Image2Recipe

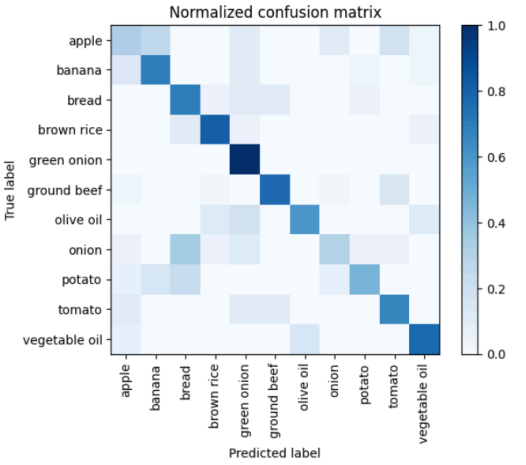
Dennis Phun, Christine Widden, Yohan Sofian, Gordon Luu

Project Description

The original objective of our project was to create an innovative application that bridges the gap between image classification and recommendation systems. We started with a convolutional neural network to identify individual ingredients present within individual food images and a recurrent neural network to generate recipes based on those identified. However, while experimenting with recipe generation, we figured that it might not be the best idea since the results were varied and often inaccurate. Instead, we decided to use a predefined set of recipes, sourced from an API, and expanded the scope of our project by focusing on the part of identifying ingredients. Rather than an image classification model, we attempted to create an object detection model without images annotated with bounding boxes. Our current application allows users to upload a single image, of say, their countertop or refrigerator that contains multiple ingredients, to then receive recipe recommendations tailored to the ingredients identified.

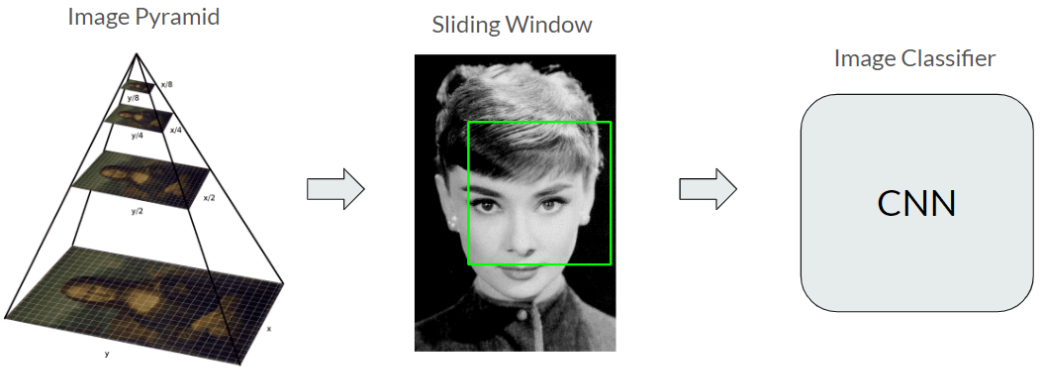
Project Design

The base model that our application uses is a CNN. To train it, we had to compile our own dataset of food ingredients, as there is no large, publicly available one. This consisted of web crawling images from search engines, namely Google and Bing, and manually filtering out any that stood out. With a

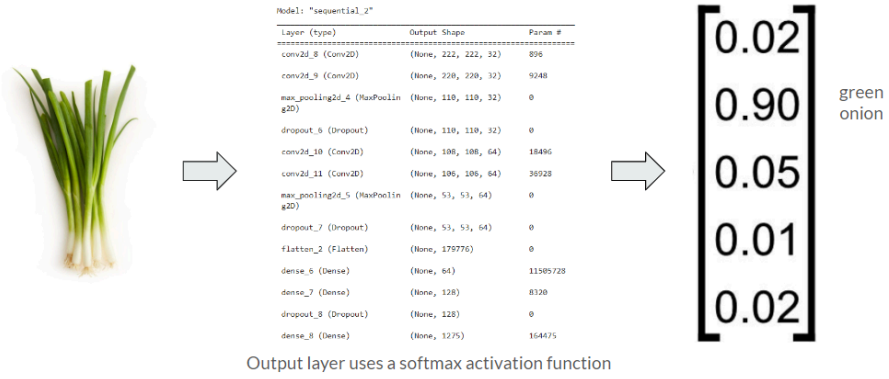


meager dataset of 1434 images, we were able to create a well performing model that is able to identify 11 different food ingredients with a testing accuracy of 76%.

To introduce object detection we could not use one of the state-of-the-art models such as YOLO or Faster R-CNN because we did not have a dataset annotated with bounding boxes. So, we decided to go with a more traditional approach of preprocessing the images with an image pyramid and a sliding window. We used an image pyramid to create copies of the image and resize them billinearly so that the aspect ratio was conserved. For example, if the input is 600x400, the scale is 1.5, and the limit is 200, it would make copies of size 400x267 and 267x178. Then, we ran a sliding window with a set size and stride across each layer of the image pyramid. Finally, each frame of the sliding window was passed into the CNN to be classified.



The output layer of the CNN uses a softmax activation function, meaning an image input results in a list of probabilities corresponding to the model's predicted classes. For each frame of the sliding window, we took the predicted class and used its probability as a confidence score. If the score passed our threshold, the frame is considered to contain that ingredient and the ingredient is added to a list.



Output layer uses a softmax activation function

With the list of ingredients gathered, users are then able receive recipe recommendations at the click of a button. The recipes are sourced from an endpoint of ApiDojo's Tasty API, where the input is a list of ingredients and output is a list of recipes; along with their a description, rating, cooking instructions, and other information.

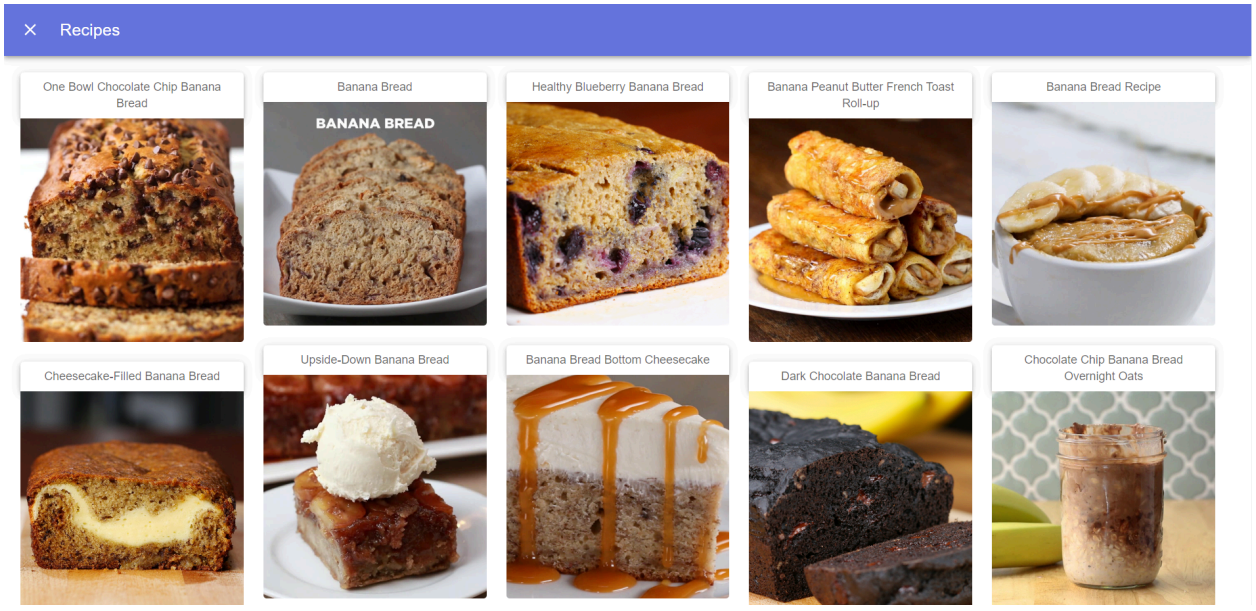
Implementation

For scraping, we used a library called icrawler in Python, which allows for the crawling (searching and downloading) of images from various search engines. We put a list of ingredients that we wanted to search and gather our data for to be downloaded locally which, after processing, was transferred into a folder on Google Drive to be used for training our model.

For the implementation of the model, we used Tensorflow with the Keras API. We worked with it primarily through Google Collab as it was easy access to a shared storage and work environment and also provided an easy to use GPU.

SELECT IMAGE

GET RECOMMENDATIONS

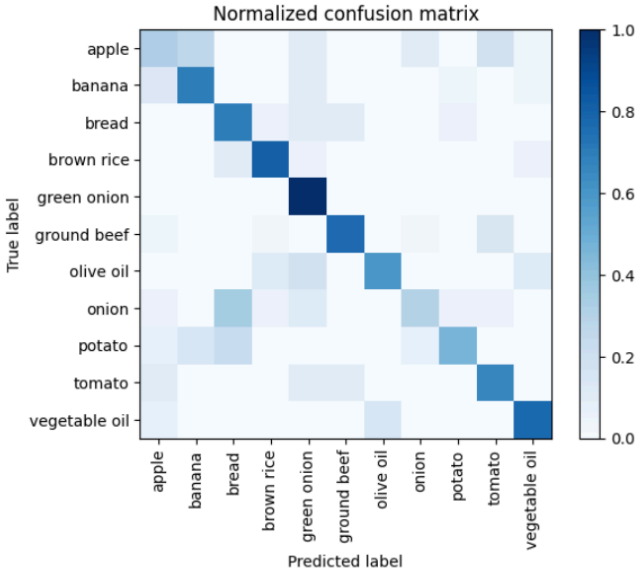


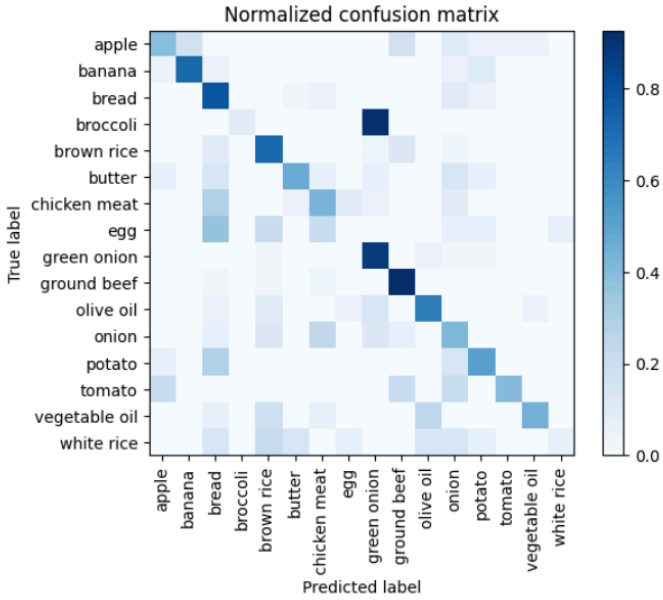
We decided the presentation medium was going to be a web application, albeit a rather simple one. It has basically no design and only two buttons: "Select Image" and "Get Recommendations." After users select an image and wait a bit of time for the backend to process all of the image frames, they can click the button to get recommendations. For its

implementation, we used a React frontend with an Express backend. In hindsight, it probably would have been a lot easier to use FastAPI or something compatible with Python, since we had to convert the model into one compatible with TensorFlowjs and also had to write all of the image preprocessing code in JavaScript.

Testing

When training our model, we randomly split our training data into a 80% training set and a 20% testing set, and further split the training subset into another 80-20 split for a validation set. We would use this validation set to determine when overfitting began to occur with training, and thus halt the training process. After finalizing the model, we would check how it performed with the testing set and more in detail on each class with a confusion matrix, normalized to account for discrepancies in the number of training images for each class.





Overall, we were able to get much better results with 11 classes versus with 16. This is likely due to there being less competition between the different classes, and the different classes being able to be more distinct from each other, with less potential overlap in similarity between images.

With 16 classes, there were some significant issues with identifying some of the classes. One of the easiest problems to spot in the confusion matrix was with broccoli being consistently identified as green onion. While green onion was consistently identified correctly, this seemed to be due to an overabundance of false positives: the model was very overzealous in its labeling of images as being green onion, with almost none of the broccoli images being identified correctly. This was a result of our unbalanced dataset. With green onions being one of the first classes we collected images for, we not only had a larger number of images (not much, maybe a few dozen), but also more conforming images, as we spent more time filtering them.

During development, this confusion matrix was used to identify which classes may have needed more filtering and more training images. After further filtering and adding to the training data, we saw some improvement in the model performance. For example, during one stage in development, yellow apples were consistently identified as bananas. By adding more images of yellow and green apples to the training data, the model was able to learn that not every image with yellow features a banana, and the images of yellow apples began to be correctly identified as apples.

The images that we chose for demonstration purposes were not a part of our dataset and selected for our specific use case. While our model had decent performance among all of them, there were definitely some flaws. The first test case is an image of a single ingredient. As you can see, it still performs rather well, correctly identifying each frame containing green onion as "green onion". Note that although there are many classifications, the output of the model would simply be an ingredient list containing "green onion."

The second test case is an image containing two ingredients: bananas and bread. It correctly identifies the banana and the top of the bread, however, it incorrectly identifies the middle of the loaf of bread as bananas. We aren't too sure why it does that, as most of our data for bread contains slices of bread. It may be due to the size of the window being too small to recognize it, and instead seeing the holes as dots, which are a similar feature to the sugar spots of bananas.

The third test case is one more specific to our use case. We were quite surprised at how well this one turned out, but it is also kind of expected. The ingredients are clearly separated from one another, they are not too large nor too small, they are not near the edges, and the background is white. It correctly labels the bread, ground beef and tomatoes. However, it incorrectly mislabels the lettuce as green onion because we have no lettuce, misses the onions because most of the onions we used are whole, and misses the seasoning because we have none.

The fourth test case is also more specific to our use case, however, it performs poorly for the opposite reasons. Ingredients are missed because they are too close together, they are not the right size, some are near the edges, and the background is brown. The last reason is why there are so many misclassifications of bread, a problem we found among

some other test cases as well. Also, with the other problems, they are not a problem with the image exactly, but rather one due to our window size and stride. With our current model and how we have preprocessing set up, those values are set and may not be compatible with all images. We chose one that works relatively well, but if we were able to run it on a more powerful machine that allowed for parallelization, multiple windows can be run without the expense of time, and the problem may potentially be fixed.

Analysis

With a small dataset, we were able to create a mildly successful ingredient identifier. However, our current system is slow, limited, and often inaccurate. Perhaps our most major limitation was the issue of sourcing images. Our images were sourced by scraping Google Images, as we could not find a pre-existing dataset that matched our needs. Searches with broad terms, such as “chicken” or “broccoli” would result in many images which poorly fit our needs (e.g. cooked dishes consisting of them), but more specific searches were severely limited by the number of images we could acquire. As a result, we saw little variety in the images we sourced for some classes, such as bananas, and the sourced images were often much more akin to stock images, rather than the sort of unprofessional photos a user might take of ingredients inside of their fridge or set out on their countertop. The dataset was also unbalanced due to this method of acquiring the images, resulting in disparate performance in the different categories. With image augmentation having shown to not give us any improved performance, it makes sense for the next step to be focused on enhancing the available data. Upon completion of that expansion, we can either continue to refine the model by adding borders during the convolution step and choosing the best bounding boxes in the image pyramid or we can spend our time annotating the images so it can be used in a SOTA object detection model. Some other problems faced involved

the Google environment where some of us had run out of computational hours or were filling up our available storage so that we could not add any more images. For example, someone had to stop in the middle of training because they could not work on it locally after having run out of computational time. Perhaps we could have created more accounts or paid for their services to have made our experience less painful, but we believed we reached adequate results.