Brandon Martin

Ethan Melero

Nicholas Piazza

Computer Vision with Symbolic Relations

Link to Project Github Repo: https://github.com/Ethan-M-123/cs3520_TheLazyChefs

Link to Deployed Application: https://airpiazza.github.io/the-lazy-chef-deploy-this/

Link to Youtube Video Presentation: https://www.youtube.com/watch?v=kEziNqVPKDc

I. Abstract:

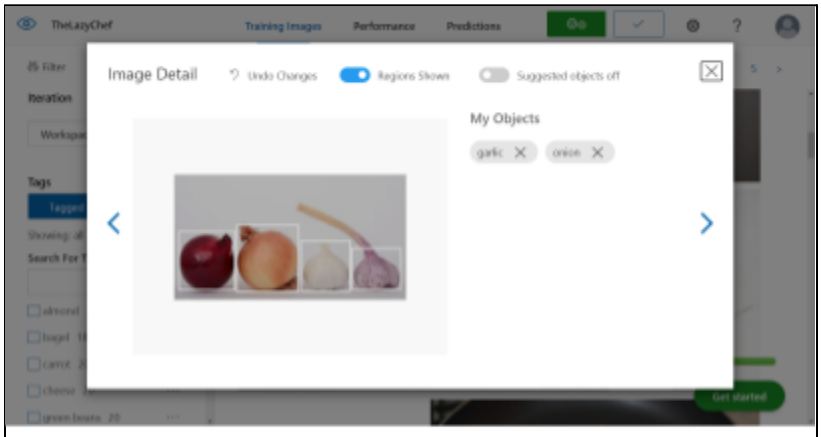
The applications of computer vision by themselves are very niche in use cases, but with the addition of a referential knowledge base, the applications of computer vision are heavily broadened. To show this we created a web application called The Lazy Chef which uses a computer vision model specially trained on grocery store items to detect items in the frame and tell the user what they could make with the ingredients detected. We utilized CustomVision for the computer vision model, Prolog for the knowledge base, and React for the web application. We began with an image classification model as a proof of concept, then evolved to a pre-trained object detection model, and finally, an objection detection model utilizing a snapshot as the final product. In the future, this can be scaled upwards with the improvement of the models precision and recall as well as moving away from CustomVision due to its limitations, improve the knowledge base by having the recipe instructions not reference 3rd party sites but rather pre-written instructions as well as increasing recipe number, and finally moving to a live video

feed to process ingredients in the frame. In conclusion, we found that the addition of a knowledge base provides practical implementations of computer vision.

II. Introduction:

Object detection is able to identify items. For example, models can be trained to recognize ingredients. However, object detection is unable to find relationships between ingredients, such as where they are included together in recipes. For a person who would like to know what they could make with the ingredients they have, object detection only takes them halfway to his or her goal, merely recognizing the ingredients. Prolog, however, can help find the relations between the ingredients, which is essential for finding the recipe. Classical AI's ability to find relations makes up for computer vision's lack of relationship finding. By linking object detection with logic programming, users can both experience object detection and relationship finding in one seamless process. Our goal is to use object detection to detect ingredients and then use Prolog relations to find recipes for users based on the detected ingredients.

The object detection model we used was tensorflow.js and it was provided by customvision.ai, a Microsoft service that allows the uploading and labeling of images to build a dataset that you can train either an image classification or object detection model on.



Example of the image labeling in CustomVIsion

It came very much in handy as we had no experience using tensorflow.js or its installable utilities, and CustomVision's simple user interface, as well as iteration performance

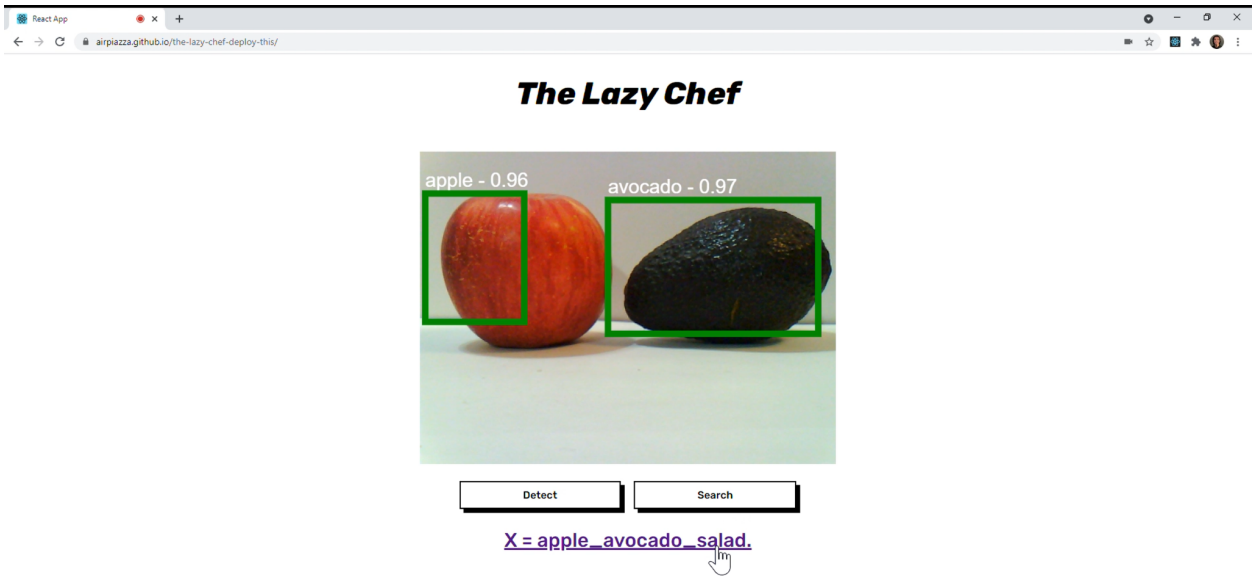
information, allowed for easy optimization of the model. Obviously, we also used what tensorflow.js had to offer in terms of API and methods for actually using the model in the application.

Once we figured out how to find what the webcam was detecting, we needed a way to find the relation between the ingredients detected. For our use case, we needed to use a knowledge base to find these relations. We decided to use Prolog to help identify recipes that can be made from the ingredients object detection would detect. Our choice library is Tau Prolog, which is a Prolog interpreter in JavaScript. This library is a good fit since the object detection and the user interface are in JavaScript. Given an array of ingredients that the object detection model recognizes, Tau Prolog is able to help us query a knowledge base that we provide and, through unification, gives the user the recipe that can be made with the ingredients. The recipe is also a key in a dictionary that has recipe links as values that we give to the user along with the recipe. Below is the Tau Prolog code where we pass in an array of detected ingredients and set the recipe to the result of the unification:

```
// consulting the knowledge base
    session.consult(knowledgeBase, {
      success:function() {
        // query knowledgeBase with the sorted list of detected ingredients
        session.query("recipe("+[ingredientArray].sort()+", X).", {
          success: function(goal) {
            session.answer({
              success: function(answer) {
                // output result of the query
                console.log(session.format_answer(answer));

                // set recipe state to the result of query
                setRecipe(session.format_answer(answer));
              }
            });
```

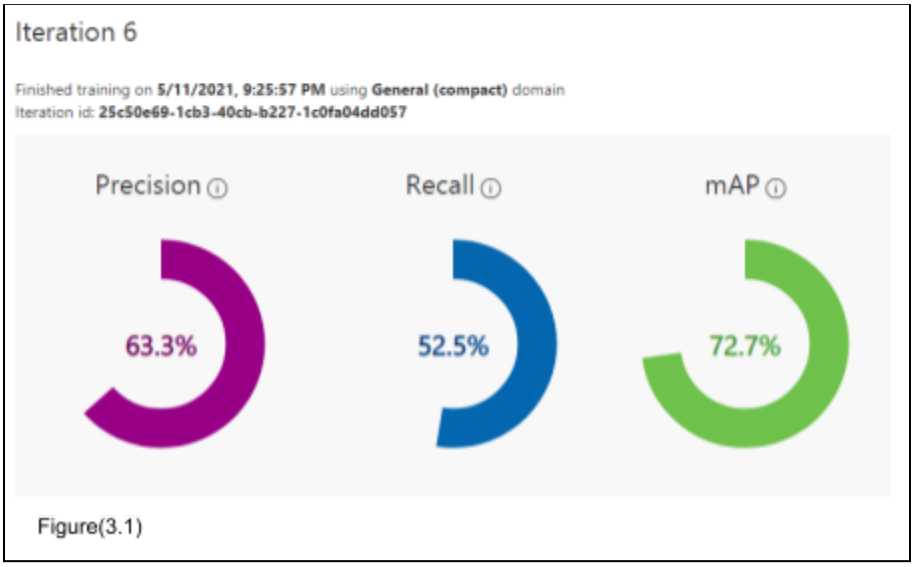
```
        }
      })
    },
    error: function(err) {
      // throw error message if there is an error
      console.log("error: "+err);

      // set recipe state to empty string
      setRecipe("");
    }
  });
```

The user then is either shown a message saying that no recipes have been found for those ingredients or the result of the unification, such as, "`X = apple_avocado_salad.`" as shown below:



III. Methods/Algorithms/Concepts:

We originally began with an image classification model as a proof of concept, but we needed to scale it as it only recognized one ingredient at a time which was impractical for a user; having the user scan each individual ingredient in order to finally get a recipe was not ideal. Therefore, we moved to object detection which would allow the model to recognize multiple ingredients at once, save the user some time, and, overall, speed up the entire process of using the application. Our original idea was to use a video camera from the computer and have the model interpret each image in the feed to say what ingredients are on screen.



Figure(3.1)

Though we didn't just scale to using an object detection model, we also scaled dramatically in the number of ingredients so we could increase the number of recipes the user can be told. In fact, we went from 3-4 ingredients to 50, though this required a large jump in the number of images in our dataset as well as putting in the manual labor of labeling each image for the dataset. Though CustomVision made our lives easy by giving us a simple user interface and model statistics (Figure 3.1), what it did not give us is the ability to use an auto labeler of any kind and as such images entered into the dataset had to be manually boxed and labeled for the model to train off of.

Though once the model was trained it was now about integrating the model into the code, and after lots of trials, we found that CustomVision had a package we could use specifically for the models that they output (as they ended up being slightly different than regular tensorflow.js model files); this made integrating the model into the code just that much simpler (as it was a single function that could be used to interpret the tensor output). In comparison, what we were doing before involved using an incorrectly implemented linear regression algorithm to interpret the output tensor our model would give us; this would cause our drawn boxes, interpreted classes, and scores to be very inconsistent.

In regards to how ingredients are stored and accessed, as seen below, each ingredient's name is associated with an individual index number. After the screenshot is handed off to the CustomVision package, an ingredients index value is returned. It is then stored so that its index value and name can be accessed throughout the program.

```
export const labelMap = {
    0:{name: 'almond'},
    1:{name: 'apple'},
    2:{name: 'asparagus'},
    3:{name: 'avocado'},
    4:{name: 'baby_corn'},
    5:{name: 'bacon'},
    6:{name: 'bagel'},
    7:{name: 'balsamic_vinegar'},
    8:{name: 'banana'},
    9:{name: 'bean'},
    10:{name: 'bell_pepper'},
    11:{name: 'blackberry'},
    12:{name: 'blueberry'},
    13:{name: 'bread'},
    14:{name: 'brocolli'},
    15:{name: 'butter'},
    16:{name: 'carrot'},
    17:{name: 'cheese'},
```

```
    18:{name: 'chicken'},
    19:{name: 'chocolate'},
    20:{name: 'cookie'},
    21:{name: 'cream_cheese'},
    22:{name: 'egg'},
    23:{name: 'garlic'},
    24:{name: 'green_beans'},
    25:{name: 'ground beef'},
    26:{name: 'ham'},
    27:{name: 'honey'},
    28:{name: 'ice_cream'},
    29:{name: 'ketchup'},
    30:{name: 'lasagna_noodle'},
    31:{name: 'lettuce'},
    32:{name: 'lunch_meat'},
    33:{name: 'marshmallow'},
    34:{name: 'mayonnaise'},
    35:{name: 'milk'},
    36:{name: 'mushroom'},
    37:{name: 'mustard'},
    38:{name: 'noodle'},
    39:{name: 'oatmeal'},
    40:{name: 'onion'},
    41:{name: 'potato'},
    42:{name: 'sausage'},
    43:{name: 'shredded_cheese'},
    44:{name: 'strawberry'},
    45:{name: 'sugar'},
    46:{name: 'sweet_potatoes'},
    47:{name: 'tomato'},
    48:{name: 'tortillas'},
    49:{name: 'yogurt'}
}
```

After the image has been processed and a list of ingredients available is created, a database with relations is accessed that is written in Prolog. It is formatted as such:

recipe(ingredient_1, ingredient_2, name of recipe). In this format, ingredients 1,2,3…. are listed in alphabetical order. This makes the database extremely proficient when parsing through to obtain any relevant information needed for output.

```
    // the prolog knowledge base with headless horn clauses that signify the
relation between ingredients and recipe
    const knowledgeBase = `
      recipe(bread, butter, buttered_toast).
      recipe(avocado, bread, cheese, lunch_meat, tomato, fancy_sandwich).
      recipe(almonds, balsamic_vinegar, blueberry, cheese, lettuce, mustard,
summer_blueberry_almond_salad).
      recipe(milk, potato, shredded_cheese, baked_potato).
      recipe(shredded_cheese, tortilla, cheese_quesadilla).
      recipe(bean, tortilla, bean_and_cheese_burrito).
      recipe(avocado, lunch_meat, tomato, tortilla, wrap).
      recipe(lettuce, lunch_meat, shredded_cheese, chef_salad).
      recipe(ground_beef, lasagna_noodle, shredded_cheese, tomato, lasagna).
      recipe(noodle, tomatoes, spaghetti).
      recipe(bread, shredded_cheese, tomatoes, pizza).
      recipe(chicken, tomato, tortilla, onion, chicken_tacos).
      recipe(bacon, egg, noodle, bacon_and_egg_pasta).
      recipe(brocolli, milk, noodle, shredded_cheese, brocolli_mac_and_cheese).
      recipe(chicken, shredded_cheese, tomato, chicken_bake).
      recipe(asparagus, chicken, garlic, lemon, lemon_garlic_chicken).
      recipe(avocado, carrot, lettuce, lunch_meat, tortilla, hefty_wrap).
      recipe(brocolli, ground_beef, ketchup, onion, sweet_potato, tomato,
meatloaf).
      recipe(baby_corn, bell_pepper, brocolli, cucumber, green_bean, mushroom,
soy_sauce, vegetable_stir_fry).
      recipe(cookie, ice_cream, ice_cream_sandwich).
      recipe(milk, strawberry, strawberry_milkshake).
      recipe(chocolate, milk, chocolate_milkshake).
      recipe(chocolate, marshmallow, smores).
      recipe(apple, chocolate, chocolate_covered_apple).
      recipe(cheese, egg, sausage, sausage_omelette).
```

```
      recipe(bacon, cheese, egg, bacon_omelette).
      recipe(bacon, cheese, egg, sausage, lumberjack_omelette).
      recipe(cheese, egg, cheese_omelette).
      recipe(bacon, bread, butter, cheese, egg, milk, potato, sausage,
cheesy_bacon_sausage_egg_hash_brown_skillet).
      recipe(bread, egg, milk, mustard, shredded_cheese, sausage,
sausage_breakfast_cassarole).
      recipe(bacon, egg, onion, potato, shredded_cheese, sheepherders_breakfast).
      recipe(egg, potato, sausage, tortilla, sausage_breakfast_burrito).
      recipe(bacon, egg, potato, tortilla, bacon_breakfast_burrito).
      recipe(bacon, egg, sausage, tortilla, potato, combo_breakfast_burrito).
      recipe(milk, potato, shredded_cheese, cheesy_hash_brown_bake).
      recipe(banana, blueberry, strawberry, berry_smoothie).
      recipe(butter, egg, garlic, sweet_potato, sweet_potato_and_egg_skillet).
      recipe(bacon, egg, milk, potato, shredded_cheese, hash_brown_egg_bake).
      recipe(egg, hard_boiled_egg).
      recipe(bread, egg, egg_sandwich).
      recipe(bread, toast).
      recipe(bread, cheese, grilled_cheese).
      recipe(avocado, egg, baked_egg_in_avocado).
      recipe(apple, avocado, apple_avocado_salad).
      recipe(bell_pepper, egg, bell_pepper_omelette).
      recipe(butter, egg, egg_butter).
      recipe(egg, sausage, scrambled_egg_with_sausage).
      recipe(egg, onion, egg_and_onion).
      recipe(egg, garlic, garlic_fried_eggs).
    `
```

With the object detection model and the Prolog knowledge base, what is left is to be able to retrieve detected ingredients from a video feed and query the knowledge base on those detected ingredients. For retrieving the detected ingredients, the customvision-tfjs package was very helpful. As shown below, with a few lines of code, we are able to load our custom model, use the user's paused video feed to do object detection on and get back an array of classes,

probability scores, and bounding box dimensions in a “result” array. This process takes O(n*m) time since customvision-tfjs traverses nested tensor arrays to yield the results.

```
// detect objects using the customvision dependency
    let model = new cvstfjs.ObjectDetectionModel();
    await model.loadModelAsync(process.env.REACT_APP_MODEL_URL);
    const image = document.getElementById('video');
    const result = await model.executeAsync(image);
```

From the result array, we are able to draw bounding boxes with drawRect and save the detected ingredients into the “ingredients” state using setIngredients. Both drawRect and setIngredients take O(n) time since they each have a for-loop to filter through the detected items that have a probability score higher than 25%.

```
    // draw the bounding boxes
    drawRect(result[0], result[2], result[1], 0.25, width,height,
camera.getCtx());

    // set the ingredient state to the array of ingredients detected
    setIngredients(output(result[0], result[2], result[1], 0.25));
```

As shown below, once we have our ingredient array, we can just query the knowledge base using that ingredient array, yield results of the query, which is a recipe, and save the recipe into the “recipe” state.

```
// consulting the knowledge base
    session.consult(knowledgeBase, {
      success:function() {
        // query knowledgeBase with the sorted list of detected ingredients
        session.query("recipe("+[ingredientArray].sort()+", X).", {
          success: function(goal) {
            session.answer({
              success: function(answer) {
```

```
            // output result of the query
            console.log(session.format_answer(answer));

            // set recipe state to the result of query
            setRecipe(session.format_answer(answer));
          }
        });
      }
    })
  },
  error: function(err) {
    // throw error message if there is an error
    console.log("error: "+err);

    // set recipe state to empty string
    setRecipe("");
  }
});
```

Once we have the recipe saved into the “recipe” state, we then display the recipe to the user along with an associated link that will take them to a website that has that recipe.

IV. Experiments:

The first experiment we set up was an image classification model following a video reference, to show some proof of concept (Train, 2018). However, we used a custom trained image classification model using Teachable Machine, trained specifically on a small number of household ingredients such as eggs and slices of bread. Though this had many limitations (only individual ingredient detection) and hence was left behind and scaled up to an object detection model.

From the image classification version, we moved onto an object detection version of the project, though this was a very experimental stage. At first, we just had a pre-trained model from COCO-SSD to even draw bounding boxes in the frame, which was the main purpose of this step, to transition from just telling you just what's in the frame like in image classification to telling you the multiple objects in the frame. The version was also using the ml5 library which is what allowed us to use the pre-trained COCO-SSD model as well as other very important functions that made tensor interpretation fast and reliable.

However, we needed to transition from the COCO-SSD model to a model of our own, and this proved difficult. You see, the ml5 library actually only allowed the use of either the COCO-SSD or YOLO pre-trained models which means we couldn't use it at all for a custom model of our own. This forced us to transition away from ml5 and onto tensorflow.js's library as well as CustomVision's. If only it was so simple, after attempting to transition we followed another video reference that showed us a special linear regression function that, supposedly, had to be used to interpret the tensor from the custom model (Dittmann, 2020). This made the model interpret frames very, very slowly to the point where live video feed interpretation just wasn't an option. Even after finding out that CustomVision itself had a package that we could use to interpret the tensors, processing a live video feed was still too slow.

From here we switched over to a snap-shot model, instead of feeding the computer vision model a live video feed to interpret each individual frame at a time, it is given just a single snap-shot of what is currently in the frame of the camera (kind of like taking a picture). This helped us fix our live video feed problem of a choppy frame rate and even slower interpretation to the point of not processing anything and hence got us to our final product, the version that is currently deployed.

We also performed some experiments with different ingredients at our disposal around the house. Experiments consisted of putting food items in front of the webcam, recording how long it took for The Lazy Chef to do object detection. The probability scores were recorded for each ingredient detected and the average of those scores was recorded as well. Finally, we tested querying the Prolog knowledge base with the ingredients detected, recording what recipe was found with the ingredients detected.

Ingredients Shown	Ingredients Detected	Detection Time (s)	Probability Scores (%)	Average Probability Score (%)	Recipe Found
apple, avocado	apple, avocado	27.15	apple: 98.18, avocado: 99.35	98.77	avocado and apple salad
White bread slice	nothing	36.26	N/A	N/A	N/A
white egg	egg	19.21	egg: 70.45	70.45	hard boiled egg
Avocado, White egg	Avocado, egg	45.07	Avocado: 97.92, Egg: 73.65	85.79	Baked egg in avocado
White bread slice, pecorino Romano cheese	Ice cream	44.82	Ice cream: 66.24	66.24	nothing
Onion, white egg	Onion, egg	45.08	Onion: 66.65, Egg: 85.53	76.09	Egg and onion
Garlic, white egg	Garlic, egg	18.43	Garlic: 85.49, Egg: 84.36	84.93	Middle Eastern Garlic Fried Eggs

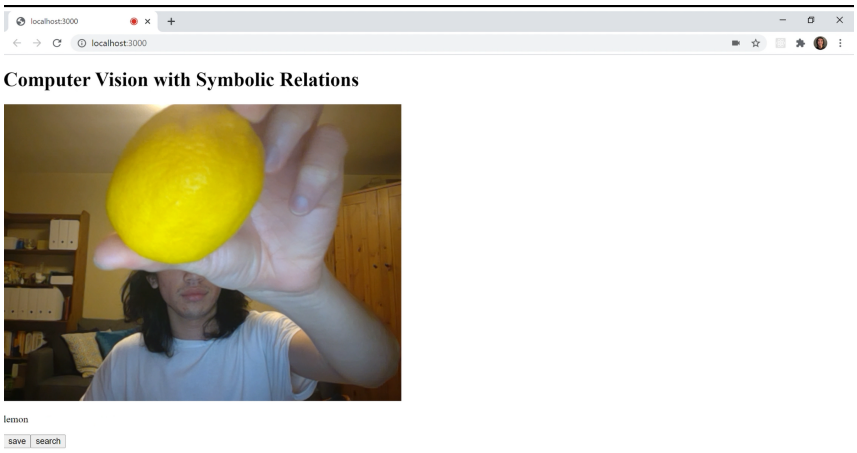
Chocolate, milk	mustard	45.83	Mustard: 61.62	61.62	N/A
Apple, chocolate	apple	45.91	Apple: 91.32	91.32	N/A
Tortilla, shredded cheese	Ice cream	45.4	Ice cream: 79.18	79.18	N/A
White egg, butter	Butter, egg	19.07	Egg: 89.18, Butter: 35.20	62.19	Munavoi - Finnish Egg Butter
White egg, white bread slice	marshmallow	45.49	Marshmallow: 33.76	33.76	N/A
Green bell pepper, white egg	Egg, bell pepper	42.16	Egg: 62.75, Bell pepper: 56.99	59.87	Bell Pepper Vegetarian Omelette
White egg, sausage	Sausage, egg	35.60	Egg: 65.93, Sausage: 82.19	74.06	Scrambled eggs with sausage
Banana, strawberry, blueberry	Banana, strawberry	45.66	Banana: 46.16, Strawberry: 47.01	46.59	nothing

Table 1 - Ingredient detection time and probability

V. Results

Throughout the lifespan of the project, multiple versions were iterated through. The result of experimenting with image classification for our use case was a p5.js (p5.js is a JavaScript library for drawing) application that performed image classification using a custom model made using Teachable Machine and the ml5(machine learning) library. The ml5.js image classification

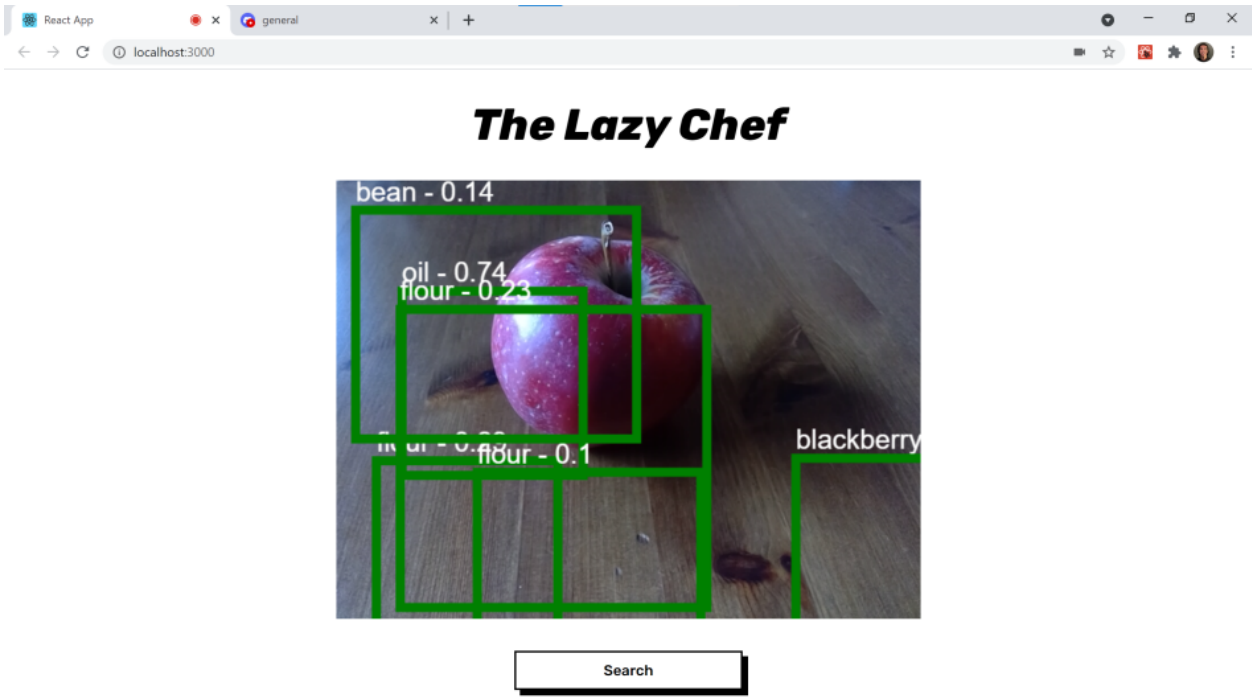
tutorial from The Coding Train helped us implement the image classification for this version of the project (Train, 2018). The user would show one ingredient at a time to the camera and query for recipes on the ingredients shown. Below is what our first iteration of the project looked like.



Our next iteration was done using object detection on a pre-trained model. We used p5.js and ml5 for this version of the project as well. Our pre-trained model of choice was the COCO-SSD (trained using the Common Objects in Context dataset and recognizes about 80 different items) model. What helped us implement object detection was a tutorial from the Coding Train on object detection with ml5.js (Train, 2018). The result of experimenting with this way of solving the problem was a live webcam feed object detection app that recognized very few ingredient items. The pre-trained model recognizes 80 classes, only several of them being viable ingredient items. We were hoping that ml5 would allow us to insert our own model, but it only does object detection on pre-trained models. We then decided to experiment with React and tensorflow.js for our use case.

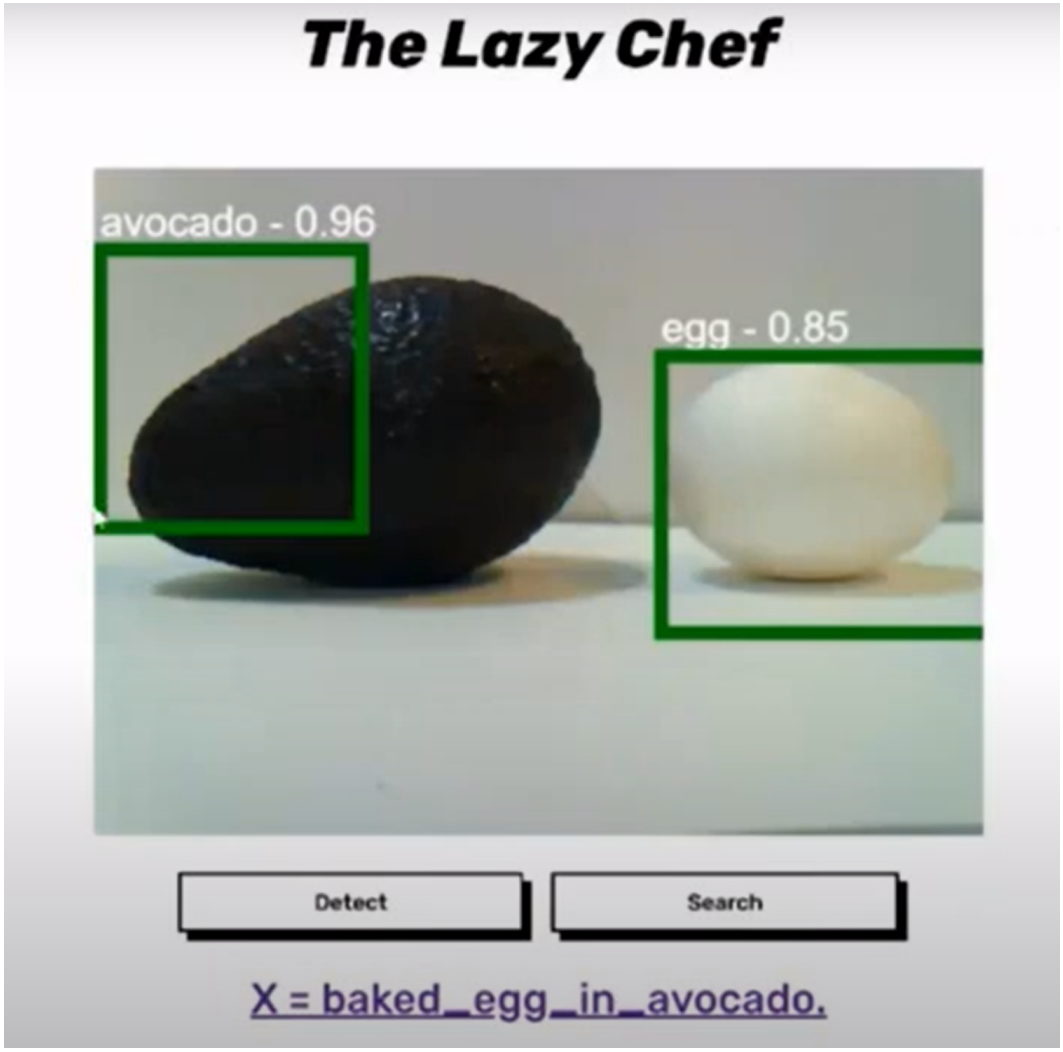
We were able to get up and running with a React app integrated with tensorflow.js by following a tutorial from Nicholas Renotte on tensorflow.js with React (Renotte, 2020). We were

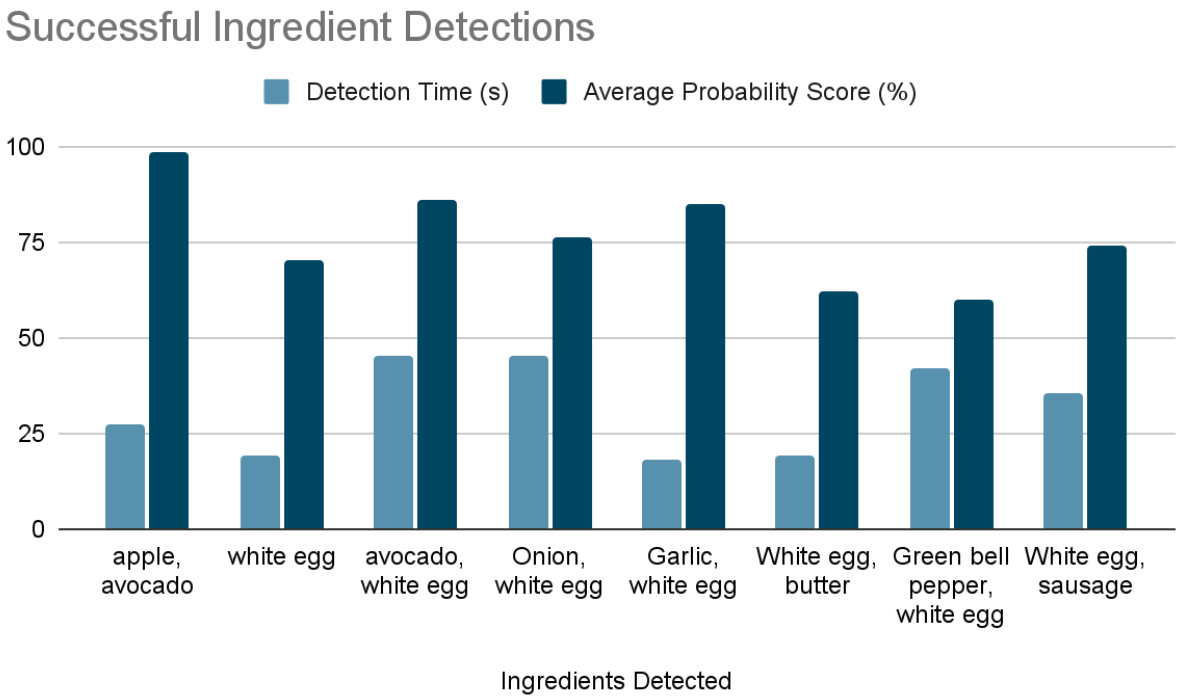
also able to find a way to mathematically derive scores, classes, and boxes by following a tutorial on object detection by Sascha Dittman (Dittman, 2020). This iteration consisted of a React app that uses the tf.js library to perform object detection. The result of experimenting with this way of solving our problem was a memory-hungry application that provided inaccurate scores, bounding boxes, and classes. This iteration was also very inefficient, unable to draw bounding boxes efficiently on the live video feed. The memory costs were caused by tensor memory management issues, and the efficiency costs were caused by the linear regression calculation which took O(n*m*k) time. Below shows how inaccurate The Lazy Chef was at this stage of development.



After experimenting with different ways of tackling this problem, we ended up using a custom object detection model made using CustomVision integrated into a React application that uses the customvision-tfjs package for object detection and Tau Prolog for querying our

knowledge base. Below is what our final product looks like, the ultimate result of all our experimentation with solving our problem,





The graph shown above shows the results of the successful ingredient detections. As for the results of the ingredient detection experimentation, Table 1 shows that there were certain ingredients that The Lazy Chef is proficient at detecting, which then allowed for successful recipe finding. Another observation we made is that querying the knowledge base, after detecting the ingredients provided, almost instantaneously gives back a recipe for the given ingredients. Among the successful trials, the graph shows that the detection time takes roughly 19 to 45 seconds to complete. We believe this detection time is caused by the time it takes the customvision-tfjs package to perform operations. The average probability scores among the successful detections shown in the graph above were roughly between 59-99 percent.

Our unsuccessful trials show areas for improvement for The Lazy Chef. It had difficulty detecting ingredients such as bread, tortillas, milk, blueberries, chocolate, shredded cheese, and other ingredients. Either the model would detect the ingredient as something else or

not detect the ingredient at a probability higher than 25%. We believe that this is caused by our model being trained on a limited amount of photos which made generalization difficult and overfitting apparent. For a product to be pushed out to market, a more efficient application with higher accuracy would be necessary. With more resources, we would be able to make a better model than what CustomVision limited us to.

VI. Future Work:

In the future, the model would be expanded beyond the 50 tag limit that custom vision restricts us to, via moving over fully to tensorflow.js software allowing us to add as many tags as we want (since we previously scrapped roughly 20 ingredients due to the tag limitation). Though the number of tags is not the only limitation of the current model, its recall and precision, though not terrible, is definitely not perfect and hence could be improved via an increase in the number of images for the available tags as the more images the model gets to train on, the better it will perform in practice. Though since the eventual scale direction is away from CustomVision, big changes will need to be done on the application end as we would be unable to use CustomVision's package to interpret the tensor and instead have to move to fully use tensorflow.js API.

One major change that needs to be made is that, as of now, all recipes are currently accessed through 3rd party websites. The reliability of these sites' continual operation is not within our control, and thus should any of the sites currently being utilized go down (not very likely, but not impossible), then potentially significant portions of our application would cease to function properly. A way to subvert this is to create a local database/website in which all recipes are stored and available with a known level of reliability. In doing this, a script/ program can be

written to scrub information from recipe websites to automate the gathering of recipes (assuming this is even legal). This not only would make the program run quicker but would allow us to have a surprisingly high amount of recipes in the form of easily modifiable data. This would allow us to also include features such as a search function and filterability by calories, specific ingredients, diet types, and many other categories. This would also allow features such as either "full" or "partial" recipes to be used with "full" being that the user has everything required to make the recipe, and with "partial" being that the user has most of the necessary ingredients available and would only need to purchase a few additional items.

One optional line of research is the integration of the concept into the Internet of Things such as refrigerators with cameras inside that can interpret what you have in your fridge and perhaps suggest what you can make for dinner. In fact, with some cursory research, Samsung and LG are right there with us, recently releasing some AI-powered refrigerators that recognize the grocery items they currently hold, though they seem to be going down a different vein of utility which is telling you that you are low on milk, so we still have some legs in the recipe suggestion market. Though the integration into refrigerators sounds incredibly practical, this is only in theory as it actually might narrow the market on who would have access to the application, while moving to a mobile application might actually prove to be more fruitful since anyone could have it! Though more research may be necessary.

In continuing with the potential for utilizing camera-equipped IoT devices, an additional possible expansion for future work would include some form of integration with grocery delivery services. In this manner, if your refrigerator were able to tell you that you had 5 of the 7 ingredients required for a recipe you wanted to make, the remaining ingredients could be added

to a “cart” and then delivered directly to your home either immediately, or at a designated time based on the user’s discretion.

As of now, the application has full functionality to provide demonstrable proof-of-concept work within a desktop/laptop environment. While functionality is present for use on mobile platforms currently, it is not refined enough in terms of practicality. Additional work can be done to optimize the mobile users’ experience so that either front or rear-facing camera can be switched between, and a native application can be built instead of relying fully on a webpage release.

The final addition we would like to add to this project would be the way that images are processed in order to identify ingredients. As of the time of this paper, a “snapshot” method is used where the CustomVision package only needs to process a still image while identifying ingredients. Ideally, we would be able to either utilize or develop a system in which fully live feed can be used. One idea to get this working would be to possibly use a “white out” method, where behind the scenes, as objects are identified, they are covertly “removed” from any future identifications until the application is closed or the user clears the list for a rescan.

VII. Conclusion:

In conclusion, the project not only provides validity in terms of a proof-of-concept in regards to the combination of newer style object detection AI supplemented with a classic relational database, but it also provides a practical implementation as well. The speed at which the information is drawn upon from a Prolog database can easily complement the bit of extra time taken by object detection AI to present usable data in a timely manner. In addition, while showing the symbiotic availability, the practicality can also be extended into saying that the use

of an application as this can also provide users with ways to utilize the foods they have to create something instead of possibly letting the food sit till its expiration date.

References

Dittmann, S. (2020, August 31). TensorFlow.js Object Detection Made Easy. YouTube. https://youtu.be/7gOYpT732ow

Lee, D. (2020, January 2). Samsung and LG go head to head with AI-powered fridges that recognize food. The Verge. https://www.theverge.com/2020/1/2/21046822/samsung-lg-smart-fridge-family-hub-instaview-thinq-ai-ces-2020

Renotte, N. (2020, November 22). Building a Real Time Sign Language Detection App with React.JS and Tensorflow.JS | Deep Learning. YouTube. https://youtu.be/ZTSRZt04JkY

Train, T. C. (2018, August 2). ml5.js: Webcam Image Classification. YouTube. https://www.youtube.com/watch?v=D9BoBSkLvFo&feature=youtu.be

Train, T. C. (2020, October 20). ml5.js: Object Detection with COCO-SSD. YouTube. https://youtu.be/QEzRxnuaZCk

Code References

https://github.com/nicknochnack/ReactComputerVisionTemplate

https://stackoverflow.com/questions/58806971/how-can-i-take-a-picture-in-react-web-application-not-native

https://youtu.be/ZTSRZt04JkY?t=2335

https://github.com/microsoft/customvision-tfjs

https://github.com/ml5js/ml5-library/blob/main/examples/p5js/ObjectDetector/ObjectDetector_COCOSSD_Video/sketch.js

https://editor.p5js.org/ml5/sketches/ImageClassification_Video

http://tau-prolog.org/manual/compatibility-with-nodejs

http://tau-prolog.org/manual/a-simple-tutorial