SEARCHING AND CLASSIFYING MOBILE APPLICATION SCREENSHOTS

by

ADIS KOVACEVIC

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science at The University of Texas at Arlington Dec, 2016

Arlington, Texas

Supervising Committee:

Christoph Csallner, Supervising Professor
Christopher McMurrough
David Levine

Acknowledgements

I want to express my gratitude and appreciation for my supervising professor, Dr. Christoph Csallner for his guidance on this thesis. I did not have the pleasure of meeting Dr. Csallner during most of my academic career, but he still graciously accepted my invitation to be my supervising Professor. The opportunity, advice, and guidance he offered throughout has been invaluable.

I would like to give a special thanks to Dr. Christopher McMurrough and Mr. David Levine for serving as my committee members. They both graciously accepted and attended my defense with short notice and an aggressive schedule.

I would like to thank all my friends who kept my spirits high. When times were hard, they knew just how to keep me motivated and moving forward.

Finally, I would like thank my mother and father for the support and encouragement they offered me throughout my academic and professional career. They have made tremendous sacrifices to give me this opportunity, I do not take it for granted.

Nov 18, 2016

Table of Contents

List of Illustrations

Abstract

SEARCHING AND CLASSIFYING MOBILE APPLICATION SCREENSHOTS

The University of Texas at Arlington, 2016

Adis Kovacevic, MS

The University of Texas at Arlington, 2016

Supervising Professor: Christoph Csallner

Searching for a particular application layout image is a challenging task. No search provider gives an adequate method to filter the query results to the look of a mobile application. Searching for a particular style of application requires lots of manual time sifting through the results returned. Search engines such as Google are too broad and return too many unrelated results without providing sufficient filters on things like category or layout type.

This paper proposes a technique that would allow the searching and classifying of mobile application screenshots based on the layout of the content, the category of the application, and the text in the image. It was originally conceived to support REMAUI (Reverse Engineering Mobile Application User Interfaces), an active research project headed up by Dr. Csallner. REMAUI has the ability to automatically reverse engineer the User Interface layer of an application by being given input Images.

The long term goal of this work is to create a full search framework for any UI image. In this paper, we introduced the first steps to this framework by focusing on mobile UI screenshots. We discuss 3 techniques to classify the layout of the image, Block Analysis, Interval Encoding, and Bag of Visual Words. We continue on to discuss

a method to classify the category of the application based on the text in the image. Finally, we put all the information together in a single REST API. The API can search input images by the image content and filter by type and layout. The results are ranked by Solr for relevance and returned as json by the API.

Chapter 1 Introduction

Searching and indexing UI images is a complex problem not truly solved. A common technique is to tag the pictures with different text and search the tags, returning the associated pictures. UX layouts come in a variety of flavors and are hard to categorize. A single screenshot could contain multiple different layout types (card layout, grid layout, list layout, form), or even a brand new type of layout which would not match any of the previous layout classifications. Current Search engines are either too broad or do not offer sufficient filtering of the results to be useful.

Categorizing an application by the text content of the screenshot is also challenging. The text content provided in the image could be very minimal. Using just the few available words, we attempt to classify the category of the application, for example under Music, Fitness & Health, Education, etc.

We discuss 3 techniques for classifying the layout of an input image into known categories, Block Analysis, Interval Encoding, and Bag of Visual Words. We then evaluate the performance of the various techniques.

To the best of our knowledge, this problem has not been solved but similar problems, like classifying documents by content and type have been discussed in previous papers. On the surface, classifying document layouts into different types, such as single column letter, 2-column letter, single column journal, etc., is a similar problem to classifying the layout of an application. The effectiveness of various techniques however varies greatly as the layout of an application has a much higher mix of graphics and text as compared to a document.

The Interval Encoding document classifier [5] is one such technique. We use a modified version of this technique on our Interval Encoding classifier for mobile screenshots. We see similar results on screenshots as this technique saw on documents.

Finally, we provide a mechanism to search the content of a Mobile Application screenshot, filtering on the Category and Layout of the application. This is the groundwork for the Search Framework. It is a RESTful API powered by Apache Solr and MySQL backbends.

The search framework was originally conceived to support REMAUI (Reverse Engineering Mobile Application User interfaces) [1], an active research project headed up by Dr. Csallner. REMAUI has the ability to generate the UI portion of an Android project based on a screenshot of the Layout. REMAUI offers additional useful data, such as the XML that generated the front end design, metadata, and Java class files, we provide a REMAUI approach were applicable to the Search Framework. In the REMAUI approach we discuss additional data the Search framework has to work with and expose that is not available in the general case, which only has a single UI image. This could lead to better classifications in the future, such as one that is able to convert a layout XML into the type of layout shown.

We evaluate our classifiers using 120 mobile application screenshots taken from the google play store. For our 3 classifiers, Block Analysis has an accuracy of 89.47%, Interval Encoding has an accuracy of 73.8%, and Bag of Visual Words has an accuracy of 65.48%, as compared to the ground truth. The Category Classifier has an accuracy of 91.67% as compared to the ground truth.

Chapter 2 Background

This section contains the necessary background information on Computer Vision, OCR, Apache Solr, and the REMAUI project, and mobile application layouts.

2.1 Apache Solr Search

Solr is a standalone enterprise search server with a REST-like API [11]. It includes advanced text search capabilities including phonetic, phrases, geospatial, wildcards and grouping. For our needs, we can index and do a “fuzzy” search on the text content of the images. The “fuzzy” or phonetic search allows for typos and words to be spelled how they sound, rather than a strict spelling. This is helpful as the extraction of text from images is not perfect. We extract all of the text from an image and load it to Solr as a single document. Synonyms can be added for common words to improve the results. There are numerous sources and dictionaries that provide these synonmys for a variety of Languages. Solr is the primary driver of our search platform. It indexes all our data and makes it searchable through our API.

Solr’s built in phonetic search can use one of 9 different methods, these are also commonly seen is spell checkers.

1. Beider-Morse Phonetic Matching (BMPM)
2. Daitch-Mokotoff Soundex
3. Double Metaphone
4. Metaphone
5. Soundex
6. Refined Soundex
7. Caverphone
8. Kölner Phonetik a.k.a. Cologne Phonetic
9. NYSIIS

As an example, here are the steps to the Metaphone technique:

- Drop duplicate adjacent letters, except for C.
- If the word begins with 'KN', 'GN', 'PN', 'AE', 'WR', drop the first letter.
- Drop 'B' if after 'M' at the end of the word.
- 'C' transforms to 'X' if followed by 'IA' or 'H' (unless in latter case, it is part of '-SCH-', in which case it transforms to 'K'). 'C' transforms to 'S' if followed by 'I', 'E', or 'Y'. Otherwise, 'C' transforms to 'K'.
- 'D' transforms to 'J' if followed by 'GE', 'GY', or 'GI'. Otherwise, 'D' transforms to 'T'.
- Drop 'G' if followed by 'H' and 'H' is not at the end or before a vowel. Drop 'G' if followed by 'N' or 'NED' and is at the end.
- 'G' transforms to 'J' if before 'I', 'E', or 'Y', and it is not in 'GG'. Otherwise, 'G' transforms to 'K'.
- Drop 'H' if after vowel and not before a vowel.
- 'CK' transforms to 'K'.
- 'PH' transforms to 'F'.
- 'Q' transforms to 'K'.
- 'S' transforms to 'X' if followed by 'H', 'IO', or 'IA'.
- 'T' transforms to 'X' if followed by 'IA' or 'IO'. 'TH' transforms to '0'. Drop 'T' if followed by 'CH'.
- 'V' transforms to 'F'.
- 'WH' transforms to 'W' if at the beginning. Drop 'W' if not followed by a vowel.
- 'X' transforms to 'S' if at the beginning. Otherwise, 'X' transforms to 'KS'.
- Drop 'Y' if not followed by a vowel.
- 'Z' transforms to 'S'.
- Drop all vowels unless it is the beginning

All these are abstracted away inside Solr. The end user of Solr just needs to configure the Solr environment and choose one of the 9 methods.

2.2 Optical Character Recognition (OCR)

Text can be distinguished from images and extracted using existing OCR tools [7]. These tools perform relatively well on documents. The text that is extracted can be fed to our Solr server to become searchable. REMAUI uses OCR to extract all its text. The text it extracts is fed into Solr as a document. We perform OCR in the general case where we only have a single screenshot. In the REMAUI case, we have a more robust

technique already implemented by REMAUI [1]. OCR has the problem of erroneously finding text in pictures and grouping things on separate lines into a word.

2.3 REMAUI – Reverse Engineering Mobile Application User Interfaces

When developing the user interface code of a mobile application, a gap exists between the digital conceptual drawings of graphic artists and the final working user interface code [1]. A developer, or team of developers, have to manually take the concept drawings from the UX artists and convert them to the UI code of the application that is being created. REMAUI is the first technique for inferring the user interface code from the provided Image or concept drawing [1]. The current prototype of REMAUI takes an image and generates the UI portion of the Android application. Figure 1 shows the high level steps REMAUI performs to convert an image into a full Android application. To do this, it uses optical character recognition (OCR) and various computer vision techniques. In doing so, REMAUI creates many intermediate files which can be used to search and index the images. Once the UI View Hierarchy is exported (3rd image in Figure 1), the metadata can be sent to the search framework to make this project searchable.

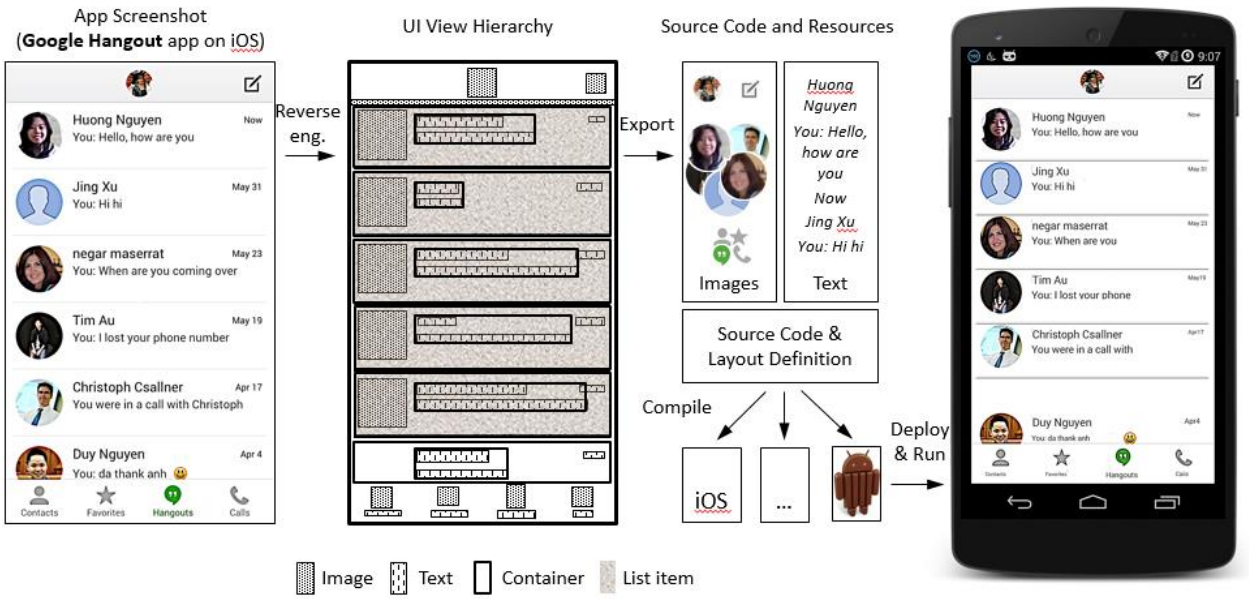


Figure 1 the high-level steps to REMAUI [1]

2.4 Mobile Application Layouts

Mobile applications come in a variety of different layouts, and to try and classify them all would lead to a large, unusable set, so we classify them into more general buckets. Grid Layout, Card Layout, List Layout, Forms, and Splash Screens. Though many apps look different, they are just small tweaks of the 5 categories. Figure 2 shows an example of a Card Layout, Grid Layout, and List Layout. A Card Layout is similar to a List but takes up much more real-estate on the screen per item. To be clear when we talk about a layout of an application, we are talking about the largest overall structure on screen, not the code behind organization, like android's ListView or GridView, but rather what the final appearance is like on the screen. This appearance is what the end user sees in his search, so it is more important than the actual code structure that created it.

Most screens have navigation buttons, headers, and footers. These are not considered in our definition of a layout. A Grid Layout, such as the one in Figure 2 is considered a Grid regardless of the 5 buttons on the bottom or the “Explore” banner on top.

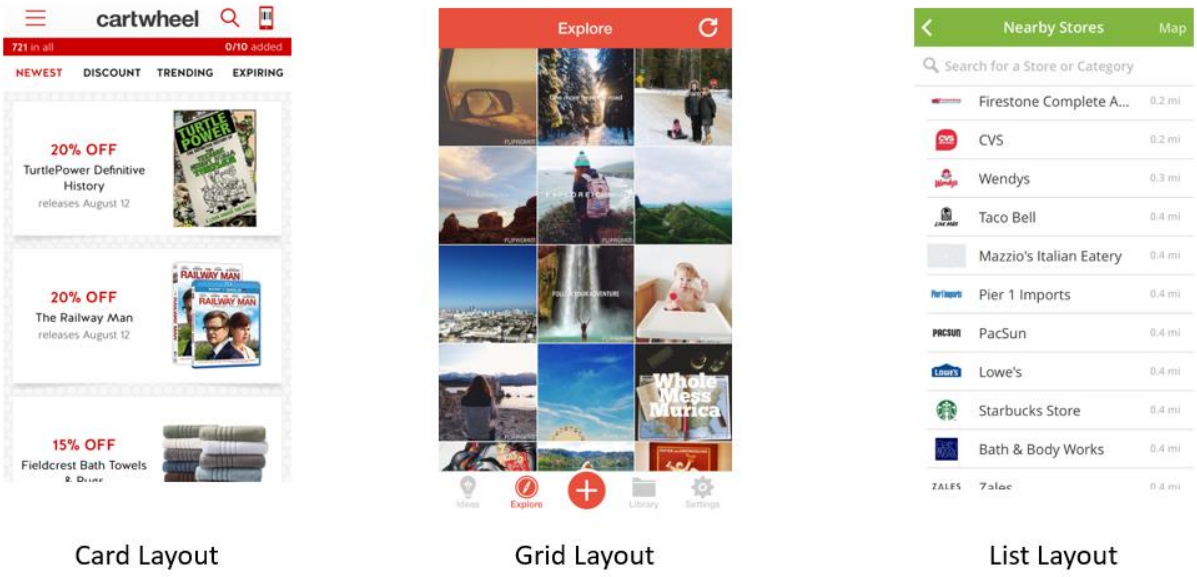


Figure 2 The 3 different layout types

Chapter 3 Image Layout Classification

Our Framework is able to classify the layout of a screenshot into 3 categories, Card Layout, Grid Layout, and List Layout. Figure 2 shows an example of a Card Layout, Grid Layout, and List Layout. A Card Layout is similar to a List but takes up much more real-estate on the screen per item. Those which do not fit in these 3 categories will remain unclassified or classified into a new bucket called “Unknown”. In this paper we describe 3 layout classifying techniques. Block Analysis, Interval Encoding, and Bag of Visual Words.

3.1 Layout Classifier Input

As input to our classifier, we assume the image has been segmented into rectangular white blocks for the content and a black background, we will refer to this image as the layout image from here on out. Creating a layout image involves many techniques in OpenCV [5], usually involving Gaussian blur, Canny edge detection [9], Hough Line Transform, and various blob detection techniques [5]. Figure 3 shows the some intermediate steps to the desired transformation on an input image. First we blur the image to remove some details, then perform binary thresholding on the image, and dilation on the result. We then find the contours and their respective bounding rectangles. Finally, we fill the outer bounding rectangles with a White color. The less detail exists and the more the content is broken into the correct Rectangular shapes, the better the classification process will perform.

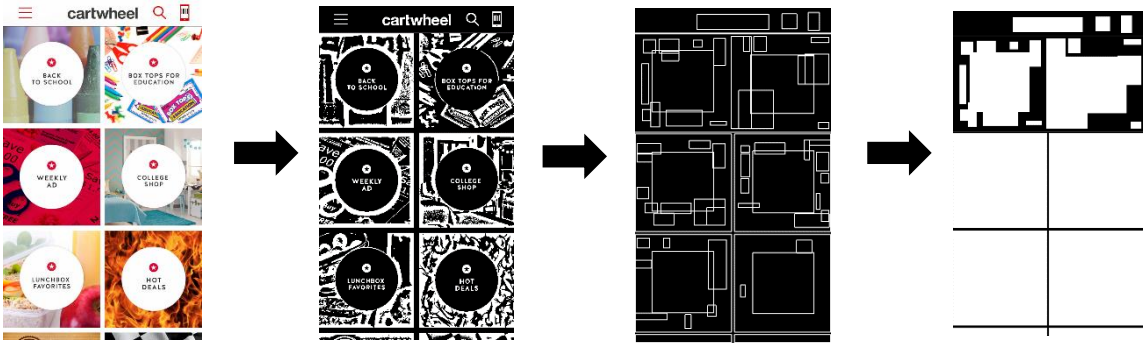


Figure 3 shows the steps to create the input image

The layout image in Figure 3 can be further improved by transforming all blobs to Rectangles and merging very close Rectangles. Indeed the quality of the input image has a huge impact on the accuracy of the classifier. Running it through the input converter a second time could smooth out the 2 into perfect rectangles.

In the REMAUI approach, the intermediate pictures can be used instead of starting with the original image, as REMAUI performs much of the same image manipulations and ends with a more detailed container hierarchy image. For our input image, we use the container hierarchy and transform it into our desired black and white layout image. Figure 4 shows the steps performed by REMAUI to create its Container Hierarchy [1].

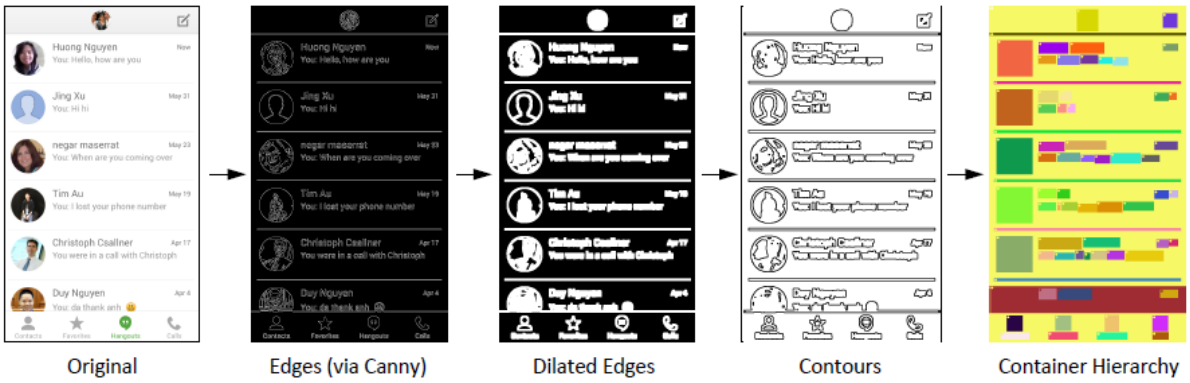


Figure 4 the Computer Vision Process used by REMAUI to generate a Container Hierarchy *[1]* which we use as our layout image

3.2 Block Analysis Classifier

The Block Analysis technique is a method proposed by this paper, and inspired by the technique used to retrieve documents by their layout [3]. It also derives some of its parts from the methods used in REMAUI to Identify Lists [1]. A distinction needs to be made about the list in REMAUI and the List Layout mentioned in this paper. A list in REMAUI is a collection of repeated instances, primarily repeated instances of containers with the same sub-tree structure, and is great for identifying a collection. The List Layout as mentioned in this paper is about the structure of the collection. If the collection is displayed in the XML in the manner seen in Figure 3, it is considered a List Layout. If however it is more similar to Figure 2, we would call the collection a Card Layout. What REMAUI refers to as a list, we will refer to as a collection for clarity.

For the Block Analysis, the input image is broken into blocks. The size, shape, and location of these blocks are analyzed and fitted into one of our 3 classifiers, isGrid, isCard, and isList. Each one checks its criteria and the likelihood that the blocks belong

to that particular layout. For example, Figure 5 shows the blocks generated for the input image of a Grid Layout. A black border is added to all of the input images to ensure none of the blocks flow outside of the contained image. Blocks are checked for their shape. Narrow and wide rectangles are said to belong to the list layout, while more square shapes with adjacent neighbors are said to be a grid. Adjacency is checked in the horizontal direction only. If more than one block exists on the same row, the chance of it being a grid increases. The height of a rectangle is analyzed and a heuristic threshold is picked to differentiate between a card and a list. The height of a list element is said to be no greater than ¼ of the height of the input image. The top and bottom of a mobile application typically contain a banner or some sort of menu selection which is not part of the overall layout. The top and bottom blocks of the input image are ignored to attempt to remove this noise. Heuristically, the top and bottom 10% are cut out of the layout image. Figure 5 shows the numbered blocks of the layout image. Each rectangle in the layout image is numbered and analyzed. The isGrid classifier would say that blocks 1, 2, 3, 4, 5, and 6 all belong to the grid class because of their rectangle shape and at least one neighbor on the horizontal row.

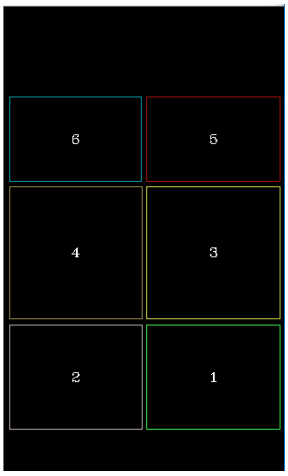


Figure 5 the Blocks of the input image are numbered and shown

3.3 Interval Encoding Classifier

Another technique to classifying the layout of an application is called Interval Encoding [6]. It was originally published describing a method to classify documents based on their layouts. We chose this method because of its high accuracy and ability to classify different types of document layouts, a very similar problem, with hopes for good results on applications layouts as well. We have a modified version to work with our layouts and input images. In interval encoding, the layout image is split into a grid of m by n, and each cell is referred to as a bin [6]. A bin either has content (is mostly white, as the grid lines do not perfectly overlap the white rectangles of the layout image) or has no content (is mostly black). Each row is an array of length n, and contains an integer for each bin. A bin has the value 0 if it has no content, otherwise value of a bin is its distance to the nearest bin with no content on its row. Figure 6 shows the partial interval encoding for the first 4 rows of the image segment to the right of it. In our example here

the bins are perfectly white or black, but partial overlap is possible, and in those cases we decide the bin by the dominant color.

Row1 = {0, 0, 0, 0}

Row2 = {0, 1, 2, 1}

Row3 = {0, 0, 1, 1}

Row4 = {0, 0, 0, 0}

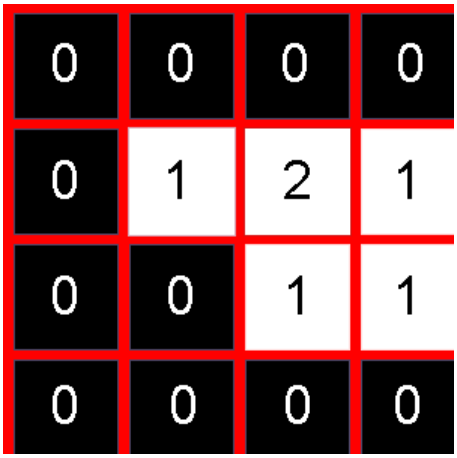


Figure 6 zoomed region of Interval Encoding

The interval encoding contains information about the content of each row. If you compare two rows by taking the absolute value of the difference of the bins, the resulting number shows how far apart the rows are. For example, Visually, Row1 and Row2 are more different than Row2 and Row3 in Figure 6 as Row 1 is all black and 0 for each bin.

|Row1 - Row2| = 4 while |Row2 - Row3| = 2. Row1 and Row4 are identical and indeed |Row1 - Row4| = 0.

To classify a layout image, we compare the interval encodings of the image whose layout is unknown with the interval encodings of images whose layouts are known [6]. The one with the smallest value or “best match” is said to be the layout of the unknown image. In our evaluation, 30 prototype images were created, 10 with List Layouts, 10 with Grid layouts, and 10 with Card Layouts. The interval encoding of the

Test image is compared to the interval encoding of the each prototype image, row by the row. The value of each row is added up:

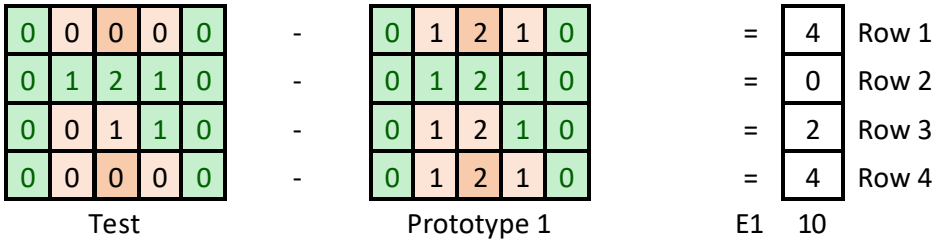
E1 = |Test(Row1) – Protype1(Row1)| + |Test(Row2) – Prototype1(Row2)| + … + |Test(RowN) – Prototype1(RowN)|

E2 = |Test(Row1) – Protype2(Row1)| + |Test(Row2) – Prototype2(Row2)| + … + |Test(RowN) – Prototype2(RowN)|

.
.
.

E30 = |Test(Row1) – Protype30(Row1)| + |Test(Row2) – Prototype30(Row2)| + … + |Test(RowN) – Prototype30(RowN)|

The 30 E values are compared and the lowest is chosen. The Layout of the prototype for that E value is considered to be the Layout of the Test image. The grid below demonstrates this process for a Test image against a Prototype image we deemed to be a List Layout. The value for | Test – Prototype1 | is 10. If this was the lowest of all the prototype images, we would classify Test as a List Layout. In the example below, If the absolute difference of the bins is 0, we color them green, otherwise we color them orange with a darker shade the higher the value.



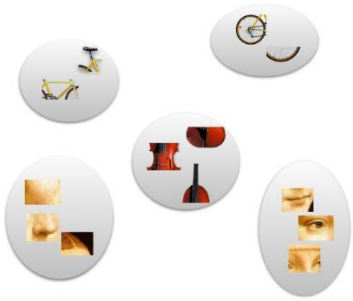
3.4 Bag of Visual Words Classifier

Another technique we use in to classify the layout of the images is the Bag of Visual Words technique [13]. It is a well-known technique for classifying images. It was chosen for its ability to detect different objects and classify them accurately. Its drawback is the inability to pin point the location of the identified object, but that is not a concern for our case. It requires both positive training images and negative training images for each classification, in our case Grid, List, Card, to classify the image with a reasonable amount of accuracy. We represent each training image by a vector (feature descriptor) and use these to learn a visual category model, and then evaluate the Layout image of our choice. In our experiment, we have 3 sets with a total of 30 training images (10 per set). We describe the steps of the bag of visual words below on a common example of a human, cello, and bicycle to help illustrate the method in a visual way.

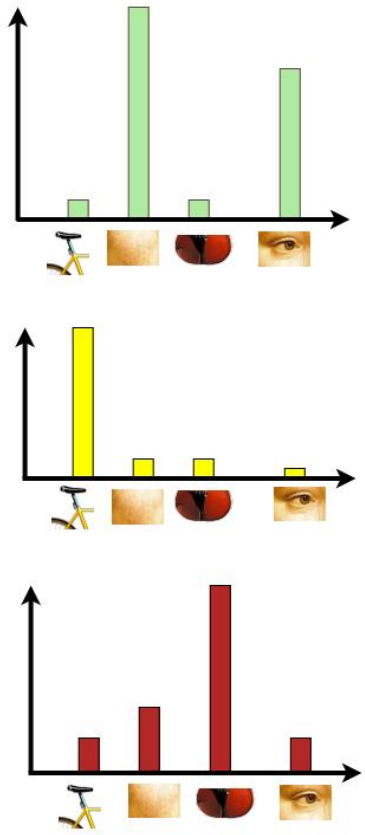
1. We extract useful features from the images [12]

2. We Learn the “Visual Vocabulary” of all the pieces [12]

3. We group features using the visual Vocabulary into visual words [12]



4. Finally, we represent the image by the frequency of these visual words [12]



Chapter 4 Image Category Classification and Search

In the previous chapter we described methods to classify the layout of the mobile application. In this chapter, we discuss the method used to classify the category of the mobile application and query the textual content. The user can perform a general query, and at the same time apply a filter by category for their searches. A search for “work out” with no category filter might lead to results from other categories, while a search for “work out” with a filter set to “Health & Fitness” will assure that no other categories are considered relevant. The user can apply multiple filters and search with multiple categories on the same query.

Applications fall into many different categories including Music, Games, Finance, Business, Education, Health & Fitness, Maps & Navigation, Photography, etc. It is highly beneficial to filter the search by the category of the app. We discuss a method to find the category of the application in the image based on the textual content seen. For our experiment, we will limit our categories to 3. Music & Audio, Health & Fitness, and Food & Beverage.

We have trained a set of classifiers with words associated to the 3 different categories. For example, in the Health & Fitness category, the classifier contains words like “exercise”, “laps” “weight”, “%Fat”, etc. These words were manually entered into the each category by taking common words found in multiple android applications that have been uploaded to the Apple App Store under a specific category. For our evaluations we took the top 30 applications for each of the 3 categories and loaded all of the text from the screenshots provided to the Apple App Store. A more automatic approach

would be to use the OCR output of hundreds of screenshots in a particular category and feed the texts into a classifier. The text could then also be weighted by number of occurrences. Each word of the input text is compared to each word in the 3 categories. If a match is found, we increment the points of that category by 1. After all 3 categories are run, we are left with 3 point values. The highest of the 3 points will decide the category of the image. If for some input text the scores are 5 points for “Health & Fitness”, 2 points for “Music & Audio”, and 1 point for “Food & Beverage”, then the “category” field in Solr will be populated to show “Health & Fitness”, the winner of the 3.

Regardless of category chosen, the full OCR text is passed into Solr and made searchable. If, for example, the OCR Text contained the word “exercise”, than a query for “exercise” will return that image, with a high matching score, for example 10. If an OCR typo has put the word “exxerccise” in Solr instead by mistake, the “fuzzy” phonetic search of Solr will still be able to return the result, albeit with a slightly lower confidence score, for example 7.5.

Chapter 5 Search API, Putting it all together

In this section we discuss the backbone of the search framework, the RESTful API. Is provides the means to add, delete, and search content from our datasets. The API will use the Layout classifier and the Content Classifier on the image it receives, and push all the data to Solr and MySQL. Figure 7 below shows the general workflow of the API. It takes an incoming image and breaks it apart into the Layout image and the OCR Text. The Layout Image is fed into the Layout Classifier, which in this case will return the result "List". The OCR Text is inputted into the Content Classifier, which in this case will return "Music". the Full text , the layout "List", and the category "Music" will be pushed into Solr. The API's communication to Solr and MySQL is described in the next section.

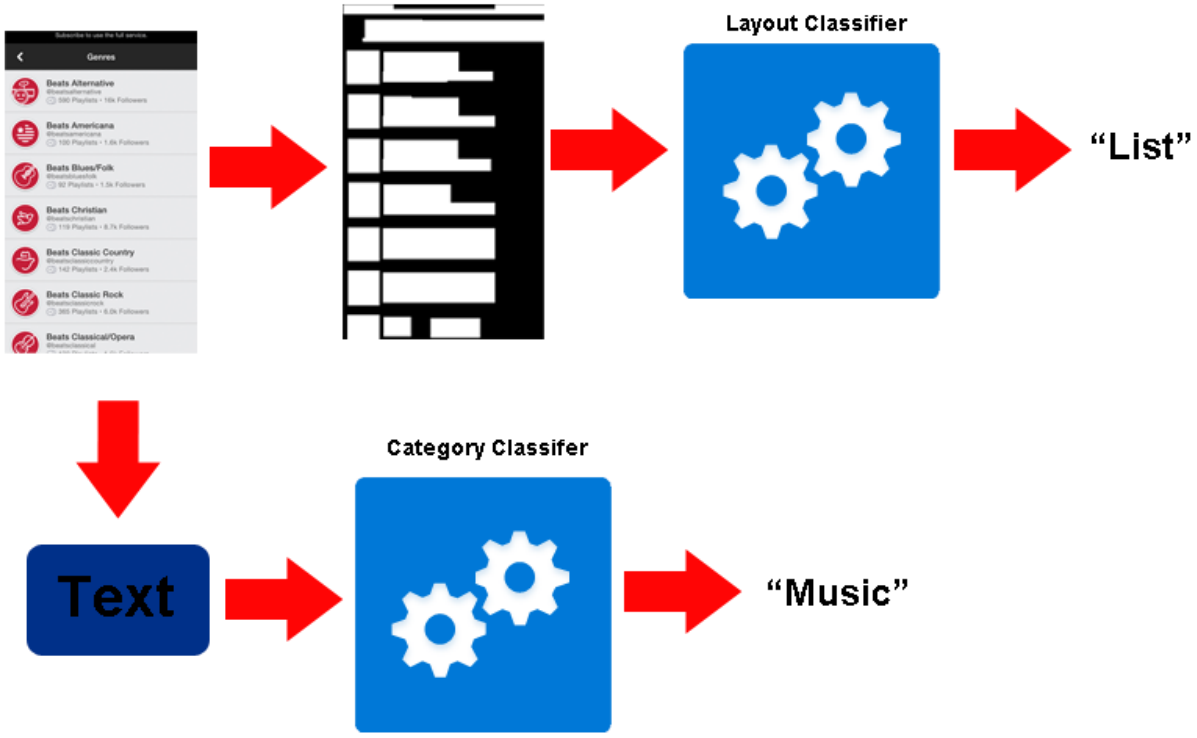


Figure 7 shows the flow of the image through the 2 classifiers and their outputs.

5.1 Components

The Search relies on 3 main components to perform its function.

1. The Search API, which is written in C# Web API. It receives the search requests and returns a list of ranked results with a relevance score.
2. MySQL Db
    a. General case - houses the location of the images. Consists of a Picture Table, which contains an Id, Name, and Location, and Timestamp.
    b. REMAUI case - houses information about the project and the pictures that are associated with it. It consists of 2 tables, the Project table, which consists of an ID, Name, and Location, and Timestamp. The Picture table, which contains an Id, Name, Location, and ProjectId, and Timestamp.
3. Apache Solr, the enterprise search platform that allows texts and phrases to be searched [12]. Solr is loaded with "search items" described in chapter 4, below is the json of the search item in Solr. Each item has a unique id that ties it back to the ProjectId for REMAUI or the ImageId in the General case.
    a. Id: - MySQL Image Id in General case or Project Id in REMAUI case
    OCRText: - Full OCR Text of the image or all relevant text in REMAUI
    XmlText: - REMAUI only - full xml of the layout generated
    XmlNames - REMAUI only- full names of all xml files generated
    PhoneticText: - Same as OCRText
    Layout: - Layout classifier output
    Category: - Category classifier output

5.2 Loading Search

Figure 8 shows the steps required to load the Search Engine through the API. The steps vary depending on if the use case is REMAUI or the GENERAL case.

5.2.1 REMAUI case

1. REMAUI makes a web call to the Search API, passing the created project name, XML documents created, pictures uploaded, intermediate pictures generated, and any Text in the strings.xml file.
2. The project is saved as a new row in the projects table. The table contains the project name, location of the zipped project and a Timestamp. Each of the pictures is saved as a row into the Pictures Table. The Picture row contains the name of the picture, the location of the picture file, a foreign key reference to the associated project Id and a Timestamp.
3. The Id of the Project row is returned to the API layer.
4. The API runs the Layout classifier, and the content classifier on the image. It creates a Solr Search Item and loads the data. The API then sends the item to Solr
5. Solr acknowledges receiving the data

5.2.2 General case

1. API received a Mobile Application Image.
2. Image is uploaded and the name, location of the picture, and Timestamp are sent to MySQL
3. MySQL returns the Id of the picture

4. The API runs OCR, the Layout classifier, and the content classifier on the image. It creates a Solr Search Item and loads the data. The API then sends the item to Solr
5. Solr acknowledges receiving the data

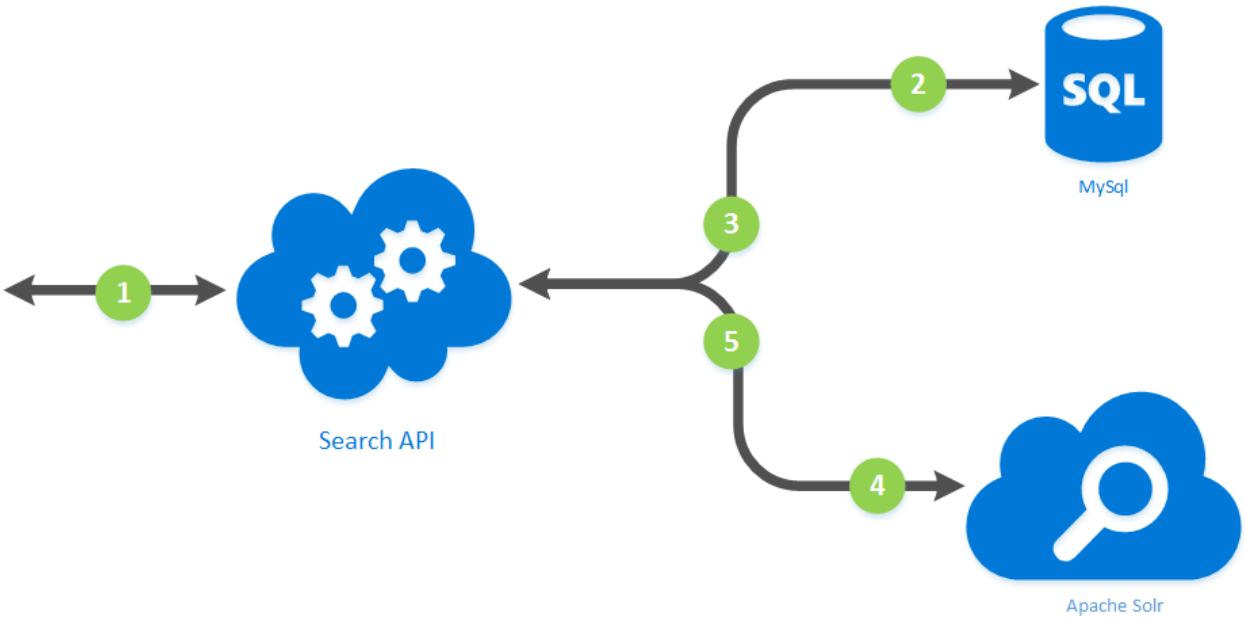


Figure 8 shows the communication between the Search API, Solr, and MySQL database.

For the prototype, in the general case, images are loaded and read one by one from a directory. This file directory was previously loaded with our desired screenshots. In the future, a web crawler could be designed to scrape the images from the google play store and the iOS app store. The REMAUI case uses REMAUI's hosted website to populate the search. When an image is uploaded to REMAUI, the website now passes the relevant data to the Search Framework.

To prevent duplicates, especially in the REMAUI case, which will add all uploaded images to the search, we can use the perceptual Hash of the image to check for a duplicate. The perceptual Hash [10] is widely used to find copyrighted material online. It converts an image to a special hash that can then be compared to other

image's hashes. If another image comes that has the same or similar hash, we can ignore this image as it is a duplicate of one already in the system. Figure 9 shows the perceptual hash of 3 different images. The first 2 are the same image but the second one is the greyscale version. Even without the color, the 2 phash values are very close together.

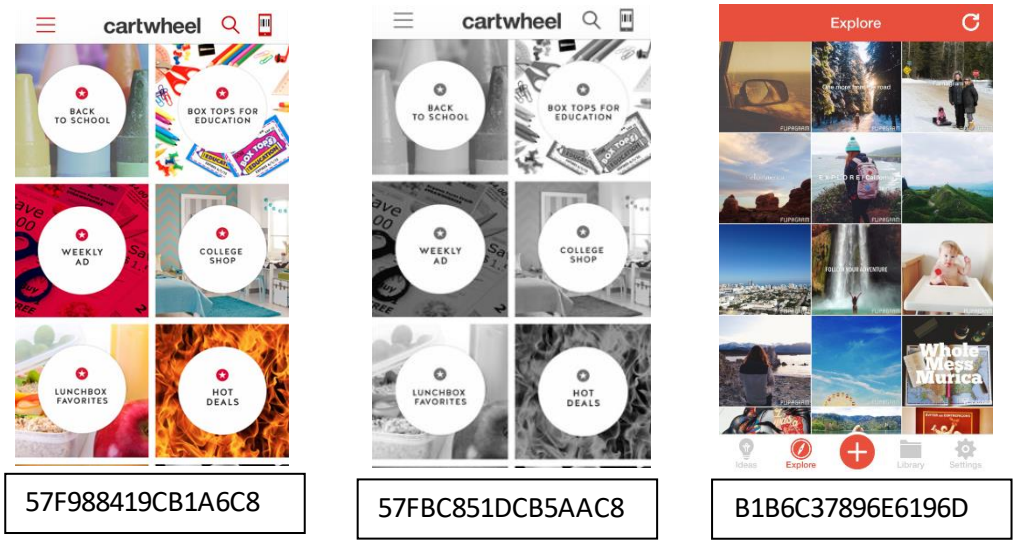


Figure 9 shows the perceptual Hash of 3 images

5.3 Searching and Retrieving Results

The API allows for the user to input a search string, an optional filter for the content type, and an optional filter for the layout category. The result of the search is a list of ranked search items with a relevance score. We created a prototype front-end client to demonstrate the end result of a search query through the Search Framework. Figure 10 is the search result output on the query “health”. It shows the 3 relevant screenshots that were returned.

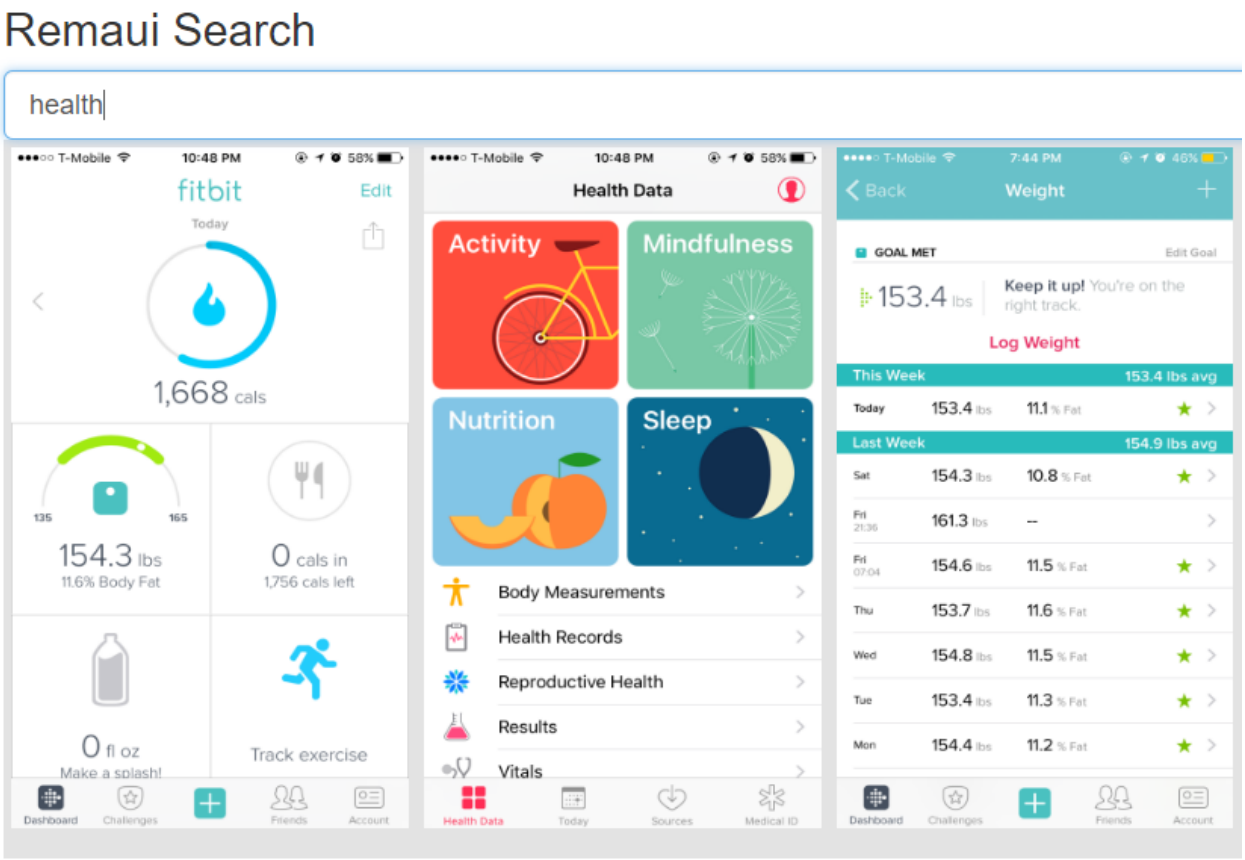


Figure 10 the results of the query "health

Figure 11 is the search result output on the query “health” but also with a layout filter of type “List” applied. Note the filter has reduced the total number of return results by removing 2 of the images from Figure 10, both of which did not contain a List layout. It is worth noting that the middle image is a hybrid image containing both a grid and a list. Because of the limitations of the current classifier, it is classified as a grid.

Figure 11 the search result *for "health" with a Layout filter of "List"*

Figure 12 shows the result difference of a search for the query “country” and the results of a query for “country” with a category of “Music”. The category of Music eliminates the Twitter screenshot leaving only the Spotify one.

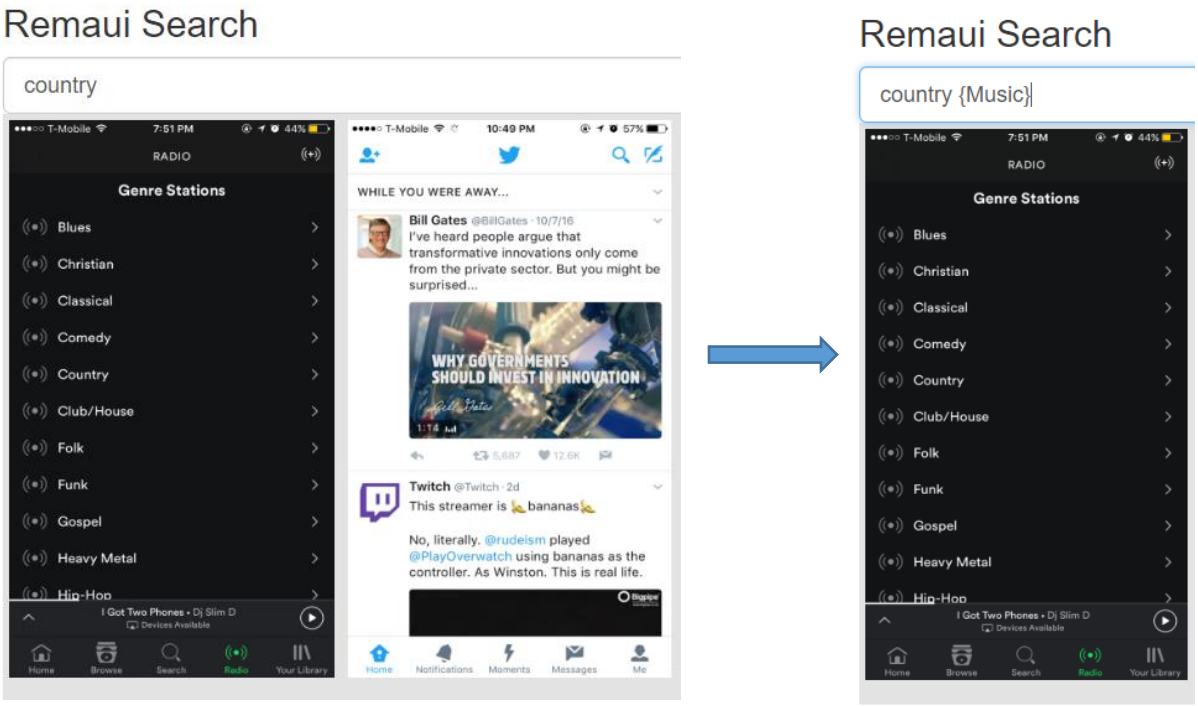


Figure 12 the difference with and without the Category of "Music" on the Query "country"

Figure 13 shows the REMAUI results, which shows the extra layers from the intermediate pictures optionally overlapped onto the original image [1]. This gives information on the location of the TextViews and ImageViews in the project XML. A link to download the full project is also available in the REMAUI version.

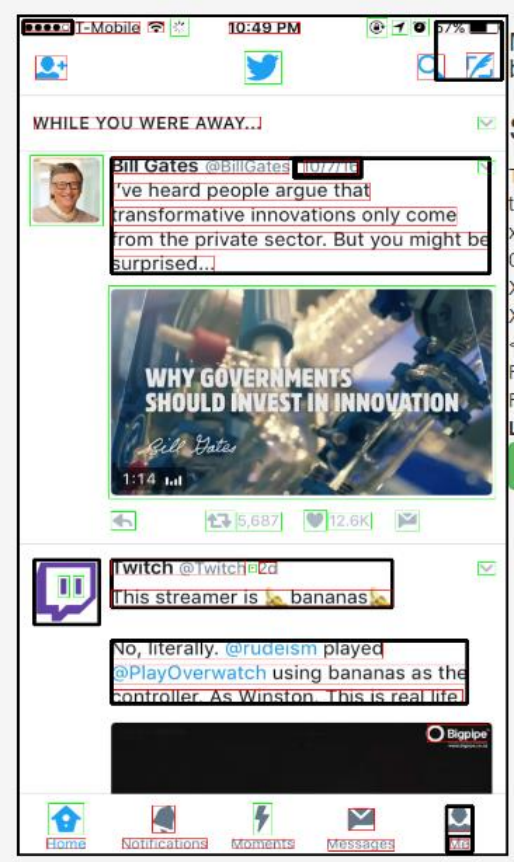


Figure 13 shows the additional information available in the REMAUI version of the Search Framework *[1]*

Chapter 6 Evaluation

In this chapter, we evaluate the different classifiers used by the framework. We evaluate and analyze the 3 Layout classifier and the category classifier using a confusion matrix to get the accuracy, precision, and recall of each.

6.1 Image Layout Classification Evaluation

We evaluation the accuracy, precision, and recall of the 3 different techniques used to classify the layout of the application screenshot. For our evaluation, we assume the input image to be a 500px by 300px image, whose Layout Image quality was controlled to rule out that factor. We have gathered a collection of 84 input images from the google play store. They are the screenshots from the top apps in numerous categories. Each image is a screenshot of a different android application. These will serve as subjects for all 3 evaluations. Of the 84 images, 28 are Grid Layouts, 30 are List Layouts and 26 are Card Layouts. Images with only 1 type of layout classification were chosen. So for example an image with a hybrid layout of a gird on top and a list below will not be included in the evaluation. The 84 images were classified manually by visually observing the image and recording its layout. This will serve as the ground truth for our evaluation. We use a confusion matrix to show the results. We add an additional 30 images with a layout that is not a Grid, List, or Card. These will serve to test the unknown category of the block analysis. Only the block analysis supports unknown layouts. The accuracy, precision, and recall of each Layout classifier is compared to these visual results manually created. Our definition for the accuracy of the entire classifier is calculated as follows:

$$\text{Accuracy} = \frac{\sum(Green\ cell\ values)}{\sum(All\ cell\ values)} \cdot 100$$

It defines the total number of correctly classified images over the total number of images, in our case 84. So for example the accuracy for the Block Analysis Evaluation is:

$$\text{Accuracy} = \frac{27+28+26+0}{84} \cdot 100 = 96.43\%$$

6.1.1 Block Analysis Evaluation

Block Analysis

	Grid	List	Card	Unknown	Total	Precision	Recall	Accuracy
Actual Grid	27	0	0	1	28	90.00%	96.43%	96.43%
Actual List	2	28	0	0	30	100.00%	93.33%	
Actual Card	0	0	26	0	26	100.00%	100.00%	
Actual Unknown	1	0	0	0	1	0.00%	100.00%	
	30	28	26	1	84			

The results for the Block Analysis shows a good overall performance with an accuracy of 96.43%. We show just 3 misclassification of a grid layout. 2 are erroneously classified as Lists and 1 is unknown.

We add the 30 unknown images and rerun the analysis with a total of 114 images.

Block Analysis

	Grid	List	Card	Unknown	Total	Precision	Recall	Accuracy
Actual Grid	27	0	0	1	28	75.00%	96.43%	89.47%
Actual List	2	28	0	0	30	90.32%	93.33%	
Actual Card	0	0	26	0	26	100.00%	100.00%	
Actual Unknown	7	3	0	21	31	95.45%	67.74%	
	30	28	26	22	114			

The recall of the Unknown images is lower than the other 3 types. The precision of the Grid Layout has also lowered as 7 of the 30 Unknown layouts are classified as Grids.

6.1.2 Interval Encoding Evaluation

Interval Encoding

	Grid	List	Card	Total	Precision	Recall	Accuracy
Actual Grid	28	0	0	28	87.50%	100.00%	73.81%
Actual List	0	16	14	30	80.00%	53.33%	
Actual Card	4	4	18	26	56.25%	69.23%	
	32	20	32	84			

The results overall accuracy is low, at 73.81%. While it can distinguish between Grids, it struggles to differentiate Lists and Cards. Of the 30 List images inserted, only 16 came back as lists. The other 5 were classified as Cards. Card had the worst precision at 56.25%. Only 18 of the 26 actual cards were classified as such. The rest being split among Grid and List, 4 and 4.

6.1.3 Bag of Visual Words Evaluation

	BoVW						
	Grid	List	Card	Total	Precision	Recall	Accuracy
Actual Grid	21	4	3	28	84.00%	75.00%	65.48%
Actual List	0	16	14	30	66.67%	53.33%	
Actual Card	4	4	18	26	51.43%	69.23%	
	25	24	35	84			

The results for the BoVW show poor accuracy at only 65.48%. Using a histogram descriptor on a black and white image might not provide enough features for a good classification, especially when considering the similarity of the card and list layouts. Though the initial results look low, the BoVW still remains an option that needs further investigation. The choice of feature descriptor might be playing a critical role.

6.2. Content Classification Evaluation

We now evaluate the Content classifier. We have a total of 120 test screenshots in 3 different categories. These were directly pulled from the google play store and belong the actual category we are attempting to classify. We grabbed the top 30 Food & Beverages apps, the top 30 Health & Fitness apps, and the top 30 Music & Audio Apps. For the 30 images for the Unknown category, we grabbed the top apps of the other categories. The category under the app store will serve as the ground truth for this evaluation. We use the same accuracy formula as section 6.1.

	Content Classifier							
	Food & Beverage	Health& Fitness	Music& Audio	Unknown	Total	Precision	Recall	Accuracy
Actual Food & Beverage	25	0	0	5	30	100.00%	83.33%	91.67%
Actual Health & Fitness	0	27	0	3	30	93.10%	90.00%	
Actual Music & Audio	0	0	30	0	30	100.00%	100.00%	
Actual Unknown	0	2	0	28	30	77.78%	93.33%	
	25	29	30	36	120			

The classifier performs really well with an overall accuracy of 91.67%. The classifier performs the best against the Music & Audio category, identifying all 30 of the screenshots in that category. There is no overlap, or misclassification of one category as another. It is either correctly classified or its category is unknown. A total of 8 were unknown. After some investigation, the result is attributed to typos from the OCR process. 2 of the 30 Unknown were erroneously classified as Health & Fitness.

Chapter 7 Related Work

Much of the related work is from document classification and layout analysis. Classifying document layouts into different types, such as single column letter, 2 column letter, journal, etc. is a similar problem to classifying the layout of an application. The effectiveness of various techniques however various greatly as the layout of an application has a much higher mix of graphics and text.

One of the techniques we use for classifying the layout is Interval Encoding [5]. It's a technique taken directly from a method to identify the layout of a document. We see similar effectiveness in our implementation as the original paper saw in theirs. Our implementation struggles to differentiate Lists from Cards, much like the original paper struggles to differentiate single column letters from single column journals. Indeed, the horizontal similarity of the two, in both cases, causes this difficulty in distinction.

Another technique for document layout classification is the "Evaluation of Distance Measures" [13] which focuses on the shape, width, and area of the different blocks in an image. It's the inspiration to the block analysis technique in this paper. Our analysis of the blocks focuses on matching them to the known desired shapes of a List, Grid, or Card, instead of to another document. A similar technique proposed in another paper focuses on the shape of the whitespace instead of the content [10], looking for long vertical and horizontal rectangles without content to distinguish between columns and other breaks in the document.

Classifying the content of the screenshot is similar to the problem of classifying the content of a document, although a document contains much more context than a

single screenshot, thus the same techniques have to be evaluated and tested. For example the tf-idf (term frequency- inverse document frequency) technique looks at the number of times words appear in a document weighted by the number of times it appears, but also inversely weighted by the number of documents that also contains this word. This helps eliminate common words like “the”. Due to the few words in a single screenshot, this technique does not perform as well as simply looking for key words.

Classifying and categorizing content using the Bag of Visual Words technique has been tested, experimented and evaluated for many different purposes, such as to identify different scenes and classify them [9]. We chose to put the flexibility of this technique to the test in our own evaluation to compare the different types of layouts. Though not directly related to our research, the novel method inspired to use of the Bag of Visual Words on our layout images.

Chapter 8 Conclusion and Future Work

In this paper, we have discussed a Search Framework that allows a Mobile Application UI Image to be searchable and classified by the layout and category. The framework consist of 3 major pieces with various implementations which were analyzed and discussed. We showed the accuracy of the Block analyzer when it comes to classifying layouts, as compared to 2 other methods. It outperformed the other methods in every metric, accuracy, recall, and precision. We also discussed the General Framework as well as a REMAUI version, for which the framework got its start.

Many mobile application screenshots contain hybrid images, such as the second image in Figure 10. The current Layout classifiers will choose one or the other layout (Grid or List) in this case. Future work needs to be able to handle hybrid layouts and perhaps even weight the many different layouts by the prevalence they have on the screen.

Hybrid Categories also pose a challenge. For example an app that gives you the food and calories to eat healthy and help you exercise. The app might belong in the fitness and health category, but it also might fall under food and beverage. More research needs to be done to handle these scenarios, potentially returning multiple weighted categories for each app.

The current focus is on mobile applications (Android, iOS) but future word would involve a larger variety of UI applications. Future work would also involve improving each individual classifier as well as adding new classifiers. For example, a classifier to

interpret different icons on the screenshot, (Ex. Play button) to help the category classifier, could be incorporated.

Another addition would be to give the Framework greater insight into the structure of the screenshot layout and allow for a deeper and more specific search. For example, if it could traverse the nested layout hierarchy of the screenshot, then more specific searches like “List layout were each element contains an image to the right of Text” would be possible.

The Framework is a great start in allowing incremental improvements in the steps to categorize and classify UI Applications. The architecture of the framework is such that we can configure multiple classifiers for each of the 3 major steps, allowing the different techniques to be used.

References

[1] T. A. Nguyen and C. Csallner, "Reverse Engineering Mobile Application User Interfaces With REMAUI," in *Proc. 30th IEEE/ACM International Conference on Automated*, IEEE, Nov. 2015, pp. 248-259.

[2] T. M. Breuel, "Two Geometric Algorithms for Layout Analysis," in *Proceedings of the 5th International Workshop on Document Analysis Systems V (DAS '02)*, London, UK, UK, Springer-Verlag, 2002, pp. 188-199.

[3] B. A. Harit, S. D. Roy and Gaurav, "Extraction of Layout Entities and Sub-layout Query-based Retrieval of Document Images," *CoRR,* vol. abs/1609.02687, 2016.

[4] G. Bradski, "opencv_library," *Dr. Dobb's Journal of Software Tools,* 2000.

[5] H. Jianying, R. Kashi and G. Wilfong, "Document classification using layout analysis," in *Proceedings. Tenth International Workshop on Database and Expert Systems Applications. DEXA 99*, 1999, pp. 556-560.

[6] S. S. Bukhari, F. Shafait and T. M. Breuel, "High Performance Layout Analysis of Arabic and Urdu Document Images," in *2011 International Conference on Document Analysis and Recognition*, IEEE, 2011, pp. 1275-1279.

[7] R. Smith, "An Overview of the Tesseract OCR Engine," in *Proc. Ninth Int. Conference on Document Analysis and Recognition (ICDAR)*, 2007, pp. 629-633.

[8] J. Canny, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vols. PAMI-8, pp. 679-698, 1986.

[9] J. Yang, Y.-G. Jiang, A. G. Hauptmann and C.-W. Ngo, "Evaluating Bag-of-visual-words Representations in Scene Classification," in *Proceedings of the International Workshop on Workshop on Multimedia Information Retrieval*, New York, NY, ACM, 2007, pp. 197-206.

[10] E. Klinger and D. Starkweather, "pHash," 2008-2010. [Online]. Available: www.phash.org. [Accessed 20 October 2016].

[11] Apache, "Apache Solr," [Online]. Available: http://lucene.apache.org/solr/resources.html#documentation. [Accessed 20 August 2016].

[12] K. Kitani, "Bag-of-Visual-Words," [Online]. Available: http://www.cs.cmu.edu/~16385/lectures/Lecture12.pdf.

[13] J. v. Beusekom, D. Keysers, F. Shafait and T. Breuel, "Distance measures for layout-based document image retrieval," in *Second International Conference on Document Image Analysis for Libraries (DIAL'06)*, IEEE, 2006, pp. 11 pp.-242.