CS461 - Mobile & Pervasive Computing and Applications

Semester: AY 2020-2021 Term 2

Section: G1 Team 2

Project Name: CUTN Grocery Shop App

Team Members:
Alwyn Ong (01336812)
Giles Chang (01339320)
Gladwin Neo (01335414)
Shaun Phua (01359422)

1. Introduction

This project aims to develop an online grocery application that is catered towards all age groups, especially amongst the older generations. We aim to utilise Machine Learning to develop a predictive method to allow consumers to easily search for products. We also aim to introduce a rewards feature that encourages our users to adopt a healthy lifestyle by giving discount codes for use in our shop when they clock in a target number of steps.

2. Project Motivations

2.1 To provide a seamless online grocery shopping experience for online consumers

Due to the COVID-19 pandemic, online shopping sector has reach an all-time peak. CNA has also reported that online grocery shopping is the fastest growing slice, with Alibaba's Redmart dominating the scene. As the pandemic persists, more elderly are encouraged to use online grocery platforms for their necessities. Navigating through the online platform and searching for groceries might be daunting for the elderly who may not be tech-savvy. For the younger generations, it is also sometimes difficult to search for certain products as they do not or have forgotten the name of the items they are looking for.

Existing grocery applications rely on conventional search methods like text search. Furthermore, these interfaces are often clunky and troublesome. To address these problems, we aim to provide a seamless online shopping journey, by creating an intuitive interface to easily search for the groceries using voice and camera.

2.2 To promote and reward healthy lifestyles

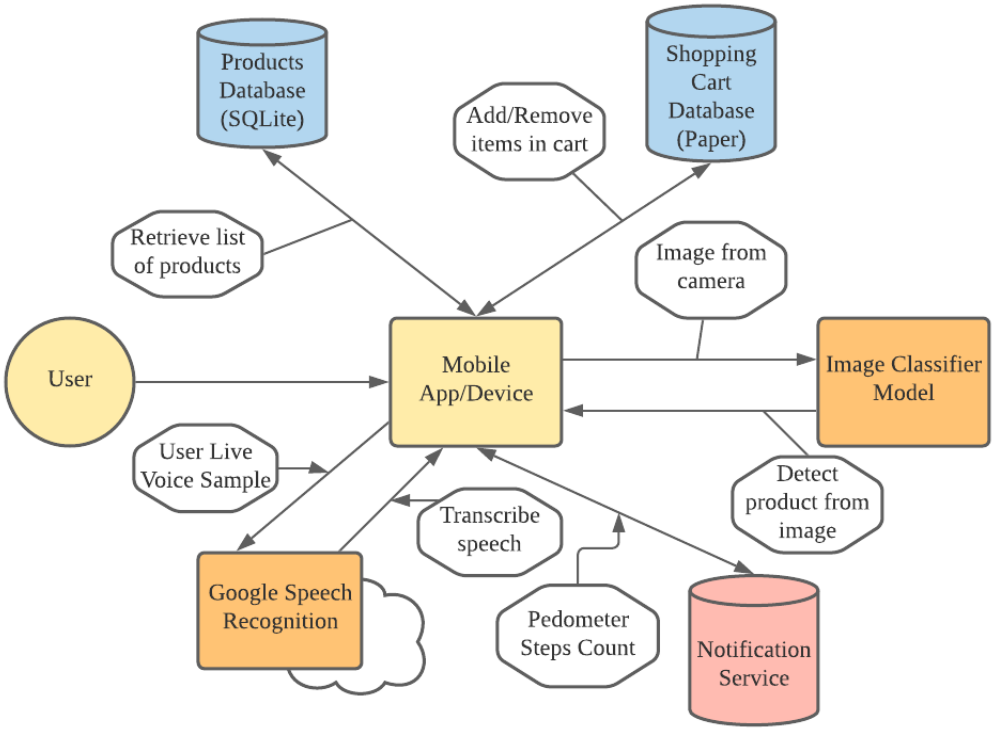
We want to promote healthy lifestyles for our users by gamifying the rewards redemption process. To inculcate this vision with our application, we aim to reward our consumers by allowing them to exchange daily steps for discount codes in our shop.

3. Implementation Details

3.1 Development Environment

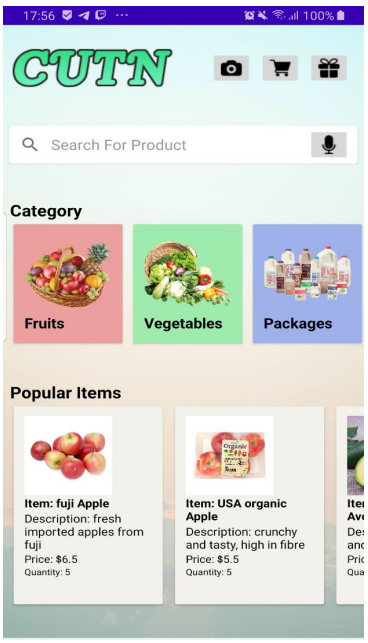
Our application is developed in Kotlin using Android Studio. Since we're working with many external libraries and also sensors like the pedometer, we realise that Sdk version of 30 works the best. The minimum Sdk is currently set to version 21. Java virtual machine target version is currently 1.8.

3.2 System Design Overview

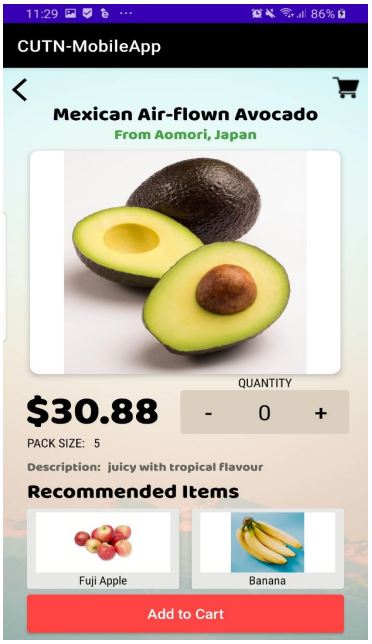


As the user navigates through the application, data is retrieved from SQLite and Paper databases. Two machine learning models are used for our application, GoogleSpeechRecognition is cloud-based, while the image classifier is deployed locally. The pedometer for our step-counter runs in the background, and the notification service will broadcast whenever the step-count target is achieved.

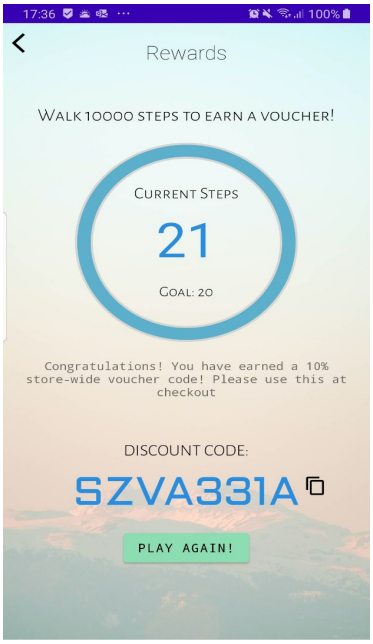
3.3 Layout Design



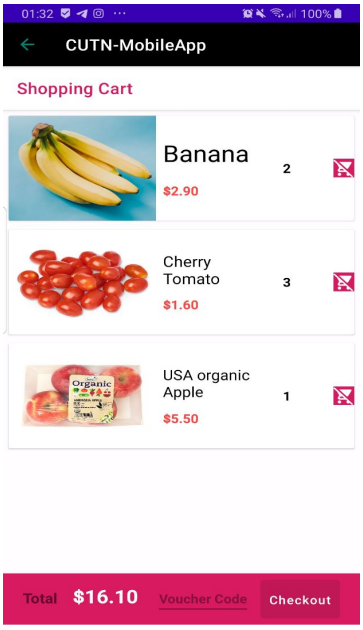
(Figure 1.1 Homepage)



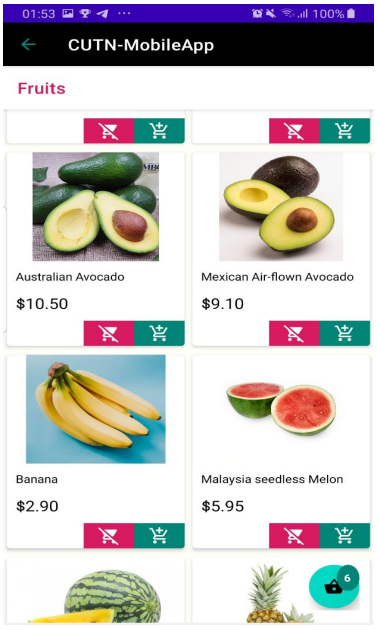
(Figure 1.2 Individual Product Page)



(Figure 1.3 Rewards Page)



(Figure 1.4 Cart)



(Figure 1.5 All Product Page)

(Figure 1.6 Classifier Page)

3.4 Design Explanations

3.4.1 Home Page - Figure 1.1

For our home page, we used Android's RecyclerView to render our category as well as the list of popular items we have in our application. Android's RecyclerView is especially useful in cases for displaying large datasets as it is a dynamic list which will be rendered as the application runs.

3.4.2 Rewards Page & Individual Product Page - Figure 1.2 and Figure 1.3

RelativeLayout is used here. Attributes like *"alignParentTop"* and "layout_below" help to align widgets by referencing to surrounding widgets, making it dynamic across all screen sizes.

3.4.3 Cart Page & All Products Page - Figure 1.4 and Figure 1.5

RecyclerView is used to render the list of products by search or category. The data is dynamically retrieved from the SQLite database. Upon adding an item to the cart, the cart will also be updated via Paper, dynamically updating the number of items shown in the floating cart button.

3.4.5 Classifier Page - Figure 1.6

The classifier page was built upon a relative layout and RecyclerView. Relative layout aligns the imageview and the cards below. The recycler view is used to render the list of products for the cards.

3.5 Database Design

We leveraged the SQLite database for basic CRUD operations. It is automatically embedded into the application upon onCreate. It is lightweight and requires little configuration, making it suitable for our project. The database stores the entity "Product" which is populated manually.

We utilised the Paper library for the shopping cart for its simplicity in storage. It allows us to store a mutable list of products accessible within our app.

3.6 Application Functionalities

Users can search for products via image recognition, voice recognition, or text search. Users can also view products by category as well as popular items. The shopping cart tracks your items added to your cart and allows for checkout. Users will receive a unique discount code for use during checkout when they complete the target number of steps. The pedometer runs in the background, and notifications will be sent to the user if the target has been reached.

3.7 Sensors & Permissions Requirements

Permissions (as indicated in AndroidManifest.xml):
1) ACTIVITY_RECOGNITION - Pedometer Sensor (Rewards)
2) CAMERA - Camera Sensor (Image Search)
3) RECORD_AUDIO & INTERNET - Google Voice Recognition (Voice Search). Internet permission is compulsory to send voice input to their server for recognition.

3.8 External Libraries used

In place of the default toast, we have used MotionToast for more aesthetic toasts. These toasts are also used in different variations and themes, depending on the situation for which the toasts are called.

To update the shopping cart's badge icon for the number of items in the shopping cart as items are added, we have utilised RxKotlin for better reactive widgets. Various libraries like Confetti, interactive widgets, and reactive search bars were also used.

4. Advanced Concepts Used

4.1 Voice Recognition

Voice recognition is initiated through intent with google's API of ACTION_RECOGNISE_SPEECH. This API streams audio input to google server, where the speech recognition model will run. We utilised the RecognizerIntent class to specify parameters required for voice detection. RecognizerIntent.EXTRA_LANGUAGE_MODEL indicates what speech model to use. Based on the speech model, the speech recogniser fine tunes the result. Currently, we opted for LANGUAGE_MODEL_FREE_FORM which considers voice input as "free form English". EXTRA_LANGUAGE indicates language used, and we specified it as the default device's language. EXTRA_PROMPT is the placeholder text that users will see, when the voice recognition button is pressed.

4.2 Image Recognition Model

4.2.1 Model Overview

Our image recognition model takes in an image of a grocery item that users take with their phone camera and our model will predict and return the name of the item. The recognition model is built using TensorFlow Lite Model Maker which simplifies the process of adapting and converting a TensorFlow neural-network to classify common grocery products. We utilised transfer learning with a pretrained model (EfficientNet-Light0) and trained the classifier model with our own input data.

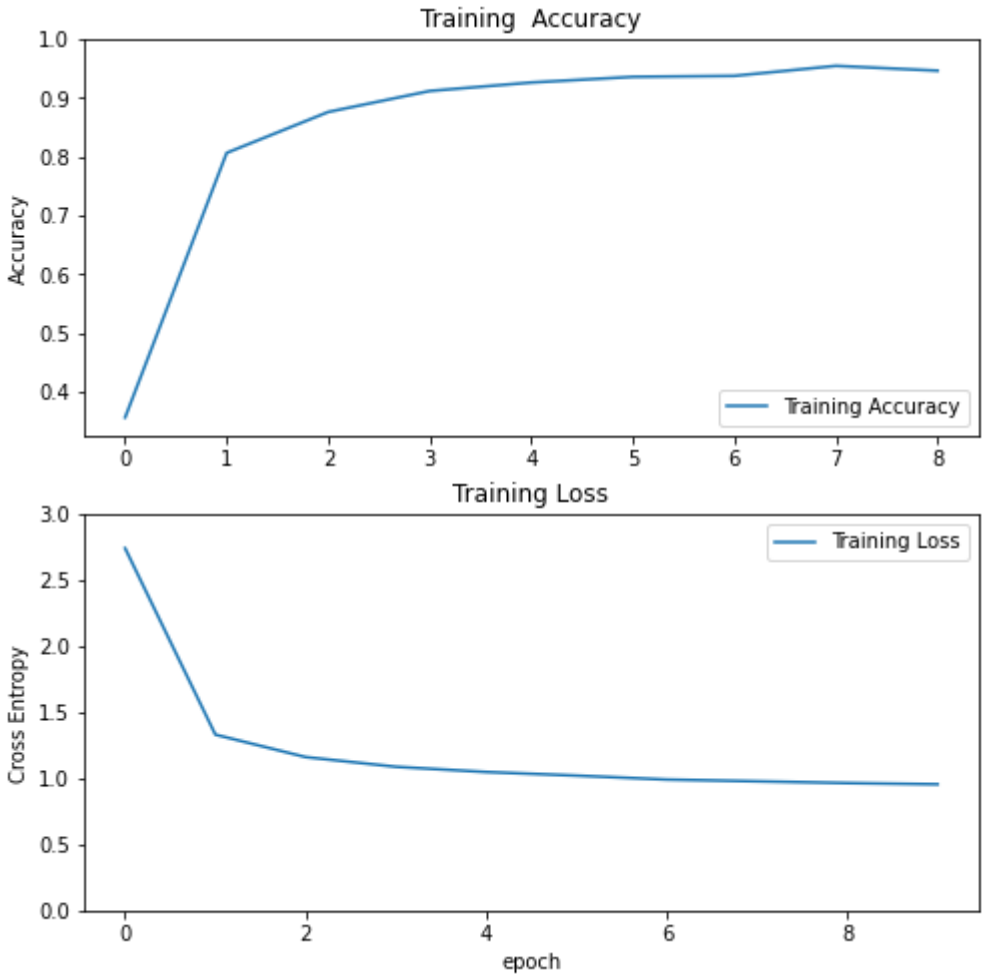
4.2.1 Training

Our input data consist of 3938 images over 35 categories of a mixture of open-source grocery dataset and pictures of groceries which our team took at the markets. (.ipynb will be submitted together with the report)

We trained our model using the following parameters:

Layer types: First Layer: Hub Keras Layer Second Layer: Dropout Layer Last Layer: Dense Output Layer	Hyperparameters: Epochs: 10, Dropout rate = 0.2 Batch size = 32 Validation data split = 0.2 Shuffle = true

4.2.2 Accuracy and Metrics



(Figure 4 Accuracy and Cross Entropy Loss during Training)

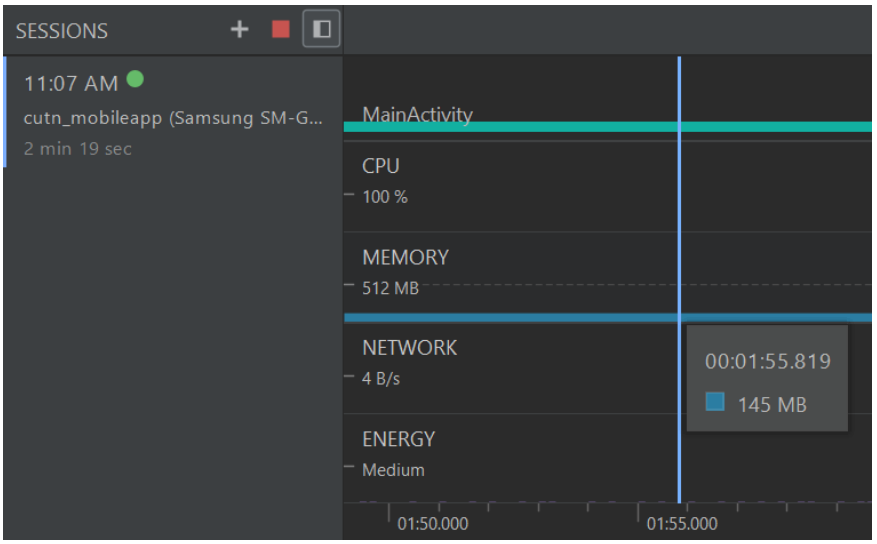
Results after training and evaluation (Figure 4):
Cross-entropy loss: 95.5%
Accuracy: 94.6%

We were satisfied with the model accuracy and we proceeded to export the model to a Tensorflow Lite model so that it can be deployed for on-device machine learning applications.

5. Comprehensive Analysis and Resource Usages

5.1 Resource utilisation with Android Profiler

This performance testing was performed on S8+ featuring octa-core chip with 4GB Ram. Background applications and Bluetooth/Wi-Fi are turned off. In this section, we'll be primarily focusing on Memory usage, as that is the main bulk of our resource consumption.

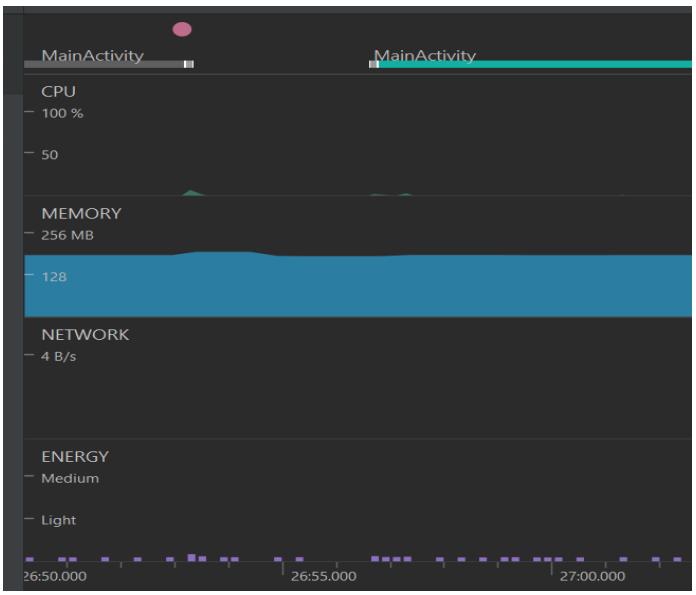


(Figure 5.1 Profiler at rest)

When resting on the homepage, energy consumption is extremely low, and CPU utilisation rate is 0-1%. Memory consumption consistently takes up 145MB.

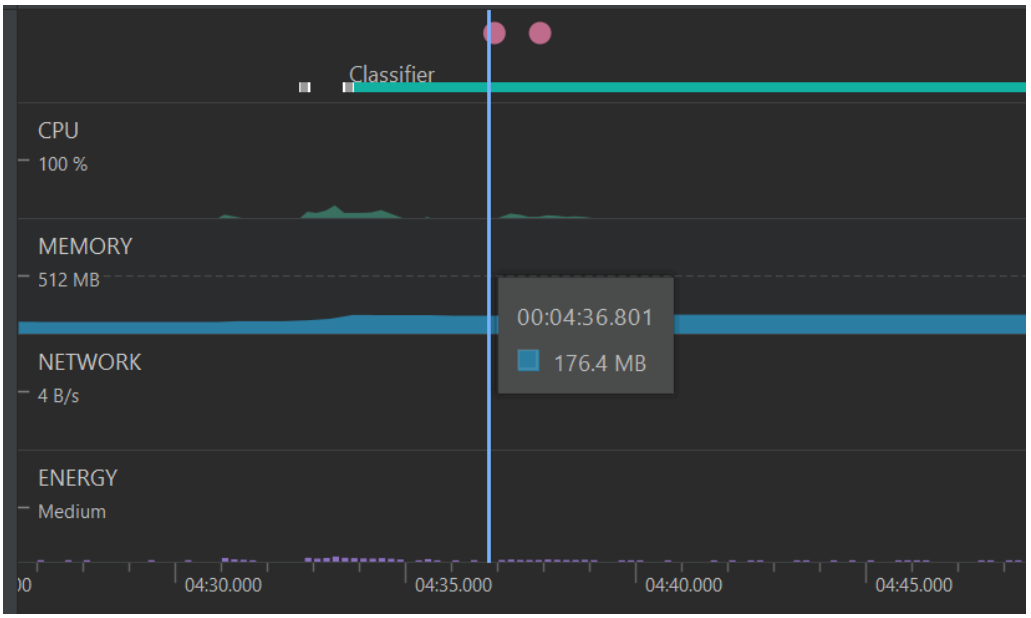
(Figure 5.2 Profile when navigating through activities)

Navigating from activities will require 1-2% of CPU usage



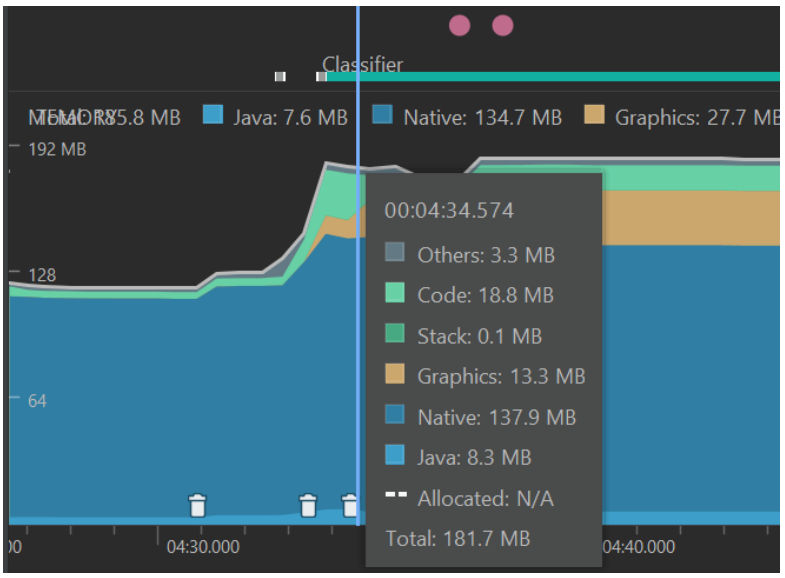
(Figure 5.3 voice recognition activity profile)

Upon voice recognition in MainActivity, metrics remained relatively low, CPU utilisation is low at 1-2%, as the speech model is based on google servers hosted on cloud.



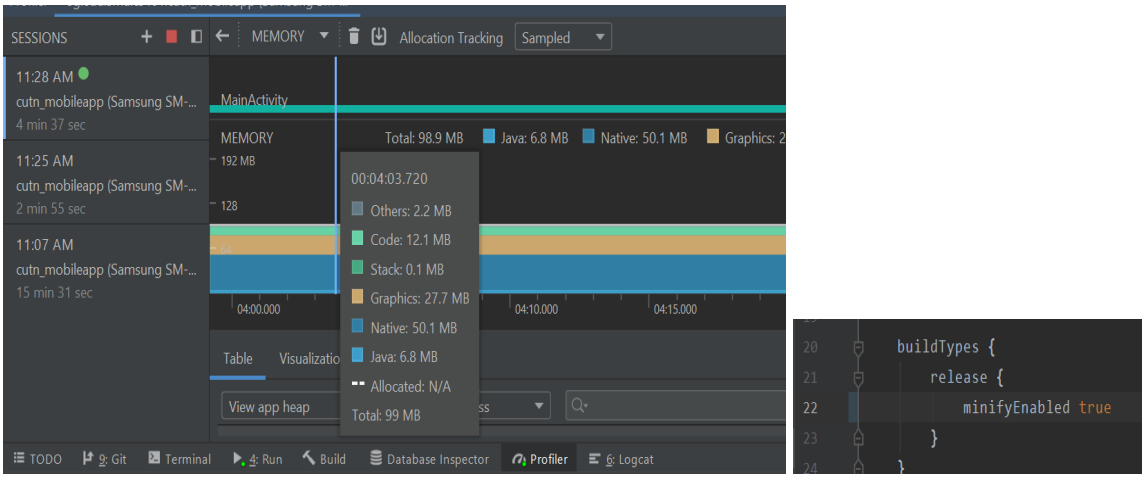
(Figure 5.4 Image recognition profile)

Upon Image recognition, memory usage has increased to 170-180MB. Battery and network consumption remains low. CPU utilisation has hit a peak of 20%. Previous attempts returned a 35% CPU utilisation. To improve the CPU utilisation rate, we refactored and optimized our code by only calling the image recognition only when images uploaded by users are successfully converted to a Bitmap, and closing the model immediately after it returns a result.



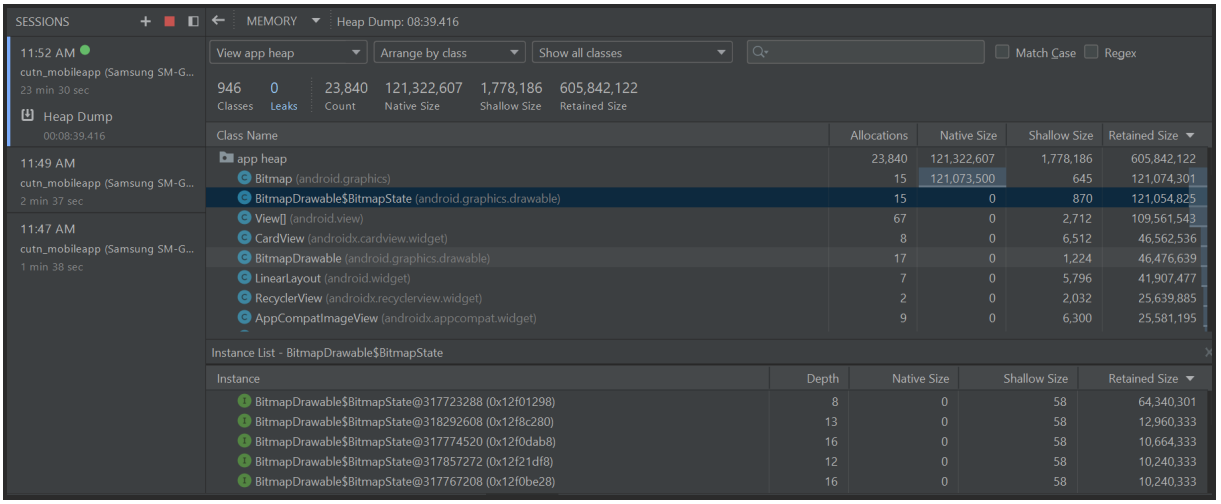
(Figure 5.5 Image classification in depth memory allocation)

As observed from Memory Profiler, the majority of memory is allocated to Native (Memory from objects allocated from C or C++ code.) Despite code written in Kotlin, Android framework still uses this memory to perform various tasks for our application like rendering image assets and other graphics. This could be due to the large number of images hosted in our application.



(Figure 5.6 Improvements to memory allocation)

To further improve memory usage, we specified minifyEnabled = true in build.gradle. This enables “Code shrinking” where the R8 compiler can help to determine any library dependencies that are not required at runtime. R8 will automatically remove any code that is considered unreachable. As observed in figure 5.6, native memory allocation has dropped to 50.1 MB and total memory allocation has dropped by ~50%. However, a limitation is that this result is inconsistent, and sometimes R8 will not be able to remove unreachable codes.



(Figure 5.7 Original Memory allocation for Bitmap - Before changes)

Performing a forced garbage collection and a heap dump displays that the majority of memory retained is due to Bitmap generated during image classification. To address these problems, we have tried to implement several methods.

Firstly, upon the camera sensor capturing an image, instead of passing the entire bitmap as a parcelable in the bundle, it is instead stored in internal storage (Figure 5.8).

```
    val mediaFile: File
    val mImageName = "groceryItem.jpg"
    mediaFile = File( pathname: mediaStorageDir.p
    return mediaFile


rivate fun storeImage(image: Bitmap) {
    val pictureFile = getOutputMediaFile()
    if (pictureFile == null) {
        Log.d( tag: "TAG",
             msg: "Error creating media file, ch
        return
    }
    try {
        val fos = FileOutputStream(pictureFile)
        image.compress(Bitmap.CompressFormat.PN
        fos.close()
```

```
private fun analyzeWithClassifier(ctx: Context) {
    val imgFile = File( pathname: getDir( name: "groceryPhoto", MO
    var bitmap: Bitmap? = null
    var userImage = findViewById<ImageView>(R.id.userImage)
    if (imgFile.exists()) {
        bitmap = BitmapFactory.decodeFile(imgFile.absolutePath)
        userImage.setImageBitmap(bitmap)
    }
```

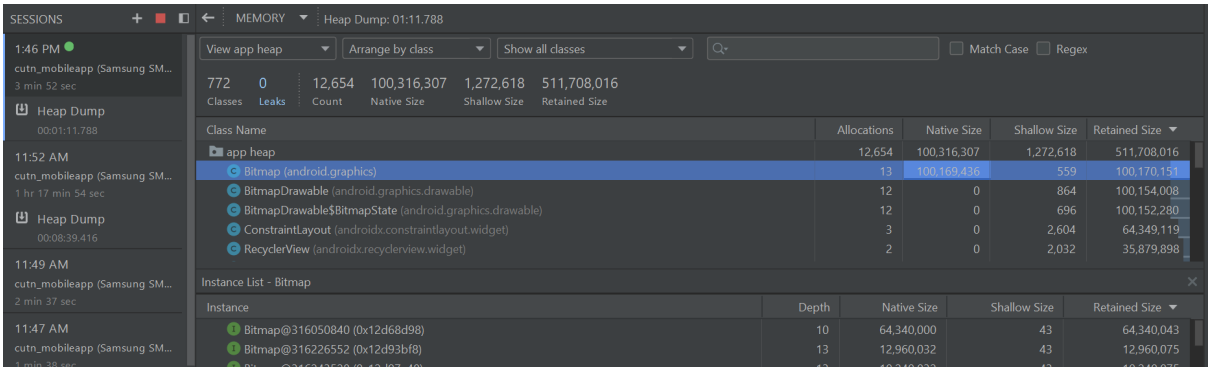
(Figure 5.8 Creating internal directory) ***(Figure 5.9 Retrieving from internal directory)***

Subsequently, the file is retrieved from the internal storage directory (Figure 5.9) and decoded to retrieve the bitmap, preventing the need to pass the memory-intensive bitmap via Intent.

```
var userImage = findViewById<ImageView>(R.id.userImage)
val width = userImage.drawable.intrinsicWidth
val height = userImage.drawable.intrinsicHeight
if (imgFile.exists()) {
    userImage.setImageBitmap(
        decodeSampledBitmapFromResource(imgFile.absolutePath, width, height)
    )
```

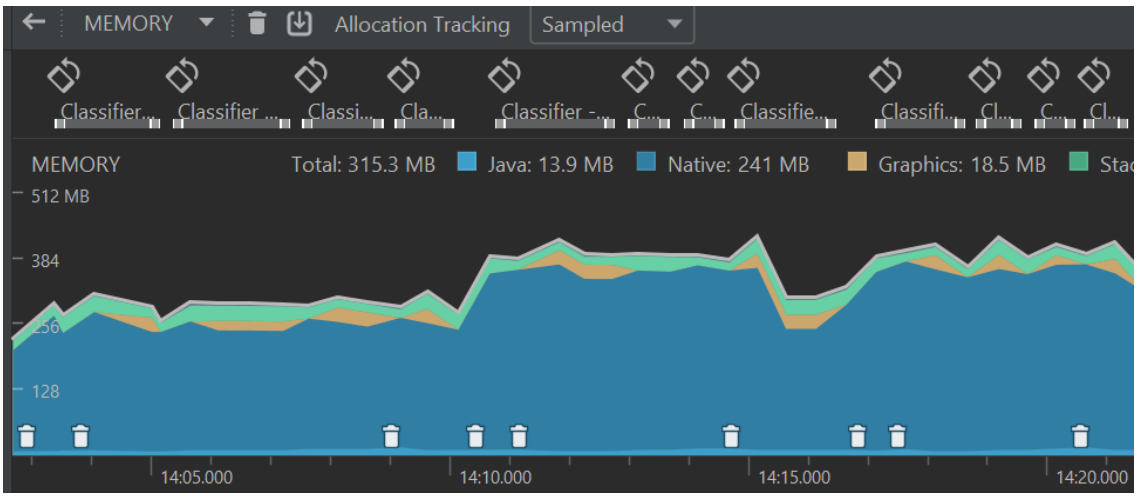
(Figure 6.0 decode Bitmap dynamically based on imageView thumbnail dimensions)

Secondly, to load large bitmaps more efficiently into imageView, we leveraged on BitmapFactory SampleSize. The sample size is the number of pixels that correspond to a single pixel in the decoded bitmap. By passing sample size in bitmapFactory options, it requests the decoder to subsample the original image accordingly. In our scenario, this ensures that the returned image is smaller, saving more memory. This is useful for our application as the thumbnail for the imageView in activity_classifier is small, thus loading in a large bitmap will waste resources. As shown in Figure 6.0, the thumbnail dimensions are passed into our function, which dynamically helps to decode bitmap to a suitable size. However, a limitation of this is that since our model is referencing the decoded bitmap, the predicted result might be compromised due to a drop in image quality. To achieve a balance, the optimised bitmap is loaded into ImageView thumbnail, while a less compressed version is sent to our model for image recognition. Therefore upon rendering ImageView, resource utilisation will be reduced, while maintaining an acceptable predictive score.



(Figure 6.1 New Memory allocation for Bitmap - After Change)

With the changes made earlier, we find that the total native/retained size for bitmap has made some improvements.



(Figure 6.2 Memory spike during orientation change)

Lastly, while the device rotates from portrait to landscape multiple times, as shown in the diagram, there is a high chance of memory leak. This happens when our system recreates an Activity every time orientation is changed, and object garbage collection is hindered by re-creation. We added configChanges="orientation|screenSize" to androidmanifest.xml to prevent this from happening.

5.2 Battery Utilization

To preserve battery life, we opted for google's voice recognition model. The ACTION_RECOGNISE_SPEECH API streams audio to google's remote servers to perform speech recognition. Offloading the recognition process to the cloud made our application more lightweight and less memory consuming. Nevertheless, bandwidth costs will certainly increase. Other open-source speech-recognition toolkits like PocketSphinx, where recognition model dependencies are directly installed on the application were also considered. However, since our image recognition model is already locally deployed, we believe that 2 machine learning models might be too taxing on resource consumption.

For sensors like pedometer, SENSOR_DELAY_UI (60,000 microsecond delay) was also specified when sensor manager registerlistener is called. This delay is the desired delay between events in microseconds. Using a larger delay as compared to the default delay will impose a lower load on the processor and therefore leads to lower battery consumption in the long run. For the step-tracker, it's not required to display updates in near real-time, therefore we opted for a more battery-preserving delay.

5.3 Further Optimisation Techniques

RecyclerView is used to display products and categories instead of a ListView. The next batches of products will only be shown in the RecyclerView if the previous products are out of the screen making it a dynamic list, which is more computationally optimal as compared to using a ListView (static list).

6. Limitations of Application and Future Works

Currently, our application only supports one language, English. In the future, we intend to include more languages by specifying it in locale directories within the res folder. Additionally, we did not include any third party payment merchants to handle the payment gateway when users check out from our application. At the moment, the checkout process is simulated.

Our application will not work with lower SDK versions, as older phones lack the sensors (e.g. Pedometer) required to run the application. For various screens, the layout rendered might also appear different, as development was done separately. This application works best with Samsung S8/S9/S10/S21, as development and testing was based on these devices.

Lastly, our image recognition dataset is not fully comprehensive and lacks many other categories. Currently, products with distinct features work the best (milk, carrots, apples, etc). In the future, as we collect more data and introduce more categories, we can fine-tune the model so that we can achieve higher accuracy rates and let our users enjoy a wide range of products.

Word Count: 2122 (excluding cover page, headers and captions)