Research Report

Pill Detection & Prescription Analysis

Gitika Bose (gb2606)
Tejit Pabari (tvp2107)

May 10, 2019

Table of Contents

Table of Contents **1**

Introduction 3

Pill Recognition **5**

- Introduction 5
- Collecting Data 6
- Shape 7
  - Opencv’s Grabcut 8
  - K-Means Clustering 9
  - Contour Detection 11
  - Template Matching 13
- Color 15
- Imprint 21
  - Initial Attempts 21
  - Google Vision Model 23
- Integration 26

Text Extraction from Labels **27**

- Introduction 27
- Text Detection 28
- Natural Language Processing 29

Extensions **33**

- Pill Recognition 33
- Text Extraction 35

Citations **36**

Appendix **37**

Introduction

The main objective of this research was to apply computer vision and artificial intelligence techniques to identify pills and extract information from the labels on pill bottles. This project began with an app we created earlier that sorts and organizes an individual's medical files. Given the data, it also creates reminders to take medicines on time, with a description of the medicine and how it should be taken. However, a major flaw in the app's design was that users would have to manually enter data from a prescription into a form. Hence, we decided to write a program that extracts information from a prescription (typically found on pill bottles) in a useful format. Additionally, given that we wanted to increase people's organization with respect to their medical lives, we decided to design an algorithm to identify pills based on their shapes, colors, and imprints.

In this report, we first discuss our approach to identifying pills using primarily opencv algorithms. Given the shape, color, and imprint of a pill, we were able to query an FDA approved database. To determine the shape of a pill, we first use various preprocessing algorithms such as grabcut and k-means clustering to remove the background. Then, we apply the the findContours() function built into the opencv framework to get contours surrounding the pill. Since we had images of pills from the database, we created templates for each shape so that we can perform template matching against the pill contour to determine the closest shape. For the color, the image is again preprocessed with grabcut. To identify the colors, we perform k-means clustering and isolated three cluster centers, one of which is always black and thus removed from consideration. To determine the name of the corresponding colors, we used k-nearest neighbours by comparing the distance between the cluster centers and every RGB value observed for each color and choosing the majority of the 5 closest colors.

Finally, we had to determine the imprint on a pill from a given picture. This proved to be one of the most challenging and important parts of pill identification. Ultimately, we used Google Vision to train using dataset from the database. Through this method, we are able to recognize most letters and numbers that appear on pills without much preprocessing. This method and all other algorithms discussed here are explained in detail throughout the report.

For the next part of this report, we discuss our approach to text detection and extraction from pill labels. To recognize the text, we used the Google Vision OCR algorithm, which was very successful. To extract relevant information from this text, we combined our knowledge of the structure of prescriptions on pill bottles and Natural Language Processing techniques. For example, to determine the name of the medicine, we first query the database for every every word identified in the label. From this, we make a list of potential names and compare them to standard drug nomenclature to determine which one is a medicine name. If we are not able to narrow down one choice for the medicine name, we choose the largest word of all potential medicine names. With similar techniques, we are able to isolate the pharmacy name, dosage and form of medication, and how and when to take it. This completes the prescription and can be easily integrated into our app.

Finally, we highlight areas of improvements and potential extensions to this research. For example, it would be useful to integrate our pill identification and prescription analysis algorithms to our app. Other improvements include training a model with more samples, using multiple databases to check for medicine names, and performing post processing of text using other NLP algorithms.

Pill Recognition

Introduction

Our first objective was to recognize a pill given a picture. We did this by extracting features from that picture that uniquely identify it.

Our first step was to gather an understanding of the problem at hand. After going through standardized pill databases, we found that each pill can be recognized by the combination of three main features: color, shape and imprint.

To elaborate on these three features:

1. Shape: The database classifies the pills into 15 shapes, including some common ones such as square and pentagon, alongside obscure ones such as freeform. To identify the shape of the pill in the picture, we use opencv's grabcut-algorithm to obtain the outline of the pill, then passed it through K-Means and processed it to clear up the background, and then used grabcut again to get the contours of the pill. We compared the contours against templates of differently shaped pills that we extracted earlier. This gives us the number of pixels in the image not matching the templates', which when passed through a threshold function, gives us the top shape possible for that pill image.
2. Color: Based on our observation, a pill can have one or two colors in itself. To identify the color of the pill, we use K-Means clustering with $k = 3$ (1 for the background) and check the hex values of the resultant clusters. If any two hex values are within the threshold set by us, then we consider them as one (the pill is only single colored).
3. Imprint: Every pill that requires a prescription is mandated to have an imprint on it. An imprint can be formed of any letter, upper or lower, or number. To identify imprint, we trained a neural network for letters and numbers and processed the result to get individual imprint and group imprints.

Collecting Data

To collect data, we used the rx-API. We considered this amongst other APIs because of three major reasons:

1. It was easier to use and query as compared to other APIs/websites. Often we had to deal with websites and that required parsing through HTML code to get the desired information. Hence we chose the most convenient option available.
2. It was updated and had concise information, information that we needed. Other API's were not updated according to the latest FDA standards.
3. It had clear images that we could use to develop our algorithms and models. Other API's/websites had images missing for some medicines, while some weren't clear.

After selecting the API, we parsed through all different categorizations we can use to group the pills in the database. We used shape and color to group the pills, since we have to cover all shapes and color for the shape and color algorithm. We downloaded all images by the category for further use.

We also downloaded data from the database, to make a mini database of our own, with data for each pill including their names, urls, imprints, colors and shapes. We made dictionaries for all single word imprint in the database (A-Z,a-z,0-9), storing each imprint letter with the pills that have the letter. These are used later to shorten the list of pills and identify pills quickly. This helps as the database allow querying of a single letter imprint, which may be possible in several scenarios. We made similar dictionaries for color and shape.

Shape

The pill database gives us a total of 15 shapes used to classify pills. Some of these shapes have a significantly high number of pills (with round as the highest at 2000+ images), while some have very few (with freeform, octagon as the lowest with 3 images each). Some shapes occur commonly, while others are obscure.

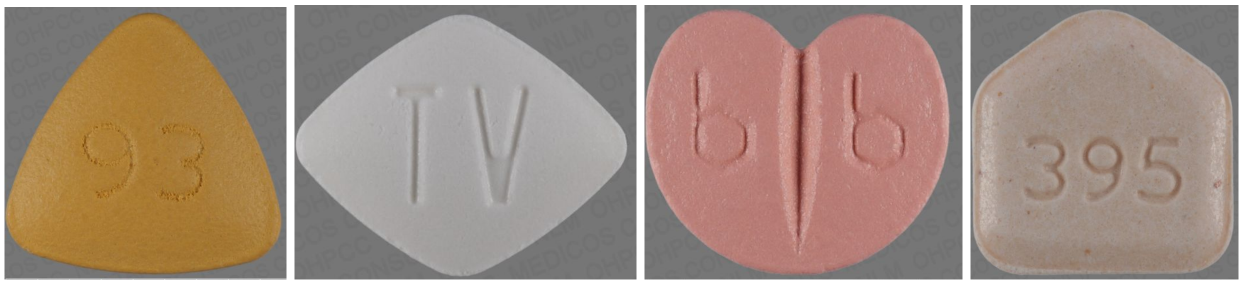


Figure 1: Pills of Different Shapes[1]

We researched multiple algorithms to extract the shape of the pill and even developed our own. One significant algorithm that we developed, but did not end up using, was to extract pixels in a 5 × 5 box, from the corners of the pill image and consider them as the background. Any pixel in the picture that comes in the set of background pixels is changed to (0,0,0) i.e. black, which gave us the foreground. This was a good method, however, it didn't work on images with multicolored and patterned background, since the pixels in the corners do not necessarily represent the background of the whole picture in such cases. Out final method of choice was opencv's grabcut-algorithm.

Applying Grabcut to a pill gives us:

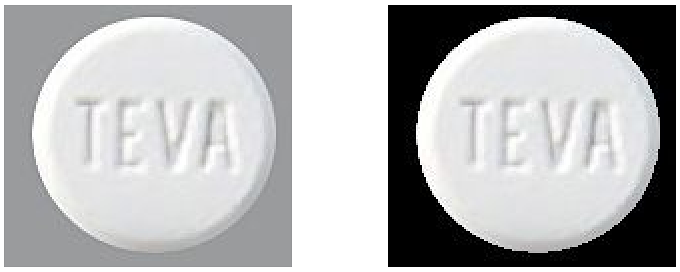


Figure 2: Pill Before Grabcut (left). Pill After Grabcut (right)

[1] https://rxnav.nlm.nih.gov/RxImageAPIs.html#

Opencv’s Grabcut

The grabcut algorithm works in the following way:

1. The user of the algorithmselects a rectangle that highlights the foreground of the image. All pixels outside the rectangle are classified as definite background - called known pixels, while images inside the rectangle can be potential background, hence called unknown pixels.
2. The foreground and background are modeled as Gaussian Mixture Models (GMMs).
3. The Foreground GMM and the Background GMM assign a label to each pixel (the most potential gaussian component for that pixel). New GMMs are learned from this.
4. A graph of the pixels is made and Graph Cut is used to find a new classification of the foreground and background pixels.
5. Steps 3-4 are repeated as defined by the user or until convergence is reached.

Code for Grabcut Algorithm:

```
# Create a mask of the image, similar to the image. This is where the algorithm marks pixels
as foreground or background
mask = np.zeros(img.shape[:2],np.uint8)

# Define the Foreground and Background Models - arrays used by the algorithm internally
bgdModel = np.zeros((1,65),np.float64)
fgdModel = np.zeros((1,65),np.float64)

# Define the Rectangular Foreground. Everything outside the box is considered a definite
background
rect = (5,5,img.shape[1]-5,img.shape[0]-5)

# Run the algorithm, with 5 iterations in this case. cv2.GC_INIT_WITH_RECT is used because
the rectangle mode is used.
cv2.grabCut(img,mask,rect,bgdModel,fgdModel,5,cv2.GC_INIT_WITH_RECT)

# The mask has four labels. All pixels labelled 0 and 2 are converted to background (black)
and all pixels labelled 1,3 are kept as foreground.
mask2 = np.where((mask==2)|(mask==0),0,1).astype('uint8')

# The mask is multiplied with the input image to give the new image.
img = img*mask2[:,:,np.newaxis]
```

Amongst all the algorithms we tried, grabcut gave the best result since it uses the most sophisticated and precise way to segment the foreground and the background. However, it sometimes leaves some part of the background out, such as:

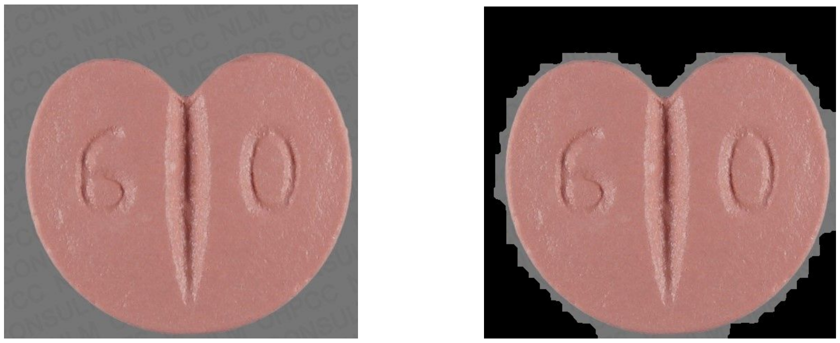


Figure 3: Faulty Grabcut - Pill Before Grabcut (left). Pill After Grabcut (right)

In the above image, grabcut does not crop the pixels near the border, as they are too near to the pill and hence, grabcut groups them with the pill. To resolve the issue, we use K-Means.

Also, at this point, the image is passed on to the Color algorithm.

K-Means Clustering

K-means Clustering takes the pixels of the image and clusters them in k parts. It starts with k random points as cluster, classifying each pixel to a cluster center and then recalculates the cluster centers as the mean of each cluster. It continues this process until convergence is reached and all the points are classified. A full explanation and a detailed implementation of the algorithm is given in the next section (Color), since it is used primarily to identify the color. We use a clustering of 4, i.e. 4 cluster centers are made. After K-Means, the image looks like:

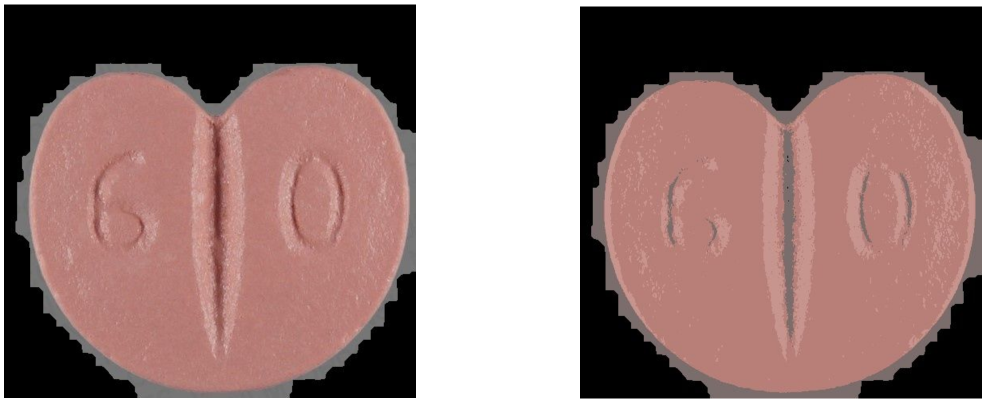


Figure 4: Before K-Means Clustering (left) After K-Means clustering(right)

We decided to use *k*=4 since we assumed that, in worse case scenario, there is some background left after the grabcut and that the pill in the picture has the maximum colors it can have i.e. 2. Summing them up gives me 3, plus one for the black, which leads to 4 clusters.

In case of this picture, clustering doesn't help because the pill is single colored and so is the background. Hence, it classifies the pink of the pill using two clusters, the background as the third cluster and black as the fourth. Hence, we post process the image after K-Means.

In post processing, we consider 5 pixels from all the edges of the picture as background. We then make a set of the pixels and remove those which have a count less than 500, since it is possible that there were a few pixels in the background with the same color as the pill, pixels left out by grabcut, that we can't see with our naked eye because of the black. This results in a set of pictures forming only the background, pixels that can be deleted from the picture to give the foreground. The algorithm for post processing is as follows:

```
# 5 pixels from the edges as background
edge_pixels = {}
w_in = [x for x in range(5,w-5)]
h_in = [x for x in range(5,h-5)]

# parse through the image and count all the pixels in the selection above
for i in range(h):
   for j in range(w):
       if i not in h_in:
           if tuple(new_img[i][j]) not in edge_pixels: edge_pixels[tuple(new_img[i][j])] = 0
           edge_pixels[tuple(new_img[i][j])] += 1
       if j not in w_in:
           if tuple(new_img[i][j]) not in edge_pixels: edge_pixels[tuple(new_img[i][j])] = 0
           edge_pixels[tuple(new_img[i][j])] += 1


# remove pixels with count less than 500
to_delete = [e for e in edge_pixels if edge_pixels[e]<500]
for t in to_delete: del edge_pixels[t]

# parse through the image and set all the remaining background pixels to (0,0,0) i.e. black
edge_pixels = set(edge_pixels.keys())
for i in range(h):
   for j in range(w):
       if tuple(new_img[i][j]) in edge_pixels:
           new_img[i][j] = [0,0,0]
```

After post-processing the image becomes:

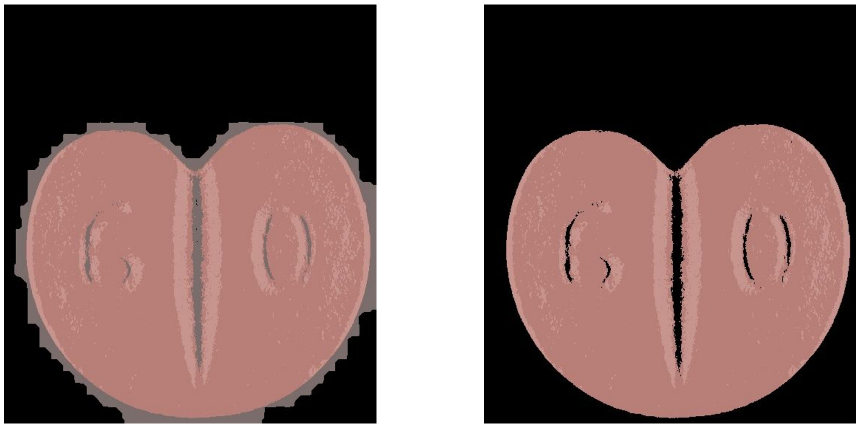


Figure 4: Image before post-processing (left)   Image post-processing (right)

As seen above, post processing effectively removes the grey background.

Contour Detection

We finally have a picture of the foreground with a black background. To get the shape of the pill, we extract the contours from this image. Contours are curves joining all continuous points along the boundary that have the same color or intensity[2]. Before getting the contours, we process the image by converting the non-black pixels of the images to white. Next we blur it, using the GaussianBlur method, which is the blurring of an image by a Gaussian Function. Then we convert the image to black and white using cv2's COLOR_BGR2GRAY function and threshold it by passing it through cv2's Otsu's Threshold, which makes all non black pixels white, converting the image to a pure binary image. All this processing helps with contour detection.

We then pass this new image through cv2's findContour function. The findContour function a list of contours; oftentimes, there are more than one, however all are insignificant except the the one outlining the shape. Post finding the largest contour, we make a blank image such that the contour lies exactly on the edge of the image, and then draw the contours out on a new image using cv2's drawContour function.

[2] *OpenCV: Contours : Getting Started*. https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html. Accessed 9 May 2019.

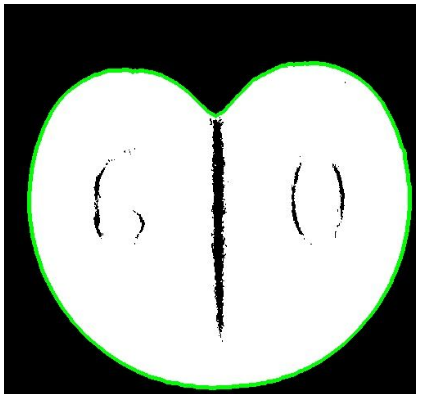
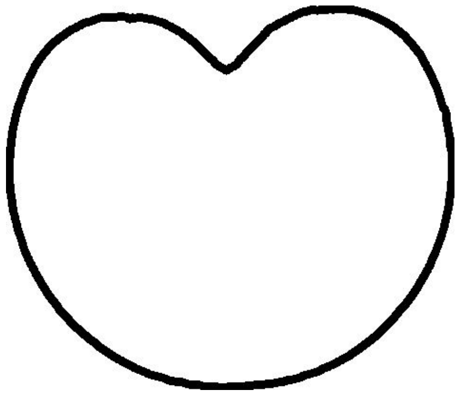



Figure 5: Image with Contours (left)   Outline of the pill made from the largest contour (right)

The algorithm for finding contours is:

```
# Changing all non-black pixels to white to ready the image for find-contour
img[np.where((img!=[0,0,0]).all(axis=2))] = [255,255,255]

# Blurr, Gray and Thresholding to ready the image for find-contour
blurred = cv2.GaussianBlur(img, (5, 5), 0)
gray = cv2.cvtColor(blurred, cv2.COLOR_BGR2GRAY)
thresh = cv2.threshold(gray,0,255,cv2.THRESH_OTSU)[1]

# Finding contours using cv2's findContours
cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
cnts = imutils.grab_contours(cnts)

# Calculating the largest contour
final = []
for c in cnts:
   if len(final)==0:
       final = c
   elif len(c) > len(final):
       final = c

# Making a blank image of the size of the contour, with 5 pixels as leeway.
# cv2's boundingRect returns a rectangle around the contour.
# x_n and y_n arrays shift the contour to the edge of the image
# Draw contour draws the contour on the new image
x, y, w, h = cv2.boundingRect(final)
white = np.zeros((h+5,w+5),np.uint8)
white[white == 0] = 255
x_n = [c[0][0]-x+3 for c in final]
y_n = [c[0][1]-y+3 for c in final]
arr = [[x_n[i], y_n[i]] for i in range(len(x_n))]
ctr = np.array(arr).reshape((-1,1,2)).astype(np.int32)
cv2.drawContours(white, [ctr], -1, (0, 0, 0), 8)
```

Template Matching

We now have the outline for the pill-image, next comes finding what the shape actually is. To do so, we use a modified version of template matching. Template matching is a technique used to find parts of an image that match the given template. It slides the template over the image and compares the template and patch of input image under the template[3]. In our case, we have pre-extracted outlines for each shape, which we use as templates, while the outline of the pill-image forms the image the template is compared against. We resize these two images to the same width and compare them using the mean square error function. This function finds the difference between each pixel, squares it, sums this for all pixels and divides that by the product of number of pixels in both images, giving the mean of the squared error. The lower the error, the better the images match each other. The Mean Square error function is:

```
def mse(imageA, imageB):
   # convert pixel values to float, find the difference and square it.
   # Sum the error for each pixel and divide by product of total pixels to obtain the mean
   err = np.sum((imageA.astype("float") - imageB.astype("float")) ** 2)
   err /= float(imageA.shape[0] * imageA.shape[1])
   return err
```

We calculate the Mean Square error of the picture with all the templates and take the lowest one as our first shape. We don't consider it as the final shape, since we found that often shapes are closer to each other and the first shape might not be the one that the pill refers to. Even though the template comparison shows otherwise, pills can still vary by a bit and the shapes can look like one another.

To counter this, we group together similar looking shapes, such as oval and diamond:

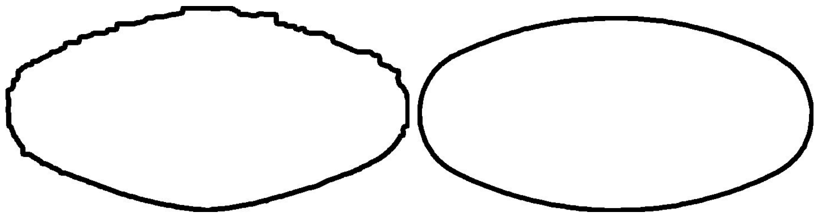


Figure 6: Diamond (left) and Oval (right)

[3] *Template Matching — OpenCV-Python Tutorials 1 Documentation*. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_template_matching/py_template_matching.html. Accessed 9 May 2019.

We find the group the first shape belongs to and multiply the mean square error for each shape in that group by the probability of not finding that shape amongst all pills. We do this since we are comparing lower numbers, i.e. if the probability of finding a shape is high, probability of not finding it is low and hence multiplying it by the mean square error error will decrease the mean square error more than that of a shape which has a low probability of being found and hence a high probability of not finding. For instance, say the mean square errors for oval and diamond are 16000 and 14000 respectively. The probability for not finding oval and diamond is 0.665 and 0.996 respectively. Multiplying them gives 10640 and 13944 for oval and diamond respectively. Here oval is much lesser than diamond, even though initially it was much larger. Thus, oval is selected since it has a higher chance of occuring.

The code for the algorithm is:

```
for shape in shapes:
   temp = cv2.imread(template_folder+shape+'.png')
   template = temp.copy()
   h_t, w_t = template.shape[:2]
   # If the height of image is >= half of the template or vice versa, don't compare them
   if h >= h_t*2 or h_t >= h*2: compare.append((99999999,shape))
   else:
       w_i, h_i = min(w, w_t), min(h, h_t)
       # finding the mean square error
       result = mse(img[:h_i,:w_i],template[:h_i,:w_i])
       compare.append((result, shape))
       compare_dict[shape] = result

# remove all not-compared shapes and sort them by the mean square errors
compare = [x for x in compare if x[0]!=99999999]
compare.sort(key=lambda x:x[0])

# take the first shape
ans = compare[0][1]
for g in groups:
   if ans in g:
       # for all shape in that group, multiply them by the probability of not being found.
       for gg in g:
           if gg in compare_dict:
               result = compare_dict[gg] * (1-probability[gg]/4400)
               compare2.append((result,gg))
compare2.sort(key=lambda x:x[0])
```

The element with the lowest error is our shape. Running the shape algorithm takes 5.7 seconds.

Color

Color is another distinguishing feature of a pill. Based on the database used, there are a total of 12 possible colors for a pill with white being the most common. Initially, we calculated the mean of pixels in different parts of the pill. Then, we chose the color (from the 12 identified colors) with the minimum euclidean distance to the mean. The first major problem with this method was that it was not reliable. While the pills came in a variety of shades, this algorithm was based on choosing an arbitrary shade for each color. This is especially problematic if the RGB value of the pixels in the image were between two of the given colors. For example, it identified the color of the pill below, shown on the left in Figure 2, as “gray” even though the label given in the database was “white”. Secondly, we could not predict whether the pill consisted of one or two colors. Hence, we were not sure which regions we should use in detecting the “mean” color of the pill. The image below shows an example of a capsule with two colors (on the right).

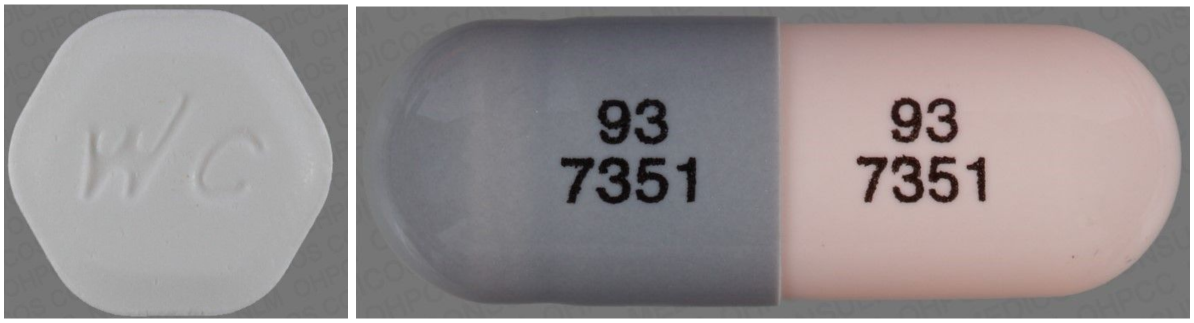


Figure 7: White pill easily mistaken for gray (left) and pill with two colors (right)

K-Means Clustering

A more suitable alternative was to use **k-means clustering** to divide the colors in the pill and background clearly. K-means clustering typically chooses *k* cluster centers arbitrarily and changes them until the data is divided into *k* clusters such that each center is the mean of all values in one of the clusters. The data is fully divided once all values in a cluster are closest to their cluster center. We decided to set *k = 3* for all images to account for pills consisting of two colors. To perform k-means clustering, we used the library “scikit-learn” as follows:

```
# Opens image using PIL and gets all pixels in a list
im = Image.open(input_file)
pixels = list(im.getdata())

# Converts the pixels from RGB values to uint8
pixels = [(x/255, y/255, z/255) for (x,y,z) in pixels]
# Clusters the data using c as the number of clusters
c = 3
cluster = KMeans(n_clusters=c).fit(pixels)

# Makes a list of the cluster centers (dominant colors)
colors = cluster.cluster_centers_
color_rgb = [(x*255, y*255, z*255) for [x,y,z] in colors]
```

This algorithm was advantageous because we no longer had to choose regions in advance to study as the algorithm adapts well to new images. To ensure that there was less disturbance from the background, we used the grabcut algorithm (introduced above with the section regarding Shape) on every image before attempting to decide its color. Since the algorithm is designed to make the background black, we made an assumption that one of one of the cluster centers would be black and could be removed from consideration. Figure 8 below shows the effect of using grabcut and k-means clustering on two images of pills.

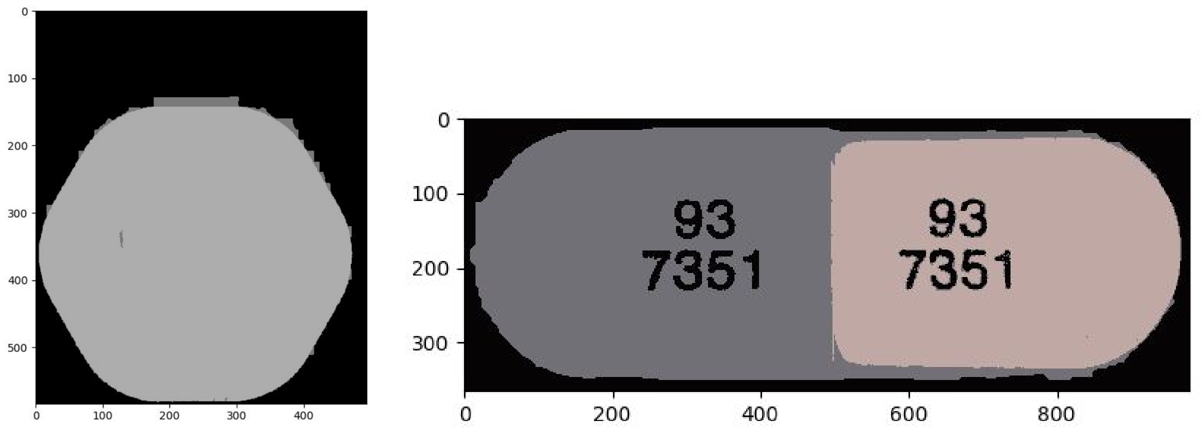


Figure 8: Grabcut and k-means clustering on a white pill (left) and a two color pill

According to these images, this algorithm is successful in dividing the pill and background into its respective colors. The pill on the left in Figure 8 highlights a caveat of using k-means algorithm with *k = 3*. It returned two cluster centers for the pill, one of which evaluated to white while the other was gray. To isolate white, we compared the predictions for all the pixels in the image. We found that while approximately 160000 pixels were white, only 4000 were gray. Given that this is not close to an equal division of the two colors we decided, it is obvious that white is the only color for the pill. Generalizing this idea to all pills, we decided that if two different colors are returned by the algorithm, we will check if the pixels of one color are more that double the number of pixels of the other color. If this is the case, we return only the more frequent one. If not, both colors are returned as if the pill consists of 2 colors (shown on the right in Figure 8).

K-Nearest Neighbours

While k-means clustering is effective in isolating the correct RGB values for the pill, this method did not solve our first major problem of colors existing in many shades instead of the one arbitrary shade. To account for this, we extracted all the RGB values for every color that appeared in the database (for only single colored pills) using k-means with $k = 2$ and saved these values to text files. The images below shows 3D visualizations (using matplotlib) of the range of colors present for various pills. On the left we show a plot consisting of all RGB values for yellow and green, while the right shows all RGB values for blue and brown. The points are marked by their respective colors.

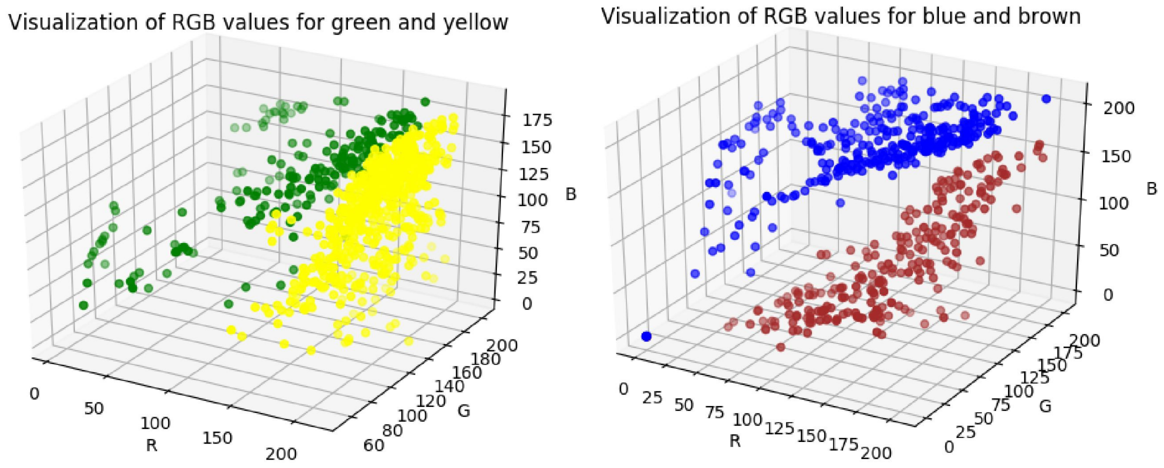


Figure 9: Visualization of RGB values for various colors in the database

After performing k-means clustering on a new image, we used **k-nearest neighbours** with the dataset described above to classify the RGB value as one of the 12 colors. Since there were less than 4000 unique colors to compare with, we realized that a brute force implementation of KNN would be appropriate in this case.

```
# Loop through the cluster centers
for c in color_rgb:

   # Creates list of tuples of euclidean distances and the color names
   all = []
   for color in colors:
       for i in color_values[color]:
           if i:
               d = dist.euclidean(c, i)
               all.append((d,color))

   # Sorts the list and selected the top k (in this case, k=5) values
   all.sort(key=lambda x: x[0])
   all = all[:5]
```

Hence, we compared the new cluster centers with each RGB value for each color using Euclidean distance and performed majority voting on the *k* nearest values. After some trial and error, we chose $k = 5$ as it often led to the correct answer and seems to be a fair number in this case. In the case of a tie, we decided to choose the set of neighbours whose total distance from the point was least. The 3D plots in Figure 9 indicate how a point is classified based on the region in which it falls on the graph. While this model worked well with most pill images, it was faulty in some instances. For example, the image below (left) shows a pink pill which the algorithm identifies as white.

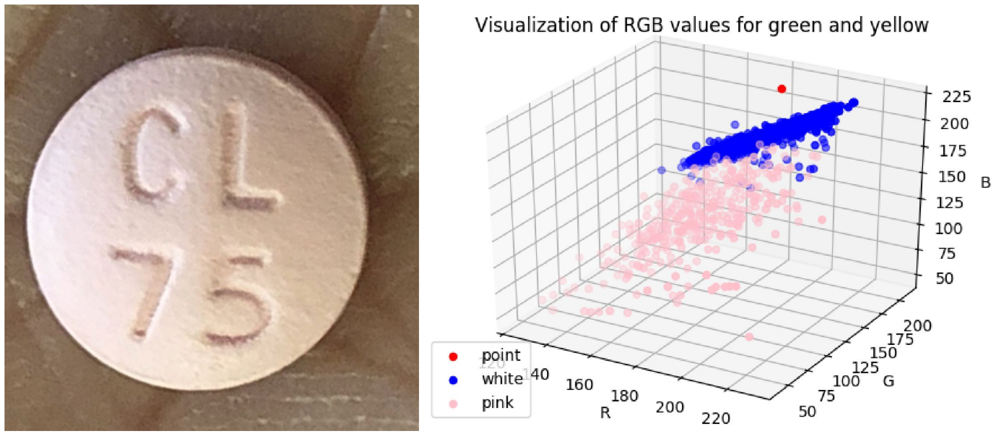


Figure 10: Misclassified image (left) and visualization of pink and white RGB values along with misclassified color (right)

The image on the right in Figure 10 is the 3D visualization of all white and pink RGB values with the cluster center for the pill, returned by the k-means algorithm. According to this plot, the B value was so high that it was categorized as white. To understand why this occurred, we found the database image for the pill and compared this processed image to the database one. When the algorithm is run on the database pill, it returns "pink". The images are shown below.

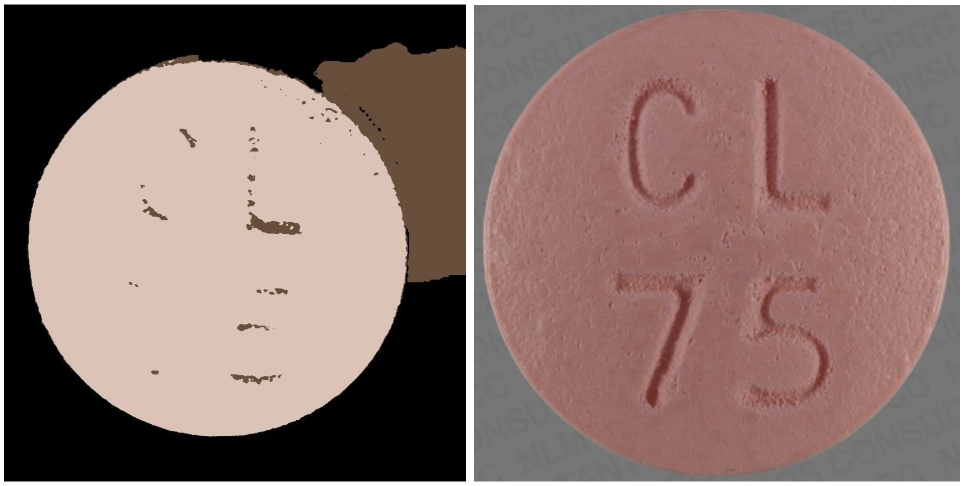


Figure 11: Processed misclassified pill (left) and pill from database (right)

This highlights the effect of lighting and quality of picture taken as there is significant difference between the picture we took and the actual pill. However, another issue is that even though we used 5 nearest neighbours, there was significant overlap between some RGB values which could potentially distort the color returned. Consider the example shown below.

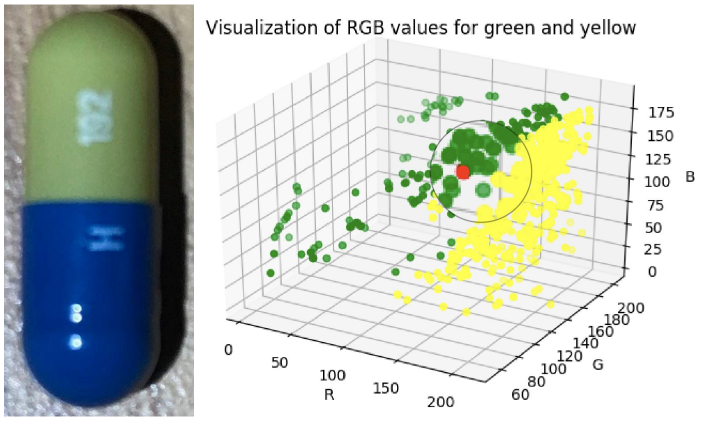


Figure 12: Classified pill (left) and visualization of yellow and green RGB values along with the cluster center for yellow from the pill (right)

The image on the right in Figure 12 is once again the 3D visualization of all green and yellow RGB values. However, this time we are focusing on the point (129, 153, 119) which was the cluster center for the green side of the pill, returned by the k-means algorithm. Clearly, it falls in a region between yellow and green where there is significant overlap. Personally, I had trouble deciding between the two colors and was leaning towards yellow before I checked the database. Similarly, the algorithm could have misclassified it as yellow, instead of green. While there were no issues in this case, this error is difficult to eradicate and hence the algorithm is only mostly effective. However, in an attempt to reduce the likelihood of such an error, we decided to clean up the data collected for the various RGB values. To do this, we calculated the mean and standard of the data for each color. Then, using their *z*-scores, we checked that each value was within 3 standard deviations of the mean. The code snippet below shows how we did this:

```
def outliers(data):

   # Creates a list of outliers to be removed later
   outliers = []

   # Calculates mean and standard deviation
   mean = np.mean(data)
   std =np.std(data)

   for v in data:
       # Checks if z score is less than 3 standard deviations
       z_score= (v - mean)/std
       if np.abs(z_score) > 3:
           outliers.append(v)

   return outliers
```

Finally, after removing outliers, we used images of pills taken by us and found that k-means clustering with k-nearest neighbours for classification of colors was accurate in most cases.

Running the Color algorithm takes a total of 3.8 seconds, however, it shares 1 second with the grabcut used for shape as well.

Imprint

Initial Attempts

Given that the imprint is one of the most important ways to uniquely identify a pill, recognizing imprints was arguably the most challenging aspect of identifying pills. To get started, we used Tesseract OCR and Google Vision in an attempt to extract text from the pill. However, we quickly noticed that these algorithms were not suited for unusual fonts such as the the one shown below in Figure 13 (leftmost image). A better alternative, we thought, was handwriting recognition algorithms. However, as seen in Figure 13 (middle) it was also unsuccessful because because the font is unlike any handwriting and the text is usually engraved into the pill making it difficult to discern, sometimes even for the human eye. The images below shows two pills in which the imprint is unclear for these reasons.

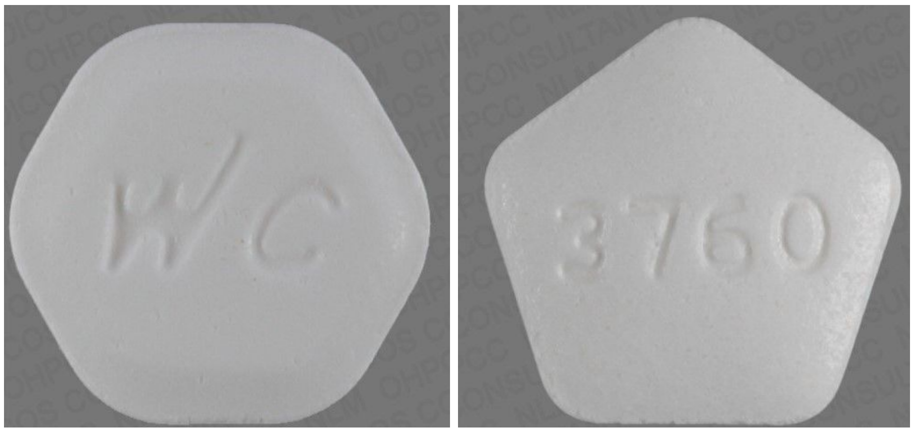


Figure 13: Pills with imprints that are unclear due to unusual fonts (left) and faint engravings (right)

To rectify these issues, we decided to focus on the preprocessing of the image. Initially, we performed Adaptive Gaussian thresholding, in an attempt to make the image black and white with clear text. However, we found that it did not affect the image significantly. Instead, we used our own algorithm for which we decided a fixed threshold. Any pixel whose RGB value was below than the threshold was immediately classified as black. The intent of this algorithm was to highlight the text, which is typically darker than the rest of the pill. The result of the algorithm is shown below in Figure 14.

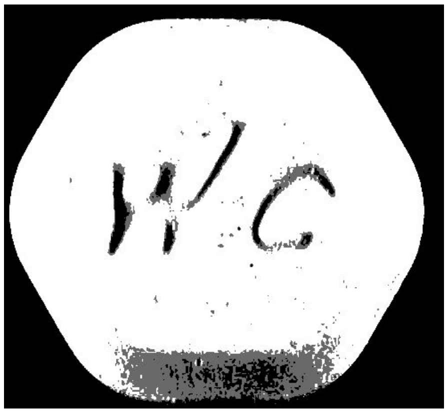


Figure 14: Pill with WC after preprocessing with algorithm

As seen above in Figure 14, the main issue with this algorithm is that the letters were not complete even though there are other regions of the pill that also turned black and gray colored. We also tried to apply k-means clustering since it was successful in determining shape and color. The results are shown below.

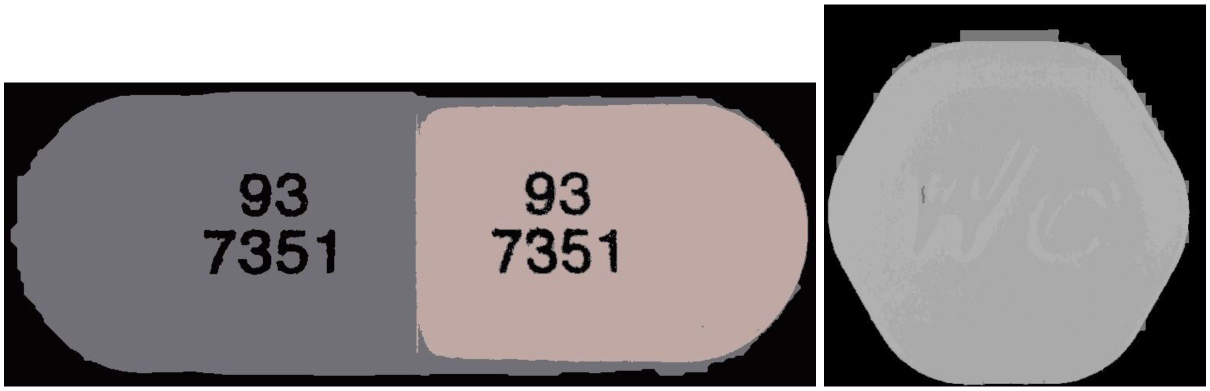


Figure 15: Pills after k-means clustering

Evidently, the pill on the left in Figure 15 was made clear as the imprint was classified immediately as black while the pill itself was a lighter color. The image on the right, however, was affected even less than the algorithm we wrote. Most of the pill is divided into different shades of gray but there are no discernable letters. Finally, we attempted to make the image clearer using simple techniques like increasing the contrast and sharpness. However, ultimately we found that none of these techniques were effective. Hence, we decided to train a model.

Google Vision Model

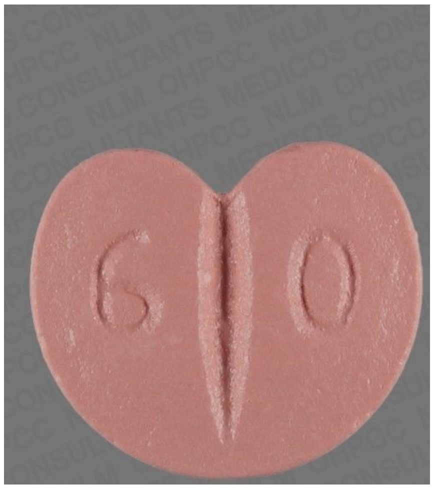
To train our model we chose Google Vision, since we found that it a better text recognition algorithm, in general with an easy interface for training. Before training we assimilated the data for all the letters and numbers separately and labelled them such that there was a max of 2.5 to 1 ratio between the number of pictures labelled for each letter (so that, for instance if J is 40, we labelled about 100 A's). We did so because this ensured seamless training of the model and prohibited skewing of the data. We chose to train the number and letters model separately since there were many more images for each number than there were for each letter.

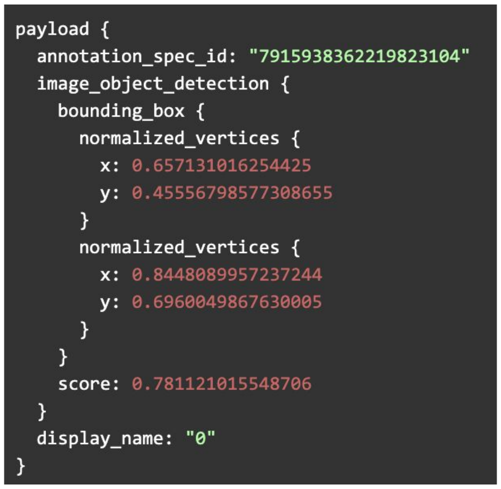
To label the images, we used a software called RectLabel. It gave us the xml files for each labelled image, from which we extracted the labels and the coordinates for bounding boxes into a csv file. We then uploaded it to google vision ML training, which gave us a model.

Post labelling, we called our model to get predictions. We used the following function:

```python
def get_prediction(content, project_id, model_id):
   prediction_client = automl_v1beta1.PredictionServiceClient()
   name = 'projects/{}/locations/us-central1/models/{}'.format(project_id, model_id)
   payload = {'image': {'image_bytes': content }}
   params = {}
   request = prediction_client.predict(name, payload, params)
   return request
```

The function returns predictions to us. The prediction look something like this:





```
payload {
  annotation_spec_id: "7915938362219823104"
  image_object_detection {
    bounding_box {
      normalized_vertices {
        x: 0.657131016254425
        y: 0.45556798577308655
      }
      normalized_vertices {
        x: 0.8448089957237244
        y: 0.6960049867630005
      }
    }
    score: 0.781121015548706
  }
  display_name: "0"
}
```

Figure 16: Image (left) and Prediction (right)

The prediction contains the display name, the threshold score for that label and the bounding boxes.

Next, we resolve these bounding boxes in the y and x direction. In resolving them, we identify groupings of the letters and numbers, based on bounding boxes.

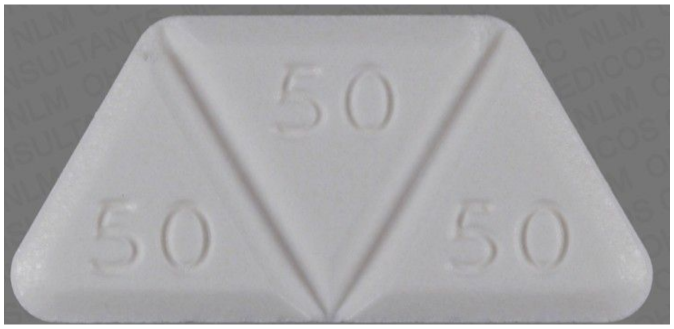


Figure 17: Sample pill for imprint recognition

For instance, for this pill, we get the labels ['5', '5', '5', '0', '0', '0'] from the model. Querying 5 and 0 separately (since we don't know the order) in the API gives us 100s of images. To remove the incorrect ones, we have to get the right grouping of the imprints, such that we get ['50', '50', '50'].

To do so, we first resolve the imprints in *y* direction. We check each *y* coordinate of the labels and group together labels that share 75% of the same area. This gives us seperate lines of imprints in the pill. The code is as follows:

```
# Parse through all the bounding labels
for r in res:
   ytop,ybot = r[2],r[4]
   check = True
   for b in bounds:

       # Calculating 75% of the y for the bounds
       s5 = 75*(b[1]-b[0])/100

       # Checks for checking if the label falls under already defined bounds.
       if b[0] <= ybot <= b[1] and b[0] <= ytop <= b[1] and (ybot-ytop) >= s5: ...
       elif ybot >= b[1] and b[0] >= ytop and (ybot-ytop) >= s5: ...
       elif b[0] <= ybot <= b[1] and (ybot-b[0]) >= s5: ...
       elif b[0] <= ytop <= b[1] and (b[1]-ytop) >= s5: ...

   # If it doesn't fall under defined bounds add to bounds
   if check:
      bounds.append([ytop, ybot, count])
return [sorted(v,key=lambda x:x[1]) for k,v in result.items()]
```

We do similar bound checking for x as well, grouping together labels that are near each other, as compared to the threshold (0.3). The code is:

```
for rr in res:
   bounds = []
   for r in rr:
       n, x1, x2 = r[0], r[1], r[3]
       check = True
       for b in bounds:
           b_n, b_x1, b_x2 = b[0], b[1], b[2]
           # If the two labels are less than 0.03 far, then consider them in a group
           if abs(x1-b_x2) <= 0.03:
               b[0] += n
               b[2] = x2
               check=False
               break
       # If it doesn't fall in any bound, add new bounds
       if check:
           bounds.append([r[0],r[1],r[3]])
   for b in bounds: result.append(b[0])
return result
```

Our final step includes making a set of all pills that have the imprint, the color and the shape. We intersect these three to get set of pills. If we get a return value from the resolved imprints, we parse through the set of pills we have and find which pill has the resolved imprints. If not, we just output the set of pills.

The final output of our Pill-Detection Algorithm Looks like this:

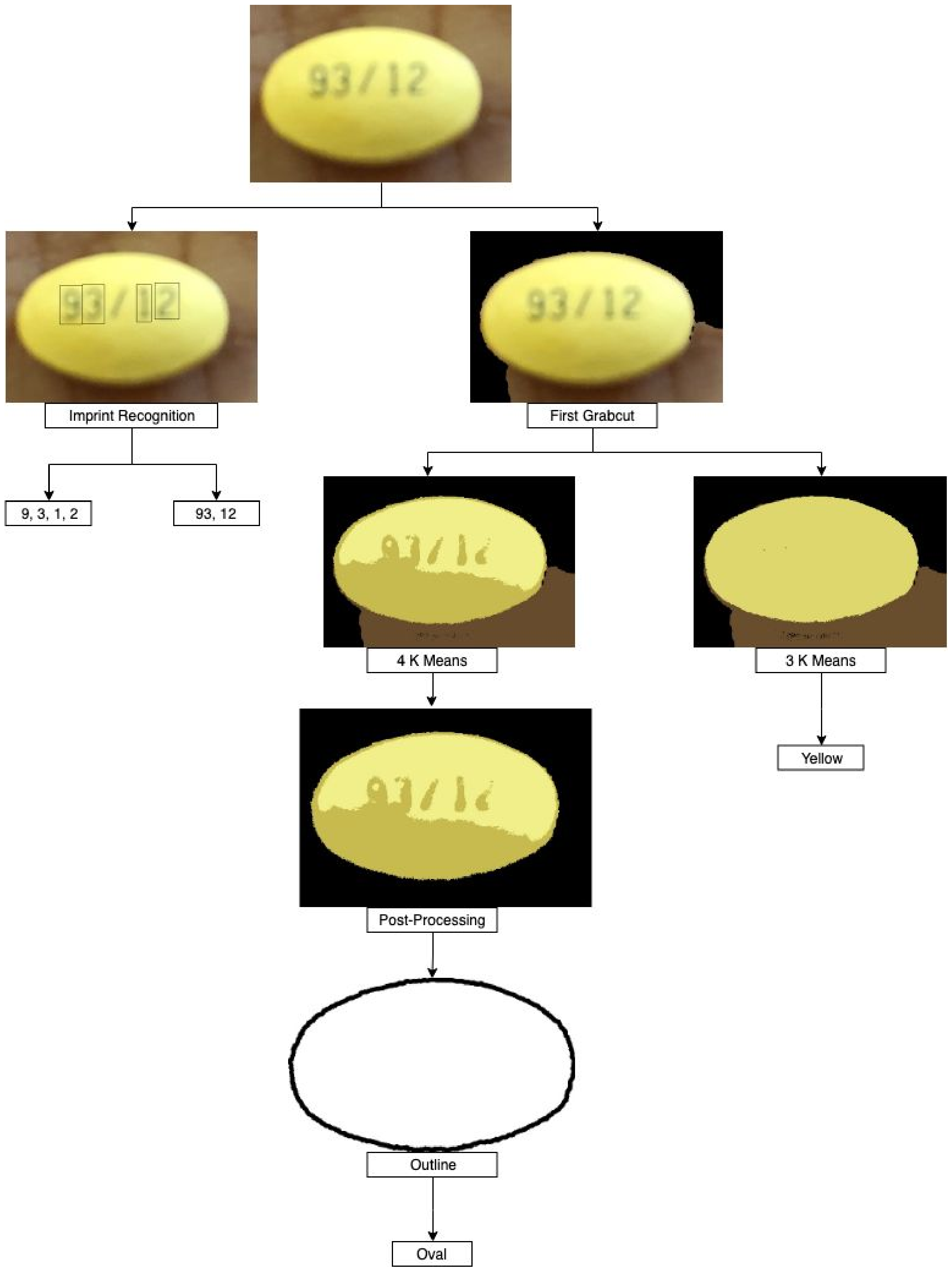
```
Color:  ['white']
Shape: ['trapezoid']
Imprint1: ['5', '5', '5', '0', '0', '0']
Imprint2: ['50', '50', '50']
Pill: 0001trapezoid, url:
https://rxpillimage.nlm.nih.gov/RxImage/image/images/gallery/original/50111-0441-02_RXNAVIMA
GE10_CB1265D3.jpg
```

We provide a pill name, which is how we have stored basic information about the pill in our database. We also give a URL to the picture and information of the pill in the actual database.

The imprint algorithm takes 2.3 seconds.

Integration

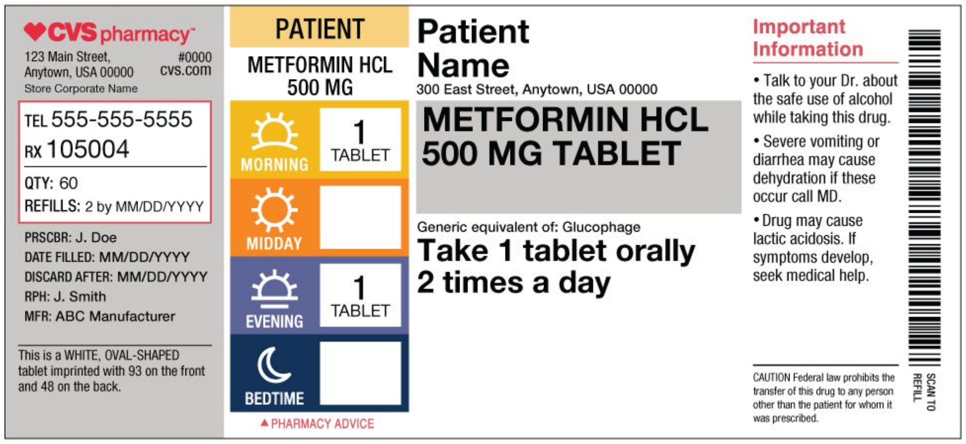
Our algorithm runs in 11 seconds. Below is the final design, shown using the pill “pantoprazol”:



Text Extraction from Labels

Introduction

Our second objective involves extracting information from prescriptions found as labels on pill bottles. The image below shows an example of such a label, found on CVS bottles.



CVS pharmacy
123 Main Street, #0000
Anytown, USA 00000 cvs.com
Store Corporate Name

TEL 555-555-5555
RX 105004
QTY: 60
REFILLS: 2 by MM/DD/YYYY

PRSCBR: J. Doe
DATE FILLED: MM/DD/YYYY
DISCARD AFTER: MM/DD/YYYY
RPH: J. Smith
MFR: ABC Manufacturer

This is a WHITE, OVAL-SHAPED tablet imprinted with 93 on the front and 48 on the back.

PATIENT
METFORMIN HCL
500 MG

MORNING	1 TABLET
MIDDAY	
EVENING	1 TABLET
BEDTIME	

▲PHARMACY ADVICE

Patient Name
300 East Street, Anytown, USA 00000

METFORMIN HCL
500 MG TABLET

Generic equivalent of: Glucophage
Take 1 tablet orally
2 times a day

Important Information

- Talk to your Dr. about the safe use of alcohol while taking this drug.
- Severe vomiting or diarrhea may cause dehydration if these occur call MD.
- Drug may cause lactic acidosis. If symptoms develop, seek medical help.

CAUTION Federal law prohibits the transfer of this drug to any person other than the patient for whom it was prescribed.

SCAN TO REFILL

Figure 18: An example of a typical label found on CVS pill bottles[4]

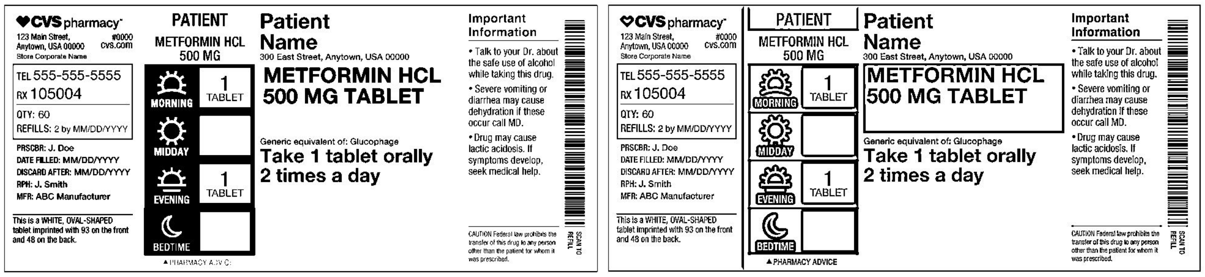
The process of extracting relevant information from labels can be divided into the following steps:

1. The first step in our research was recognizing the text shown in sample labels. To ensure the text would be accurate, we first researched on techniques to preprocess the image so that it is readable. Then, we focused on choosing a seemingly reliable algorithm that worked consistently for various labels. Ultimately, we applied adaptive gaussian thresholding on the image and converted to a grayscale image, which is preferred by Google Cloud Vision, the text detection framework that was finally used.
2. The next step was to apply Natural Language Processing to extract relevant information such as the name of the medicine, patient name, pharmacy, and other such details related to the medication.

[4] "ScriptPath Prescription Labels Help Make Adherence Easier." *CVS Health Payor Solutions*, https://payorsolutions.cvshealth.com/insights/scriptpath-prescription-labels-help-make-adherence-easier. Accessed 10 May 2019.

Text Detection

Initial research on optical character recognition (OCR) showed that Tesseract OCR was commonly used for text detection. While this algorithm was occasionally effective, it was highly dependent on the image itself. With further exploration, we discovered that an ideal image would have black text on a white background with minimal noise. To do this, we first attempted to create our own image thresholding algorithm that used the RGB of each pixel to determine whether it was closer to black or white, classifying it accordingly. This is shown in the image below in Figure 19. Later, we tried a number of thresholding algorithms, which we compared until we decided to use **Adaptive Gaussian thresholding** to create the most clear black and white image. The advantage of this thresholding is that the algorithm determines the threshold for a pixel based on a small region around it. Hence, we can get different thresholds throughout the same image leading to better results, especially when there is varying illumination.


CVS pharmacy
123 Main Street, Anytown, USA 00000 #0000 cvs.com
Store Corporate Name
TEL 555-555-5555
RX 105004
QTY: 60
REFILLS: 2 by MM/DD/YYYY
PRSCBR: J. Doe
DATE FILLED: MM/DD/YYYY
DISCARD AFTER: MM/DD/YYYY
RPH: J. Smith
MFR: ABC Manufacturer
This is a WHITE, OVAL-SHAPED tablet imprinted with 93 on the front and 48 on the back.

PATIENT
METFORMIN HCL 500 MG
MORNING 1 TABLET
MIDDAY
EVENING 1 TABLET
BEDTIME
▲PHARMACY ADVICE

Patient Name
300 East Street, Anytown, USA 00000
METFORMIN HCL 500 MG TABLET
Generic equivalent of: Glucophage
Take 1 tablet orally 2 times a day

Important Information
• Talk to your Dr. about the safe use of alcohol while taking this drug.
• Severe vomiting or diarrhea may cause dehydration if these occur call MD.
• Drug may cause lactic acidosis. If symptoms develop, seek medical help.
CAUTION Federal law prohibits the transfer of this drug to any person other than the patient for whom it was prescribed.
SCAN TO REFILL

CVS pharmacy
123 Main Street, Anytown, USA 00000 #0000 cvs.com
Store Corporate Name
TEL 555-555-5555
RX 105004
QTY: 60
REFILLS: 2 by MM/DD/YYYY
PRSCBR: J. Doe
DATE FILLED: MM/DD/YYYY
DISCARD AFTER: MM/DD/YYYY
RPH: J. Smith
MFR: ABC Manufacturer
This is a WHITE, OVAL-SHAPED tablet imprinted with 93 on the front and 48 on the back.

PATIENT
METFORMIN HCL 500 MG
MORNING 1 TABLET
MIDDAY
EVENING 1 TABLET
BEDTIME
▲PHARMACY ADVICE

Patient Name
300 East Street, Anytown, USA 00000
METFORMIN HCL 500 MG TABLET
Generic equivalent of: Glucophage
Take 1 tablet orally 2 times a day

Important Information
• Talk to your Dr. about the safe use of alcohol while taking this drug.
• Severe vomiting or diarrhea may cause dehydration if these occur call MD.
• Drug may cause lactic acidosis. If symptoms develop, seek medical help.
CAUTION Federal law prohibits the transfer of this drug to any person other than the patient for whom it was prescribed.
SCAN TO REFILL

Figure 19: Standard CVS prescription after thresholding

We were easily able to read such an image after thresholding. However, our first major caveat was using real pictures taken in imperfect conditions. These pictures are affected more severely by lighting and many of them tend to be curved or rotated given that most of them can be found on pill bottles. The set of images below in Figure 18 are pictures taken by us that show a complete prescription on an actual pill bottle.

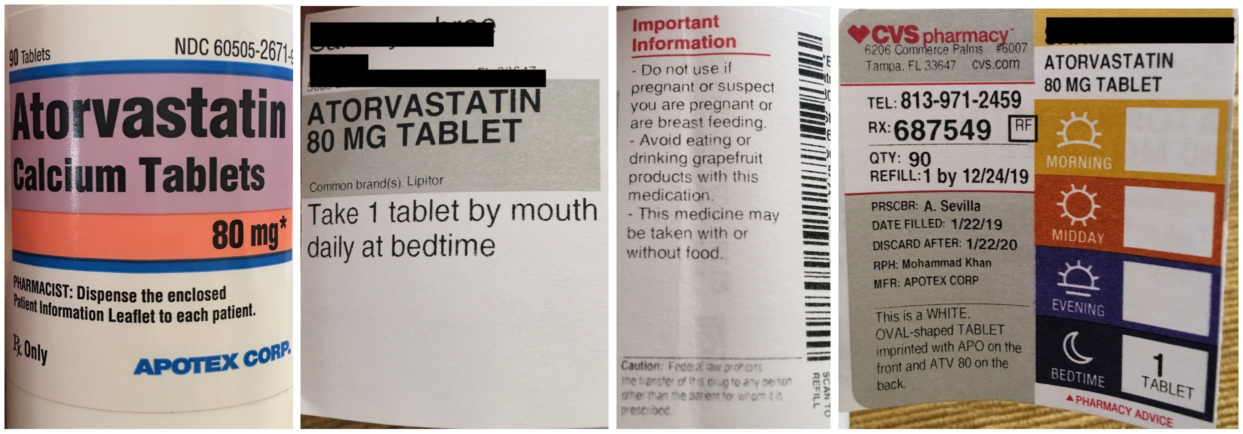

90 Tablets
NDC 60505-2671
Atorvastatin Calcium Tablets
80 mg*
PHARMACIST: Dispense the enclosed Patient Information Leaflet to each patient.
℞ Only
APOTEX CORP.
[redacted]
[redacted]
ATORVASTATIN 80 MG TABLET
Common brand(s): Lipitor
Take 1 tablet by mouth daily at bedtime
Important Information
- Do not use if pregnant or suspect you are pregnant or are breast feeding.
- Avoid eating or drinking grapefruit products with this medication.
- This medicine may be taken with or without food.
Caution: Federal law prohibits the transfer of this drug to any person other than the patient for whom it is prescribed.
SCAN TO REFILL
CVS pharmacy
6206 Commerce Palms #6007
Tampa, FL 33647 cvs.com
TEL: 813-971-2459
RX: 687549 RF
QTY: 90
REFILL: 1 by 12/24/19
PRSCBR: A. Sevilla
DATE FILLED: 1/22/19
DISCARD AFTER: 1/22/20
RPH: Mohammad Khan
MFR: APOTEX CORP
This is a WHITE, OVAL-shaped TABLET imprinted with APO on the front and ATV 80 on the back.
[redacted]
ATORVASTATIN 80 MG TABLET
MORNING
MIDDAY
EVENING
BEDTIME
1 TABLET
PHARMACY ADVICE

Figure 20: Set of images describing a prescription for Atorvastatin

As seen above, the leftmost and rightmost image in Figure 18 appear especially curved while the others are not too clear as the paper is not of perfect quality. Hence, to improve the quality of recognition of text, we switched to **Google Vision**. This accounted for rotation and proved more accurate in reading all kinds of text.

Natural Language Processing

Once we had the text, the challenging part was extracting the relevant information for it and building a electronic prescription for the user. Since the medicine name is the most essential part of a prescription, we decided to extract that first. To do this, we recognized three distinguishing facts about how the medicine name is typically presented. Firstly, it follows drug nomenclature in that it satisfies one of many possible prefixes and suffixes. Secondly, it generally has the generic name of the drug somewhere on the cover meaning that it would be present in the database used for pill identification. Lastly, a pill name would normally be one of the largest words on the prescription. All these are obvious from the four pill bottle images shown above in Figure 18. Since the name would be in the database, first we queried every word to see if there is a match in the database. The code snippet below shows how this is done.

```
URL = "https://rximage.nlm.nih.gov/api/rximage/1/rxnav?name=" + t
r = requests.get(url = URL)
r = r.json()
if r['replyStatus']['imageCount'] != 0:
   potential.append((t, i))
```

We found that while we could identify the pill name through this method, there were often other words such as tablet and drug that also appeared in the database. At first, we considered ignoring words like tablets and drug when we encountered them as they are common on pill bottles but insignificant. However, the database also had results for chemicals like “calcium” and brand names like “Lipitor” which appeared in some of the prescriptions we considered. Hence, we decided that we would also incorporate drug nomenclature to isolate only valid names. The code snippet below shows the list of prefixes, suffixes and word fragments that appear in medicines. It also shows how we compare the suffix of each potential medicine name with the list of suffixes. The same is done for the prefixes and fragments.

```
prefixes = ['ceph', 'tretin', 'pred', 'sulfa', 'cef']
suffixes = ['setron', 'cycline', 'pramine', 'trel', 'asone', 'vir', 'tyline', 'zolam',
'profen', 'zepam', 'onide', 'tretin', 'nacin', 'gliptin', 'cillin', 'dronate', 'phylline',
'zosin', 'parin', 'glitazone', 'lamide', 'mab', 'oprazole', 'semide', 'sartan', 'dazole',
'bital', 'zodone', 'eprazole', 'tinib', 'statin', 'fenac', 'afil', 'caine', 'terol',
'floxacin', 'nazole', 'tadine', 'mustine', 'vudine', 'iramine', 'triptan', 'mycin',
'dipine', 'bicin', 'pril', 'ridone', 'olone', 'thiazide', 'olol']
middles = ['cort', 'tretin', 'pred', 'parin', 'vir']
med_names = set()
if len(potential) > 1:
   for s in suffixes:
       l = len(s)
       for p in potential:
           m = p[0]
           if m[len(m)-l:len(m)].lower() == s.lower():
               med_names.add(p)
```

This method was successful for all medicines that we tried except “Aspirin”. This is because Aspirin is a common brand name and hence does not need to follow drug nomenclature. Hence, as a last resort, if drug nomenclature is not able to isolate one name or the medicine does not satisfy these naming rules, we incorporate the size of each potential medicine name. The code below was used to extract the bounding boxes for every word and find the area associated with each of them. Then, I chose the largest medicine name from the list of potential ones.

```
boxes = [b.bounding_poly.vertices for i, b in enumerate(text_response.text_annotations) if i
!= 0]
box_sizes =
[((i[1].x-i[0].x)**2+(i[1].y-i[0].y)**2)**(0.5)*((i[2].x-i[1].x)**2+(i[2].y-i[1].y)**2)**(0.
5) for i in boxes]
```

Another important aspect of a prescription is how and when the patient is supposed to take the medicine. After viewing numerous pill labels, we concluded that this direction usually begins with “take” and continues for 2 consecutive lines. Since Google Vision provides the bounding boxes of every word identified, we were able to isolate these two lines to process further. Using regex, we broke down these two lines and extracted all the schedule related information. For example, the following code snippet shows how we extracted the number of tablets from the line. Similarly, we extracted whether it was taken by mouth, the timing in relation to meals, and which times of the day.

```
if "tablet" in schedule:
   reg = '(\w+)\s+tablet'
   m = re.search(reg, schedule)
   m2 = words_in_sched.index(m.group(1))
   words_in_sched[m2] = ""
   words_in_sched[m2+1] = ""
   sched['No. of tablets'] = m.group(1)
```

While looping through the rest of the text in the prescription, we were able to extract more information using the following heuristics:

1. If a word matched any word in the list of pharmacies in the United States, it was chosen as the pharmacy name
2. If it was equivalent to “mcg” or “mg”, we used regex to find the preceding word and treated it as the dosage of the medicine
3. If the word contains “tablet” or “capsule” it is classified accordingly under “type” of pill
4. If it matches one of the patients names (stored in the app)

Figure 19 shows two images from pill bottles that contain a majority of the information discussed above. The first label is for the drug Atorvastatin. The dosage is 80 mg and it comes in tablet form. In terms of schedule, it directs the patient to take it once a day by mouth at bedtime. The second label is for Aspirin. The dosage is 81 mg in tablet form. It was prescribed once a day with meals, also by mouth. As we extracted information from the label, we added it to a dictionary that is included below the images.

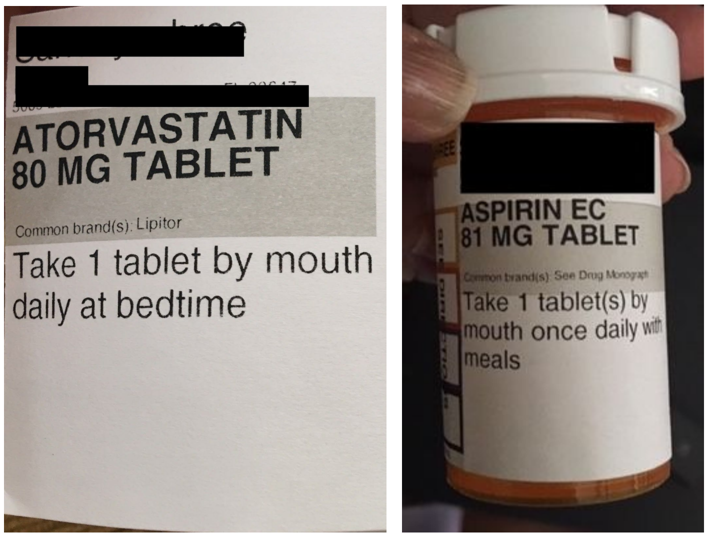

[redacted]
[redacted]
ATORVASTATIN
80 MG TABLET
Common brand(s): Lipitor
Take 1 tablet by mouth daily at bedtime
[redacted]
ASPIRIN EC
81 MG TABLET
Common brand(s): See Drug Monograph
Take 1 tablet(s) by mouth once daily with meals

Figure 21: Pill labels for Atorvastatin and Aspirin

The dictionaries embedded below are samples of the output for two different pills shown in Figure 19 and described above.

```
{'Dosage': '80 MG', 'type': 'Tablet', 'Schedule': {'How': 'by mouth', 'No. of tablets': '1',
'When': 'daily at bedtime'}, 'medicine_name': 'ATORVASTATIN'}
{'Dosage': '81 MG', 'type': 'Tablet', 'Schedule': {'How': 'by mouth', 'No. of tablets': '1',
'Meals': 'wih meals', 'When': 'once daily'}, 'medicine_name': 'ASPIRIN'}
```

Hence, we are able to extract all the necessary information almost perfectly using various heuristics based on our knowledge of pill labels and drug nomenclature.

Extensions

Overall, we hope to extend the pill identification and prescription analysis algorithms by integrating them into the app such that we can input images of pills and bottle labels in a user friendly interface. Aside from this, we had some improvements that we would like to match to each project.

Pill Recognition

The first improvement could have been to the kind and quantity of data used to train the model for imprint detection. There are a number of other accessible databases that we could have used to supplement us with more data regarding medicines and their features. This would ensure that we are covering almost all potential pills, including over the counter and prescription drugs. Additionally, while the images from the database were effective in training the model, a better alternative would be images clicked by people that are more affected by lighting, different angles, and noisy backgrounds. As the dataset increases, precision also generally increases. Since this dataset is not easily available, we could possibly collect the data from users on the app who upload pictures of pills they want identified.

Another issue with imprint detection was that we were unable to account for pills that had curved text, as shown in the example below. This is because we could not include labels of letters that appeared both sideways and straight it would confuse the model. To fix this in the future we could rotate the image and read text at 30° angles until a full rotation is achieved.

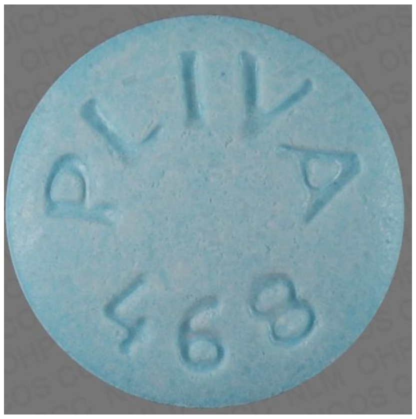


Figure 22: Image with rotated text that says “PLIVA” and “468”

Additionally, since the two models were trained separately (for numbers and letters), it was possible that the models could identify the same engraving differently. For example, an “I” could be mistaken for a “1” and thus be recognized differently in both models. To handle this error, we would have to compare the bounding boxes returned by each model. If there is an overlap of 75% or greater between the two bounding boxes, we would choose the model that returned a higher probability. Another improvement that we can do is based on an observation we made that letters and numbers are mostly followed by letters and numbers respectively (in case of grouped imprints). Hence, if we see an ‘I’ preceded or followed by numbers, we can safely assume it is ‘1’ and vice versa. This is important since there are a significant number of pairs that can be mistaken by the model, since our letter and number models are seperate) and applying this would improve the model accuracy. Another possible solution for this would be to train a model for each conflicting pair. This would give us a sure result between the conflicting labels.

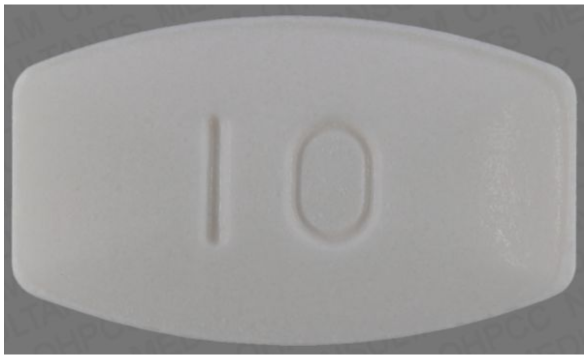


Figure 22: ‘1’ recognized as ‘I’ by the algorithm. Based on the context, we can guess 1

Lastly, we can improve the shape and color algorithm. To improve the shape algorithm we can make more templates of the pills and then use K-Nearest neighbours to identify what shape does the pill belong to. This improves the accuracy by allowing us to consider different templates for the same image (such as oval, which has different types of templates since it is so diverse). We could also train a model to recognize shapes. To improve the color algorithm, we can find the cluster centers for all pill-images ourselves, get their color names and then make a database of our own, classifying pills by color. This would help since the colors given by the database are often labelled wrong. Hence, classifying them ourselves would greatly improve our output.

Text Extraction

The first improvement we intend to do with text extraction is spell checking and common word removal. Spell checking is necessary as our algorithm hunts for specific terms in the text. Since there is no way to predict what Google Vision will read and output, we need to get the spellings right for the words for our queries to work smoothly. Alongside this, sometimes names of pills have common words in them, such as patient or drugs. We would improve in the algorithm to remove such common words from the text or at-least not count them while querying the database for the pill name and information.

Another improvement to this algorithm is to make add code to return information about the pill. This is important as the user can then directly access information about the drug he/she is taking. Also, since pill-label extraction is going to be added to our app, providing users information about the pill they are taking increases convenience for them and also allows them to understand what medicines they are taking and what their side effects are.

Citations

1. “ScriptPath Prescription Labels Help Make Adherence Easier.” CVS Health Payor Solutions, https://payorsolutions.cvshealth.com/insights/scriptpath-prescription-labels-help-make-adherence-easier. Accessed 10 May 2019.
2. Template Matching — OpenCV-Python Tutorials 1 Documentation. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_template_matching/py_template_matching.html. Accessed 9 May 2019.
3. OpenCV: Contours : Getting Started. https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html. Accessed 9 May 2019.
4. RxImage API. https://rxnav.nlm.nih.gov/RxImageAPIs.html#. Accessed 10 May 2019.

Appendix

Gaussian Mixture Model(GMM):

A gaussian mixture model is a probabilistic model that assumes that all data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. In a way, it is like generalizing K-Means Clustering, incorporating information about the covariance of the structure of the data.[5]

Graph Cut:

In context of the grabcut algorithm, the graph cut algorithm works by generating a graph using the foreground and background pixels as nodes. Two more nodes are added, the source node, to which all foreground nodes are connected, and the sink node, to which all the background pixels are connected. The edges for the foreground and background node have weights defined by the probability of a pixel being in the foreground or background respectively. The algorithm then segments the graph into two, separating source and sink node using a cost function calculated by the sum of all weights of the edges. Post segmentation, the pixels connected to the source node are labelled as foreground and those connected to sink node are labelled as background. The process is repeated as specified by the user.[6]

[5] *2.1. Gaussian Mixture Models — Scikit-Learn 0.20.3 Documentation*. https://scikit-learn.org/stable/modules/mixture.html. Accessed 8 May 2019.
[6] "Python | Foreground Extraction in an Image Using Grabcut Algorithm." *GeeksforGeeks*, 28 Dec. 2018, https://www.geeksforgeeks.org/python-foreground-extraction-in-an-image-using-grabcut-algorithm/.