Advanced Predictive Modeling Project

Group: Brooks Beckelman | Dallas Griffin | Davis Townsend

Estevan Gonzales | Sean Kessel | Zackery Bilderback

Abstract

In this paper, we explore the various tools and models available for image recognition as a means to address a real business problem faced by a local Austin business. Popspots – a startup specializing in digital point-of-purchase (PoP) advertisement systems – is interested in developing a new add-on service: automating the the compliance and out-of-stock checking of racks in check-out lanes for their clients in the grocery industry. Our project aims to use an archive of images captured by Popspots devices to implement an SKU identifier as a first step towards meeting Popspots business objective.

Over the course of this paper, we investigate many of the most popular methods and tools available in the realm of computer vision. These include techniques in template matching using the popular tool OpenCV as well as an application of the deep learning framework Faster R-CNN on Caffe. We will conclude our paper with the challenges we encountered, our key findings and lessons learned, and a discussion of the additional steps necessary for Popspots to implement a production-ready stock-check system.

Background

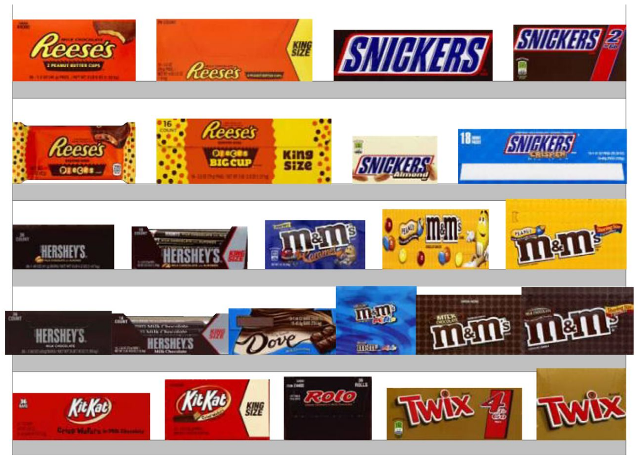
Popspots Overview

Popspots is an early-stage startup based in Austin, TX. They work with grocers & retailers to install tablets in check-out aisles then sell advertisements (still images and/or videos) in 7-second increments on the marketplace. Using proprietary object-detection software, the tablets only display advertisements when a customer is standing in front of the tablet, assuring suppliers that each advertisement is as effective as possible at generating customer attention and follow-through. Furthermore, Popspots offers a dynamic ad marketplace, allowing advertisers to bid on the specific regions, retailers, and times most important to their business.[1]

Business Problem

The PoP rack is a major revenue generator for retailers – netting on average $24,000 in revenue per rack per annum. However, these simple money makers are susceptible to outside conditions that can drastically affect each rack's value to the retailer. It is estimated that up to 10% of a single rack's revenue is lost due to non-stocked product. Retailers also face a significant challenge keeping their PoP racks in compliance with product placement agreements signed with vendors.

Sample PoP Rack Compliance Schematic



[1] https://www.popspotsadvertising.com/

PopSpots hopes to address both of these issues by expanding their existing resources to automate the current error-prone, time-consuming processes of stock and compliance checking. An image recognition system would be able to visually analyze a rack, report which products are out of stock or nearing depletion, and (hopefully) trigger an automatic restock order for the appropriate products. Additionally, this system would ensure all product placement agreements with vendors are in compliance as well as provide valuable analytics for optimizing product placement and mix.

Popspot's current generation of tablets are equipped with cameras pointed directly at racks used primarily to detect when customers are in the checkout aisle. Thus, an automated stock-service would incur few costs beyond initial software development and is well-aligned with the company's long-term strategy of leveraging innovative technology to create value at retailers' points of purchase.

Data Scope

Popspots has provided us with around 1,000 images captured by their tablets identical to those that will be used for the new image-detection system. To capture some variability, the images were taken from 25 different racks at 10 grocery stores at a rate of 10 images per day, over 10 business days. These 1,000 images equate to 30,000 product instances and over 300 million raw pixels(!).

To train the models, an online scraper was implemented to automatically download product images at varying points of view and rotations. Due to time limitations, we had to limit the number of product images considered for identification to around half a dozen.

OpenCV

To familiarize ourselves with the world of computer vision, we first explored OpenCV. Due to the complexity of our image recognition problem, we assumed we would ultimately require more sophisticated approaches like convolutional neural nets. However, OpenCV provided a relatively simple introduction into this emerging field - a primer that would allow us to gain a rough understanding of how to begin. In the following section we cover what we learned by implementing OpenCV, the results we obtained from using this software, as well as a summary of why OpenCV did not meet our requirements for this particular project.

History

OpenCV was initially developed by an Intel Research Team in Russia in 1999 to create an open-source platform written in C++ that would provide a common infrastructure for computer vision applications and accelerate the use of machine perception in commercial products.[2] We were initially attracted to OpenCV due to its open source nature, relatively easy setup, and its great interface with python environments, including Jupyter Notebooks.

Image Preprocessing

Before we started trying to detect the candy bars in our images we needed to do some “image data prep”. Since the cameras on Popspots’ tablets were only used to detect someone’s presence, the images that were supplied to us were understandably not high-resolution. A sample image is shown below:

[2] https://en.wikipedia.org/wiki/OpenCV

In a production system, you'd need a standardized ETL process to clean images up and crop out the portion of the image where the rack is in order to eliminate the chance that the training image would be identified in an irrelevant portion of the image. For our purposes, we did this manually for a handful of pictures to mimic what we assumed would ultimately need to be done later on:

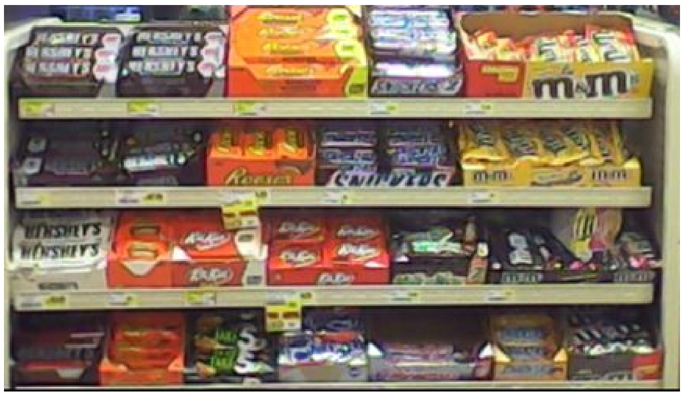
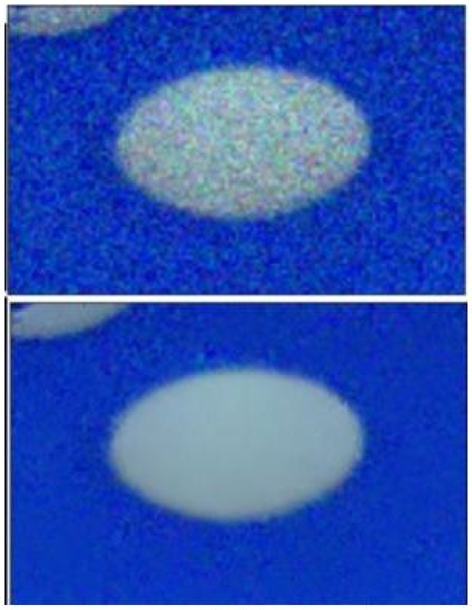


Image Denoising

The next thing we did to prepare our images was use Image Denoising[3] to smooth our images. This process works via Non-Local Means Denoising, which takes a small window around a pixel, searches for similar windows in the image, averages all the windows, and replaces the pixel with the result. An example of an image pre and post image denoising is shown below:

[3] http://docs.opencv.org/trunk/d5/d69/tutorial_py_non_local_means.html



We tried image denoising with each of our approaches and found it usually didn't make that much of a difference. When it did, the result was often worse than without it. We assumed this was due to the fact that image denoising sacrifices some of the image's detail which had an outsized impact on our already small photos of individual products.

Edge Detection

Next we implemented the Sobel - Canny edge detection algorithms in order to clean the images, as we theorized the template matching algorithm (explained below) would produce more accurate results. After gray-scaling the image so that each pixel is represented by an integer between 0 and 255 inclusive and running it through a gaussian blur, the image was passed through the Sobel operator. The operator runs the kernels shown below over the image to transform them.[4]

Vertical Kernel:

-1	-2	-1
0	0	0
+1	+2	+1

Horizontal Kernel:

-1	0	+1
-2	0	+2
-1	0	+1

[4] http://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm

One way of interpreting this kernel convolution it is to say that it displays how contrast changes over the image. Note that these kernels are, in essence, taking a “pseudo-derivative” of the gray-scaled image in the horizontal and vertical direction respectively. This is not meant to imply that they are taking an actual derivative over the gradient, as adjacent columns in the vertical kernel and adjacent rows in the horizontal kernel influence the Sobel operator’s output. The concept, however, of a derivative being a measure of the change in magnitude of a function (in this case an image) over a small distance is reflected in this operator.

After this process is complete, each cell (which represents each pixel location) will have two integers: the Sobel operator output from the x-direction kernel and from the y-direction kernel. These two integers can be transformed into one float representing the combined magnitude of the x and y direction operator outputs, and another float representing the angle from the horizon of the two Sobel outputs. The results of the functions given below will be the output of the Sobel operator.

$$Magnitude: \quad G_{x,y} = \sqrt{G_x^2 + G_y^2}$$

$$Angle: \quad \theta = arctan\left(\frac{G_y}{G_x}\right) \approx \left(\frac{G_y}{G_x}\right) - \frac{\left(\frac{G_y}{G_x}\right)^3}{3} + \frac{\left(\frac{G_y}{G_x}\right)^5}{5} - \frac{\left(\frac{G_y}{G_x}\right)^7}{7} + \frac{\left(\frac{G_y}{G_x}\right)^9}{9} - \frac{\left(\frac{G_y}{G_x}\right)^{11}}{11}$$

Next, the Sobel operator output is run through a Canny edge detector algorithm. This is a two stage algorithm that focuses on outputting an image containing an edge which has the width of 1 pixel. It also aims to ignore edges produced by glare, reflections, shadows, and noise as these aspects of the image are generally less important in object detection algorithms.[5]

The first step of the Canny algorithm is to find all the local magnitude maxima of the Sobel outputs. Notice the transformations done near the end of the Sobel algorithm (the above equations) all yield positive magnitudes, so the focus can be solely on local maxima. Here, the goal is to find which pixel best approximates the edge of the object, so comparisons are to be made perpendicular to the edge of concern for each pixel. Note that this is necessary as otherwise the algorithm would be comparing which pixel along one edge has the higher magnitude which would not lead to the desired output. If the angle float, from the Sobel

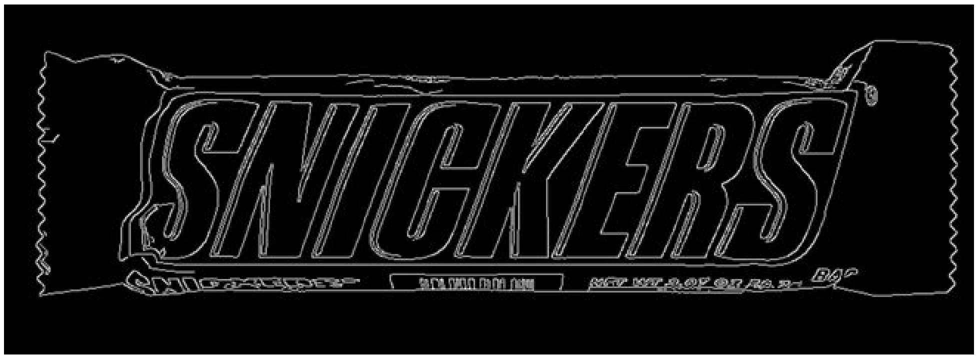
[5] http://homepages.inf.ed.ac.uk/rbf/HIPR2/canny.htm

operator, in the cell in question is closest to zero, the edge runs horizontally so cells in a certain, specified neighborhood around the cell in question are compared only in the vertical direction. If the angle is closest to pi/4 radians, the edge is running diagonally, so cells in a small neighborhood neighborhood around the cell in question are compared only in the orthogonally, diagonal direction (e.g. for an edge angle that rounds to pi/4 radians, comparisons would be made in the 3*pi/4 radians direction). The process is identical for all angles in question, where each angle is rounded to the nearest divisor of pi/4 radians and compared to cells lying orthogonal to that rounded angle.

The output of this first step are all the local maxima, defined as such with respect to the cells in the orthogonal direction of the edge. All cells that are not local maxima have been set to zero, while all local maxima still contain the magnitude of their Sobel outputs. The second step of the Canny algorithm is to determine which of these local maxima should remain. It does this by a process known as Hysteresis Thresholding. Here, an upper and lower threshold are set by the user and the following rules are applied:

- Any cell that has a magnitude above the upper threshold is automatically kept.
- Any cell that has a magnitude below the lower threshold has it’s magnitude automatically set to zero.
- Any cell that has a magnitude between the two thresholds is kept only if it is directly adjacent to a cell that is either above the upper threshold or within a line of cells kept because there exists at least one cell in the line above the upper threshold and all other cells within the line are above the lower threshold.
- Any cell that has a magnitude between the two thresholds that does not appear next to a cell above the maximum threshold, nor next to a cell that extends from a line of cells described above, has its magnitude set to zero.

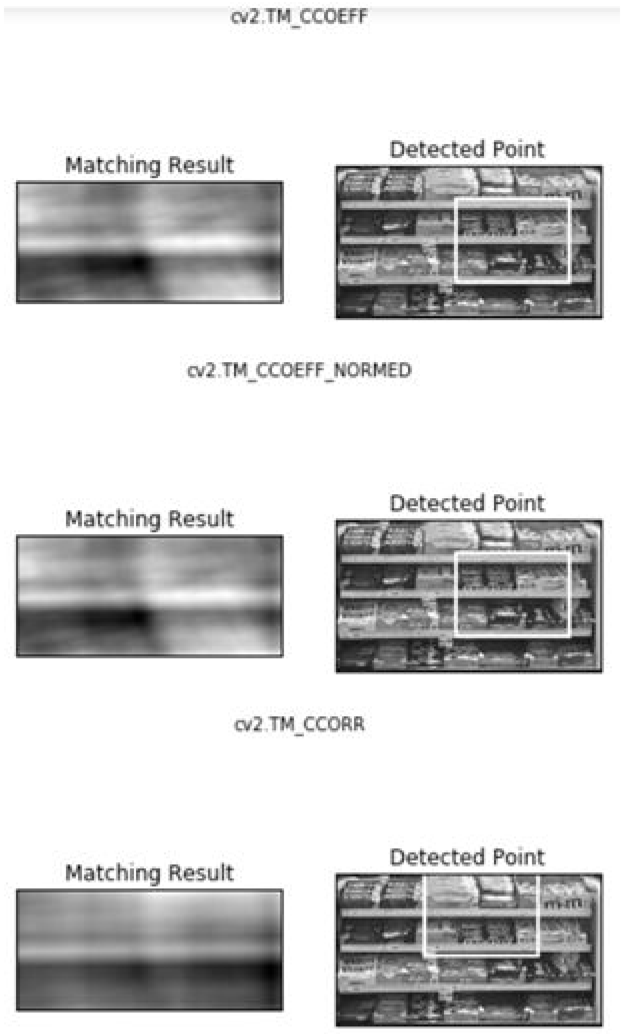
Below is an example of a resulting image after being passed through Sobel and Canny:



The Sobel and Canny edge detection algorithms can be useful preprocessing steps to improve results of image detection algorithms. We initially attempted using a template matching algorithm on images that had been gray scaled and gaussian blurred. Subsequently, we ran these images through the Sobel and Canny algorithms then used these preprocessed images to run the template matching algorithm. Additionally, we later used the Sobel and Canny algorithm to preprocess images for feature selection. These approaches, and and their results, are described in subsequent sections.

Template Matching

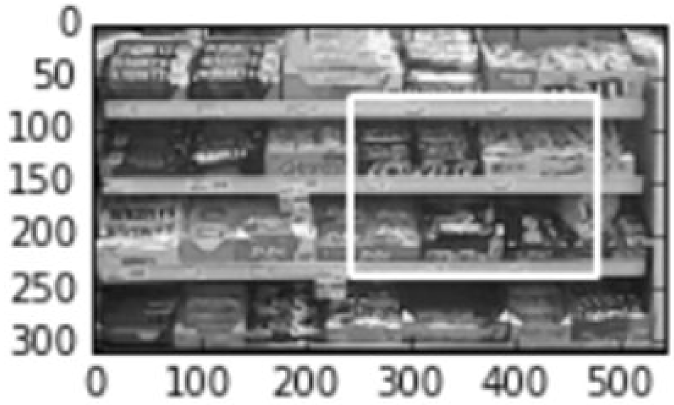
The first approach we took to the object detection and classification problem was template matching. Template matching slides a template image over the input image and compares the template and the patch of the the input image under the template image. The closeness of the match is then evaluated by correlation coefficients, the squared difference or a normalized version of either or both methods. To test each method, we set up a loop then visually compared their results. To visualize the match quality for each method, we plotted a box around the x,y coordinates indicating the location of the match:



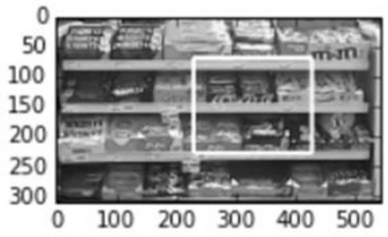
You can see from the above picture that we weren't getting very good results from this approach. We initially attributed the poor performance of this method to template matching inherent susceptibility to different scale & rotation of images.

To address the issue of scale-invariance we used the multi-scale template matching trick outlined in a post about matching logos associated with the Call of Duty video game series.[6] Essentially this trick allows you to do template matching at different scales by looping over the input image as it gets progressively smaller in size and keep track of which match had the largest correlation coefficient. Once you have this, you can convert back into the desired scale and draw a rectangle around the match as before. This resulted in a mediocre result as seen below, so we moved onto trying to fix the issue with rotation-invariance.

[6] http://www.pyimagesearch.com/2015/01/26/multi-scale-template-matching-using-python-opencv/



We found two workarounds to potentially address this problem. First, we could make a folder with a bunch of pictures of the candy bar at various rotations using our webscraper and hope that one of the rotated templates would be similar to the rotation of the candy bar on the actual rack in the image. The second would be to have a single or small subset of candy bar images and artificially rotate them using OpenCV. These two approaches are similar in that you must iterate through an additional subset of images and keep track of where you had the best match. We chose to go with the first workaround since it was easier to implement and leveraged the pictures that we could from our webscraper.



```
C:\Users\Owner\Desktop\Python\AdvPredModeling\Project\snickers\snickers_23.jpg
```

The result we got was a match that had iterated through a folder of snickers bar pics, as well as doing the multi-scale template matching, and output the best match as before with a rectangular box around it as well as printing out the template candy bar pick that gave the best match. Despite our best efforts, results were similarly mediocre.

In summary, despite our attempts to work around the big issues surrounding template matching, we still couldn't get good results from matching online pictures of candy to find the candy bars in our rack images. We did test this method using the cropped portion of the candy from its own rack as well as from other racks. As expected, the method worked perfectly when using the cropped picture from its own rack, but not from any others. Unfortunately, due to the nature of our problem this would not work since we want to automatically detect the candy bars, not crop them out and then find them in the same picture. This seems in fact to be the crux of the problem with template matching is that it’s useful for finding a template cropped from the original picture within that picture. The classic example used for introductory template matching using python is finding a picture of waldo in a where’s waldo picture.[7] This is something inherent in this method that cannot be overcome with any easy workaround. Therefore, we decided to explore more powerful algorithms that effectively dealt with the problems of scale and non-affine transformations such as rotated images, and that could be used for feature matching later on.

Template Selection

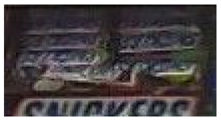
The success of template matching is often contingent upon the template image used. We tried several different templates in an order to optimize our results. The best results, discussed in the previous section, were obtained using template images that we scraped from the web. To do this, we first compiled a list of all of the candies and other SKUs that we were interested in gathering templates for. We were then able to code an image compiler that loops through this list, defines the organization (i.e., how the file will be stored in the directory) and the search query (i.e., what the actual search query will be), runs the query through the Bing search engine, pulls in images from the search, and stores the images in the defined directory. A couple of the results for Snickers are shown below.

When performing the template matching, we would loop through all of the Snickers templates that were pulled from the web and find the best match on the rack image. Out of all of the

[7] http://machinelearningmastery.com/using-opencv-python-and-template-matching-to-play-wheres-waldo/

matches, the best one was selected and displayed to give us the output shown in the previous section. Our efforts were largely unsuccessful due to the low resolution and blurriness of the rack images. The actual Snickers boxes on the racks were very dissimilar to the templates that we were using.

In an attempt to overcome this issue, we tried a different approach to template selection. Rather than use the clean images that we scraped from the web, we decided to try using cropped portions of the rack images as templates. For example, we would go through several of the rack images, manually crop the Snickers box from the rest of the image, and use the cropped image as the template. This resulted in templates such as the one shown below.



As expected, we found this method to be very effective when the template was used on the rack image that it was cropped from. However, when applying the templates to other rack images, the results were far worse than when we used the images from the web. This could likely be attributed to the low resolution of the images, the fact that many of the SKUs were cut off or distorted, and the fact that the same SKUs often looked very different on different racks.

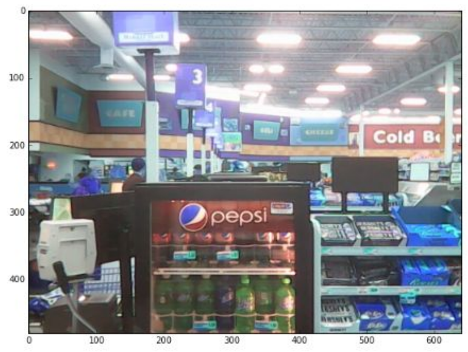
Template Matching with Color

Unsatisfied with the results that were produced from black and white template matching, our next approach was to incorporate color images for the templates to match to.

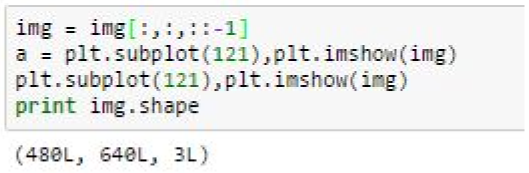
Transformation of Color Spectrum

OpenCV automatically imports colored images in an altered color spectrum (instead of the usual RGB color space, OpenCV defaults to using BGR). Objects that should be red appear blue, objects that should be blue appear red. This has a particularly harsh effect on the entire image as pretty much any pixel that isn’t pure green gets shifted to an incorrect hue.

```
(<matplotlib.axes._subplots.AxesSubplot at 0x43eb438>,
 <matplotlib.image.AxesImage at 0x71e8ef0>)
```

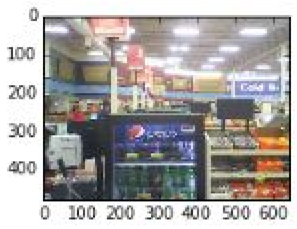


To remedy this obstacle, we discovered that inserting the following code shifts the image to its appropriate color space.



```
img = img[:,:,::-1]
a = plt.subplot(121),plt.imshow(img)
plt.subplot(121),plt.imshow(img)
print img.shape
```

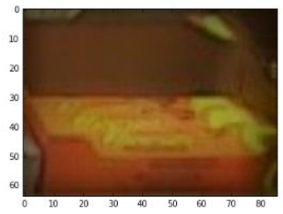
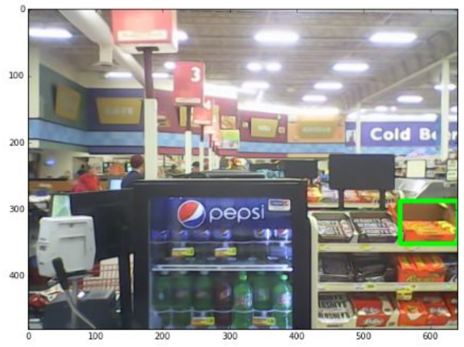
```
(480L, 640L, 3L)
```



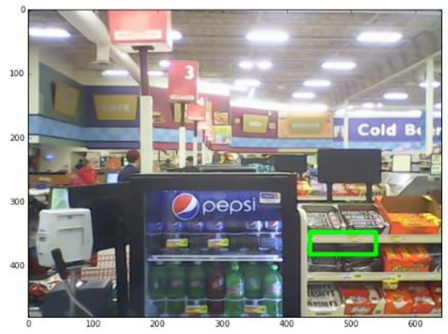
However, like most things worth learning, it is not that straightforward. This small change caused a cascading effect down our original black & white template matching code, that required minute, unexpected changes throughout.

Implementation of Color-Based Template Matching

After exhaustive research and code fixes, we finally produced a working model for color-based template matching. Our initial results seemed promising; like the previous b&w template matching, the new model was able to accurately identify a product when the template was cut exactly from the referring image. It even successfully handled slight color changes in the image (shown below). Notice how the template is slightly darker and faded compared to the rack picture.

As with the b&w model, the model became less accurate once template images cut from other rack images were used. In fact, using color images did not seem to increase or decrease the effectiveness of template matching at all. As previously stated, this could be due to a variety of issues including image noise and resolution.



ORB (Oriented FAST & Rotated BRIEF)

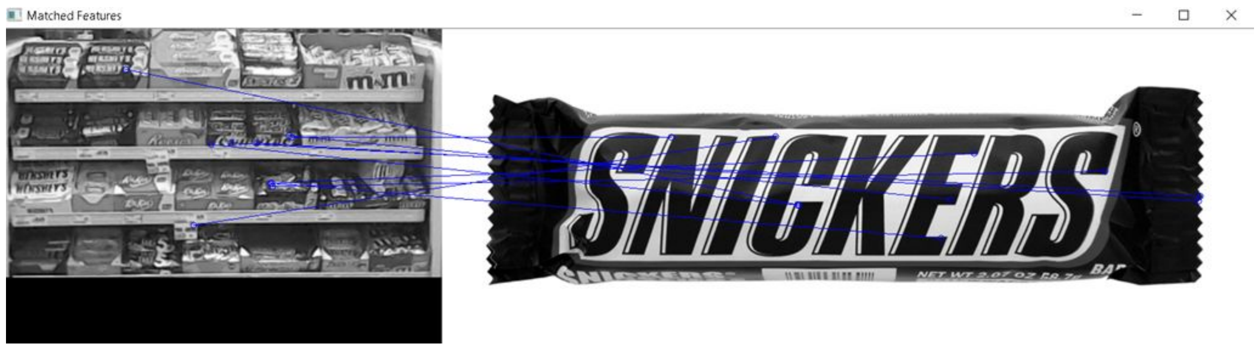
At this point, we realized using standard OpenCV template matching would not be as beneficial as we had hoped. The next approach we explored was using the ORB[8] algorithm developed by OpenCV labs, which is an efficient alternative to the SIFT[9] and SURF[10] algorithms which are

[8] http://www.willowgarage.com/sites/default/files/orb_final.pdf
[9] http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf
[10] http://www.vision.ee.ethz.ch/~surf/eccv06.pdf

patented, and therefore not available freely for commercial use. ORB is a fusion of the FAST[11] (Features from Accelerated Segment Test) key-point detector algorithm and the BRIEF[12] (Binary Robust Independent Elementary Features) descriptor algorithm, with many modification for improved performance. This algorithm essentially finds “keypoints,” meaning corners usually, via the FAST algorithm, and then the BRIEF algorithm uses the area surrounding these key points to construct descriptors. The usefulness of this algorithm comes in when you run it on 2 images, because you can compute the “match”, essentially just a measure of similarity, between the 2 images based on these descriptors. Our implementation of this will be discussed in the following section. While ORB is a powerful algorithm, the authors note in their original paper that while the method account for scale-invariance holistically, it may not adequately account for it on a per keypoint basis.

Feature Matching with ORB

After using ORB to find our keypoints and descriptors we used a Brute-Force Matcher[13] to match the descriptors in our candy bar and rack images using the Hamming Distance as the measure of similarity. We drew heavily from a stackoverflow post[14] in order to draw the matches in our images since OpenCV v2.4’s native draw matches function does not interface with Python 2.7. By using this method we were able to get a visual representation of what our matches look like, denoted by the blue lines and circle. This picture shows the ten best matches:



We can see that ORB with feature matching performs much better than template matching at finding the snickers box, but there are still several mismatches. We attempted to use

[11] https://www.edwardrosten.com/work/rosten_2006_machine.pdf, https://arxiv.org/pdf/0810.2434.pdf
[12] http://www.robots.ox.ac.uk/~vgg/rg/papers/CalonderLSF10.pdf
[13] http://docs.opencv.org/trunk/dc/dc3/tutorial_py_matcher.html
[14] http://stackoverflow.com/questions/20259025/module-object-has-no-attribute-drawmatches-opencv-python

Homography[15] in addition to feature matching to outline the object in the picture but were unable to get this functionality to work. We doubt that it would have been able to clearly represent the object anyways, since there was still a substantial number of mismatches for every combination of candy bars and rack we sampled.

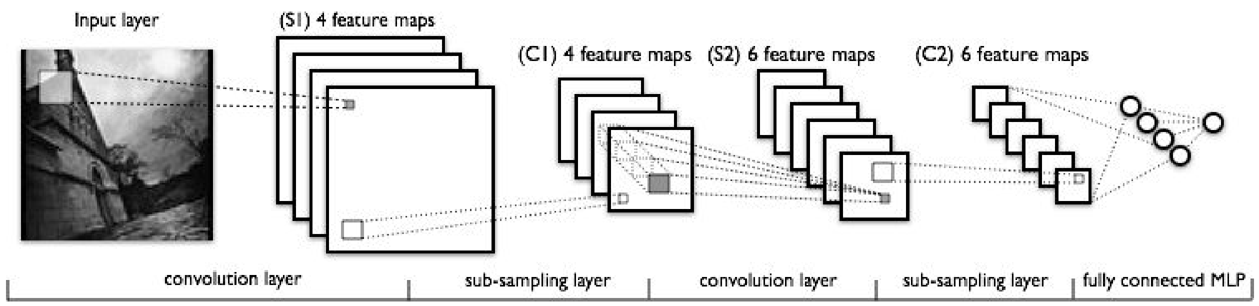
Summary of Findings from OpenCV

Even after mixing and matching countless image transformation techniques (i.e. denoising and edge detection) and image detection methods (i.e. Template Matching and ORB Feature Matching), we were still unable to consistently detect a product on any rack other the one we used to train it. This is because these methods are dependent on careful feature selection and tuning, which severely limits their ability to deal with the variability inherent to the products on a check-out rack. We needed an approach able to learn the defining features of each product, rather than being dependent on us to supply all possible variations.

Deep Learning

To overcome the limitations of template matching, we next turned to one of the more popular and proven tools in computer vision: convolutional deep neural networks (i.e. "deep learning"). Convolutional neural networks are loosely designed to emulate the behaviour of a visual cortex, which allows them to perform quite well on visual recognition tasks. Unlike template matching - which requires clearly defined image features and a means to account for image variability in a class - deep learning automatically performs much of the feature engineering. Thus, deep learning (in theory) simultaneously reduces the effort required to prepare the images and improves results. A typical CNN consists of convolutional, pooling, ReLU, fully-connected and loss layers that collectively form a sort of pipeline encoding the various properties of the images and ultimately outputting classification probabilities[16] [17].

[15] http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_feature_homography/py_feature_homography.html
[16] https://en.wikipedia.org/wiki/Convolutional_neural_network
[17] https://deeplearning4j.org/convolutionalnets.html



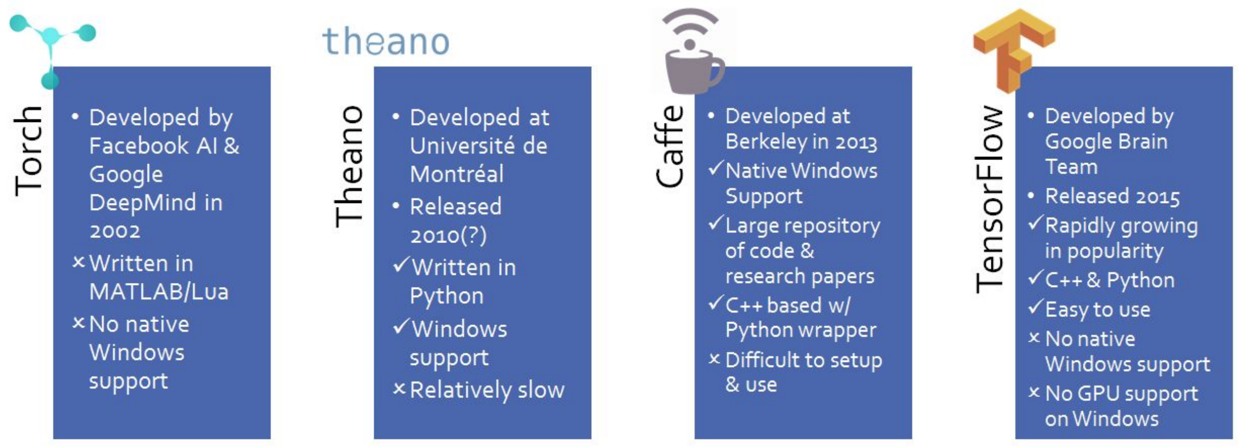
For our problem, the neural net would have to perform two sequential tasks:

1. *Feature Detection:* The layers associated with this task would be responsible for disregarding background noise and segmenting the rack image into various object proposal regions (i.e. the different products).
2. *Feature Classification:* The object region proposals would then be fed to a series of layers responsible for classifying (or rejecting) the objects as their respective product.

Once we had developed a conceptual understanding of deep learning, we set to evaluating which of the various deep learning tools and frameworks was best fit for purpose.

Caffe

To determine which deep learning tool was best suited for our problem, operating systems, and skillset, we evaluated many of the most popular deep learning frameworks in use today[18] [19]:



Torch
- • Developed by Facebook AI & Google DeepMind in 2002
- × Written in MATLAB/Lua
- × No native Windows support

theano

Theano
- • Developed at Université de Montréal
- • Released 2010(?)
- ✓ Written in Python
- ✓ Windows support
- × Relatively slow

Caffe
- • Developed at Berkeley in 2013
- ✓ Native Windows Support
- ✓ Large repository of code & research papers
- ✓ C++ based w/ Python wrapper
- × Difficult to setup & use

TensorFlow
- • Developed by Google Brain Team
- • Released 2015
- ✓ Rapidly growing in popularity
- ✓ C++ & Python
- ✓ Easy to use
- × No native Windows support
- × No GPU support on Windows

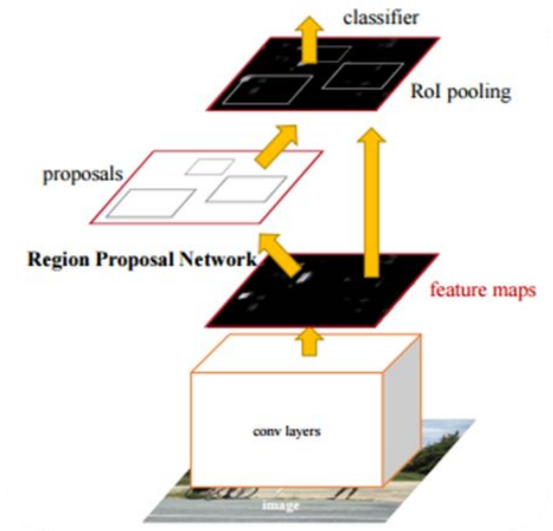
[18] https://github.com/zer0n/deepframeworks

[19] https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software

Ultimately, we settled on Caffe because it can be natively installed - with a Herculean effort - on Windows, has a Python wrapper and has a large repository of code and research papers to draw on. This included code for our deep learning model of choice: Faster R-CNN.

Faster R-CNN

Faster R-CNN is a model developed by Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun[20]. It performs region proposal (i.e. object detection) and object classification in a single, linked convolutional network using RoI pooling. This sharing of information between region proposal and classification improves accuracy while dramatically reducing run-time to near real-time.



While Faster R-CNN is hypothetically a state-of-the-art model for multi-class, multi-objects problems like ours, we have unfortunately not been able to successfully implement the model due to operating system and training data issues outlined in the challenges section. However, we're confident that - after countless hours of reading research papers and pursuing dead-ends - we've finally arrived at the right solution for Popspots business problem.

Challenges

Like most projects, our team experienced a variety of setbacks. Some could be overcome through research and better understanding of the tools we were working with, while others were

[20] https://arxiv.org/abs/1506.01497

out of our control. For the purposes of explanation, we have partitioned these challenges into two groups: Learning Challenges (those we faced as we were building our models, and prohibited us from further advancement) and Implementation Challenges (those mostly out of our control that we viewed as potentially detrimental to the success of any image-based recognition system).

Learning Challenges

Most of our development problems arose from the fact that none of us had prior experience in computer vision which, somewhat expectedly, has an exceedingly gradual learning curve. Before we could even explore the tools available to us, we first had to install various programs and libraries on our computers. Unfortunately open source and user-friendliness rarely go hand in hand. From outdated GPU cards, outdated software, and everything in between, a good chunk of our time was allocated towards installing everything. Nobody in our group is a heavy Mac or Linux user, but it seemed that most of the software was built primarily for use on these systems.

Implementation Challenges

However, once the software was set up and we could finally begin to use it, we were met with additional challenges that would need to be addressed before the system could be implemented. One of the most concerning is incorrectly placed cameras. For the best results, images of the racks must show the full length of the rack. This issue could easily be fixed if each rack's position was marked on the floor so a store employee could place it back in the right spot. We also saw poor performance from some of the models due to the quality of rack images. Some of the images were blurry or had glare to a point which obstructed the models from working properly. Popspots plans to upgrade the camera and re-arrange the positioning of their tablets early next year, which might reduce the impact of this problem. Additionally, customers often picked items off the racks then returned them to different spots. For instance, in the image below we can see a Reese's placed in a M&M's box:

For our testing purposes, we manually excluded images that were not up to par. However, this really can't be done once the system is in place. A bad image would result in inaccurate or unobtainable results.

KEY FINDINGS & LESSONS LEARNED

Template matching techniques proved unable to deal with the natural variability of the rack images. However, more comprehensive models like Faster R-CNN are well-equipped to handle this variability, provided enough training data is available. An unrelated but equally critical lesson learned is that if you want to be a productive data scientist in the realm of computer vision, you better have a Linux or OS X operating system.

While Deep Learning approaches mitigated some of the issues we faced with template matching, it introduced its own challenges. As it turns out, Faster R-CNN is also written for Linux machines and is exceptionally difficult to install on Windows. More critically, the 5 or so images per class falls well short of what we'd need to implement a deep neural network without overfitting. For the 30 racks in our analysis, there are around 100 unique SKUs. To implement a deep learning model with a high degree of predictive power, we would likely need over 50 unique photos of each SKU on the rack to avoid overfitting. For our analysis, the average SKU - even for the most common products - appeared only 4 to 6 times. Finally, training a neural net to accurately pick-out multiple objects within an image requires training images with annotated bounding boxes similar to the style used for the PASCAL VOC dataset[21], rather than our approach of cropped images.

Conclusion

Faster R-CNN should, at minimum, be able to locate and classify products from the rack image, which would allow Popspots to ensure product placement on the racks is in compliance with the retailer's vendor agreements as an add-on service. With a large enough training set of images, future models should also be able to classify products as in-stock or out-of-stock with a decent degree of accuracy.

Over the winter break, we plan to address the issues we encountered implementing Faster R-CNN. For the Linux OS dependency, we plan to leverage Amazon Web Services to build a virtual Linux machine on EC2. To develop a larger repository of training data in the PASCAL VOC format required, we plan to enlist the services of Amazon Turks and open-source image annotation software like LabelImg.[22] Though we fell short of our original objective, this project left us with the critical knowledge and skills necessary to perform computer vision. We're confident that once we overcome the remaining technical hurdles, we will be able to deliver a classification module to meet the business needs of Popspots.

[21] http://host.robots.ox.ac.uk/pascal/VOC/
[22] https://github.com/tzutalin/labelImg

Appendix

Project GitHub Repository

https://github.com/JDallasGriffin/APM-Project/

NOTE: The GitHub repository is private as it contains Popspots rack images. Please provide your GitHub UserIDs to Dallas Griffin and he will grant you access.