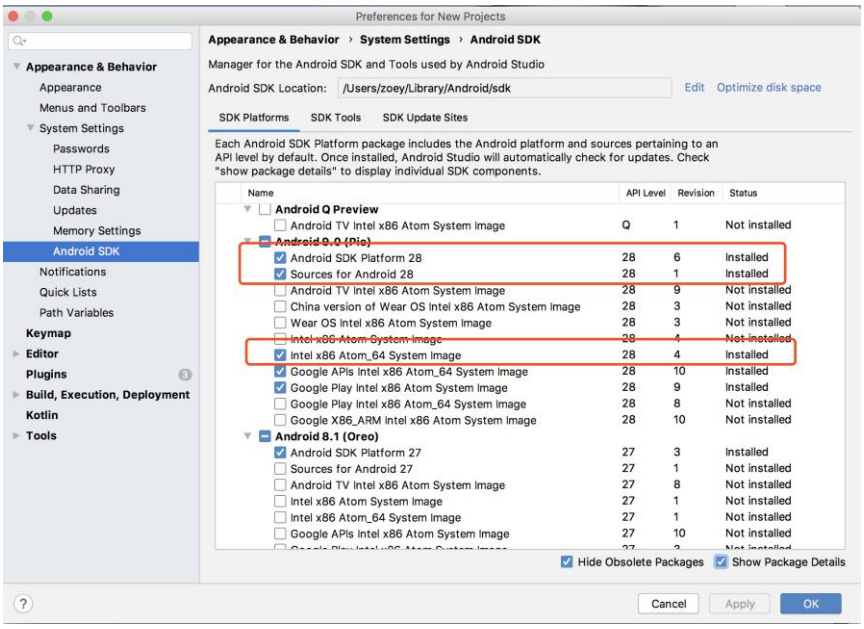

Figure A-2: SDK Platform Settings

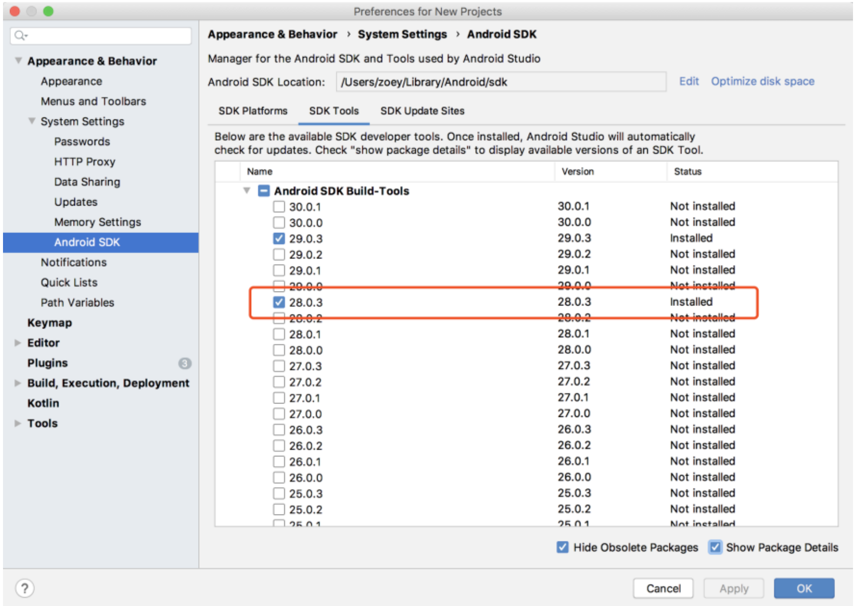

Figure A-3: SDK Tools Settings for SDK

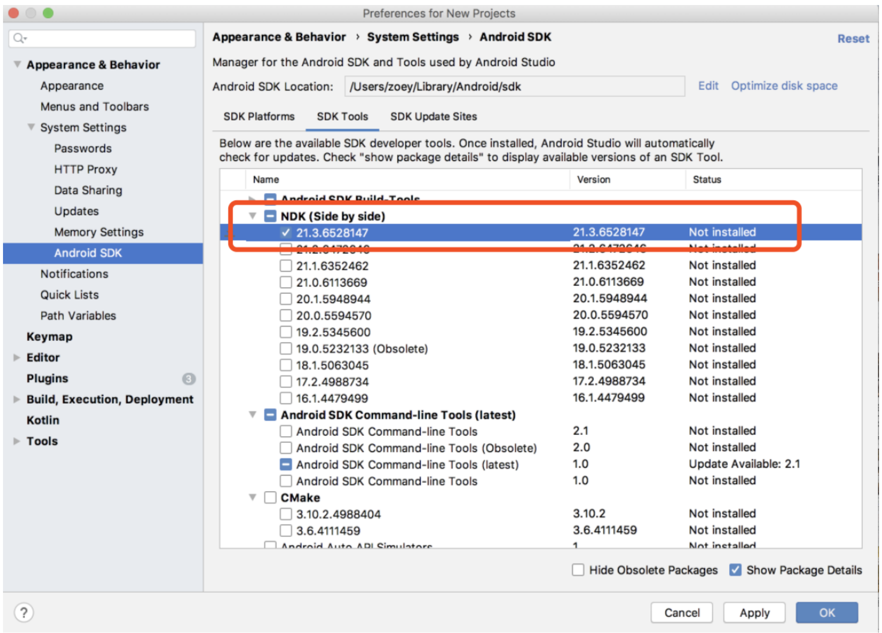

Figure A-4: SDK Tools Settings for NDK

After downloading the source code, enter the folder *LipstickFinder*, and use "*npm install*" to install packages.

```
ZoeydeMacBook-Pro:LipstickFinder zoey$ npm install
(                  ) : diffTrees: sill install generateActionsToTake
```

Command A-1: Install command

Then run the application with the command "*npx react-native run-android*".

```
ZoeydeMacBook-Pro:LipstickFinder zoey$ npx react-native run-android
info Running jetifier to migrate libraries to AndroidX. You can disable it using "--no-jetifier" f
lag.
Jetifier found 1567 file(s) to forward-jetify. Using 4 workers...
info JS server already running.
info Installing the app...
<-------------> 2% CONFIGURING [5s]
> :app
```

Command A-2: Run command

Building it successfully will be like this figure.

```
BUILD SUCCESSFUL in 1m 33s
691 actionable tasks: 667 executed, 24 up-to-date
info Connecting to the development server...
info Starting the app on "emulator-5554"...
Starting: Intent { cmp=com.lipstickfinder/.MainActivity }
```

Figure A-5: Successfully build