Recipe Recommender

By Alina Shabanova, Bowei Ren, Harrison Mitgang, and Jane Wu

Created for the CPEN291 final project.

Brief Description

The Recipe Recommender is a web app that produces an accurate list of recipe recommendations for a user to recreate, based on what ingredients they have available and how they rate random dishes. The user is asked to upload several images of the food items they have, as well as rate 5 random dishes generated by the website. The final output is a personalized list of recipes that accustom to the users' taste and contain the ingredients they have previously uploaded as images.

A classifier is used to identify the food items present in the images uploaded by the user. The machine learning classification model uses transfer learning with a pretrained ResNet 18 Convolutional Neural Network and is trained using a carefully constructed image dataset containing images of 26 types of ingredients.

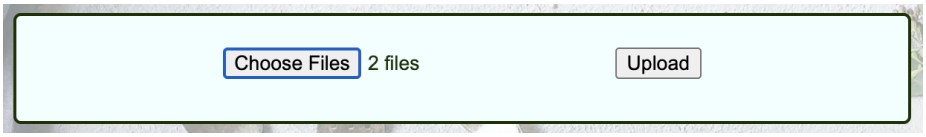
A recommender is used to create a list of recipes suitable for the users' taste, based on how they rated a randomly generated list of 5 dishes. The user can decide between using 2 different recommender models: the Deep Recommender and the Quick Recommender. The Deep Recommender is based on collaborative filtering and is built using PyTorch. The Quick Recommender is built using RF-Recommender.

The list of ingredients produced by the classifier is used to filter the list of recipes generated by the recommender, creating a customized list of recipes that all contain the food items from the list of the ingredients.

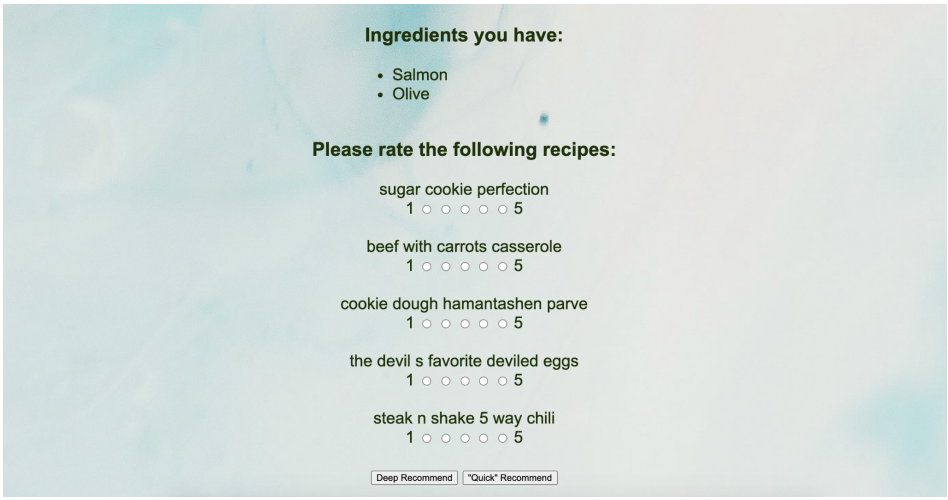
Project Walkthrough

When users navigate to the recipe recommender website, they are greeted by the following page:

Here, users can upload images of their supported ingredients. Of course, they are able to select multiple images to allow for multiple ingredients. Once uploaded, users will either see a single filename, or how many images have been uploaded.

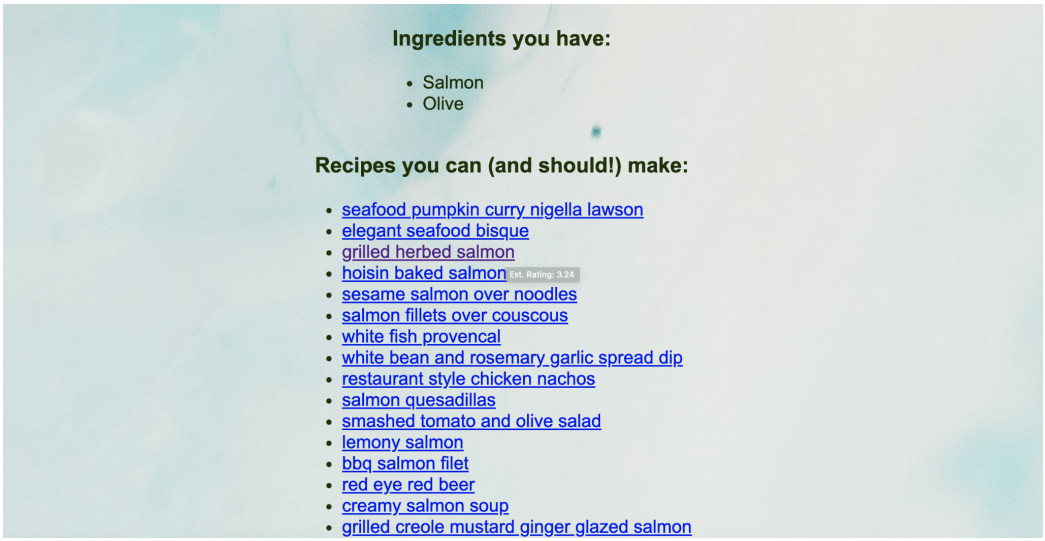


Once users click “Upload”, they will see the following screen.



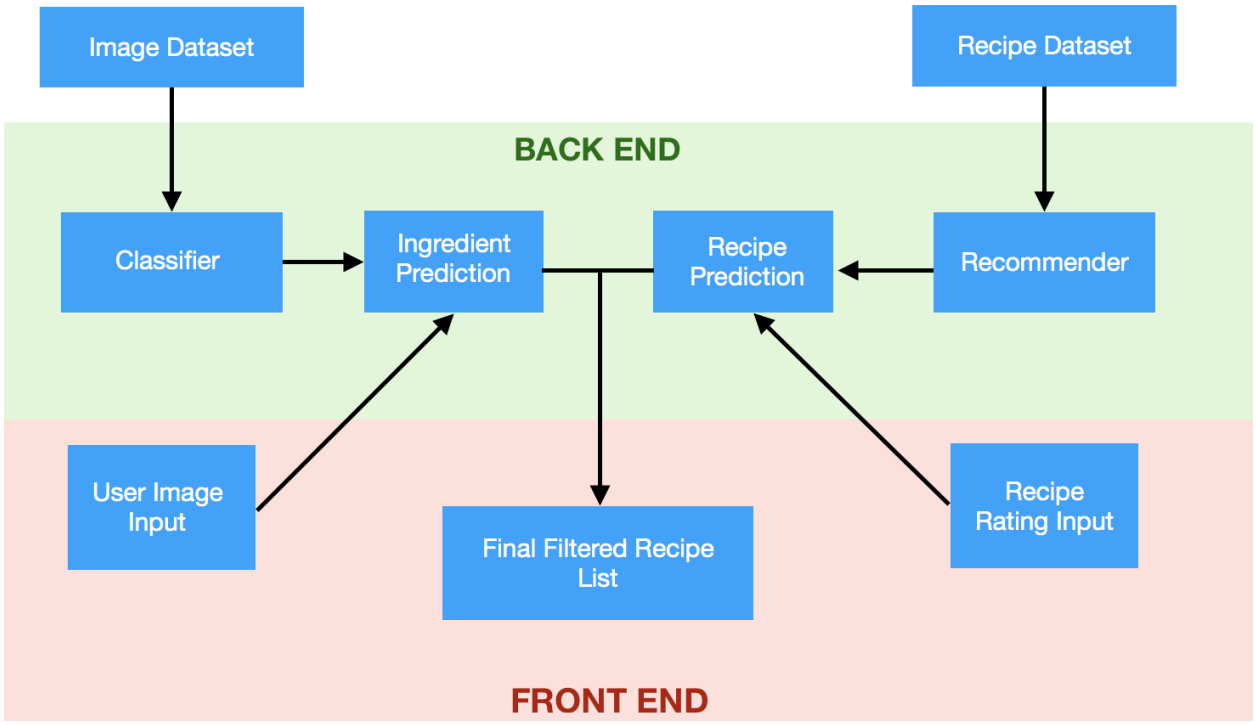
At the top, users can ensure that the images have produced the correct ingredients. Next users must rate 5 random recipes from 1 to 5. These 5 recipes will change each time, so users will have a unique experience each time they visit. Users are presented with two buttons (with tooltips reiterating their descriptions from the last webpage). The only difference between the buttons is how the backend handles the recommendation: “Deep Recommend” uses matrix factorization, while “Quick Recommend” uses a method called RF-Rec. Either way, once either button is clicked, users will see:

Users may watch the training/prediction process progress in the form of a percentage that updates live. Since the training process can take a long time (up to around 10 minutes), this feature lets users know that “stuff is happening” and to hold tight. Once the training is complete, the user will be presented with a list of recipes sorted by predicted rating:



Finally, a user can click on any recipe (hovering will show predicted rating), and see more detailed instructions on how to make it:

Diagram



Data Flow

The images of 26 ingredient types from the image dataset are used to train the classifier. The classifier is used to identify the food items present in the images uploaded by the user, producing a list of ingredients uploaded by the user. The recipe dataset, containing recipe names and previous user interactions, is used to train the recommender. The recipe rating input from the user allows the recommender to produce a list of personalized recipe predictions.

The ingredient list is used for recipe filtering to create a final filtered recipe list that only contains the recommended recipes with the food items present in the ingredient list.

Component Descriptions

Classifier

Our classifier uses transfer learning with a pretrained ResNet18 CNN (Convolutional Neural Network). The model was trained with an AdamW optimizer and a StepLR scheduler to classify images into the following 26 types of ingredients:

- Beef
- Salmon
- Chicken
- Broccoli
- Cabbage
- Carrot
- Celery
- Corn
- Cucumber
- Eggplant
- Green beans
- Bell pepper
- Olive
- Onion
- Spinach
- Tomato
- Lettuce
- Apple
- Avocado
- Banana
- Lemon
- Bread
- Cheese
- Mushroom
- Egg
- Pasta
- Rice
- Potato

As it would take too much time to accumulate a dataset with all ingredients included in the recipe dataset, we chose to accomodate a subset with these 26 ingredients that contains common meats, vegetables, and fruits instead.

Dataset Creation

First, a dataset of ~300 images of each type of ingredient was compiled. Images were scraped with the Bing ImageSearch API and images unsuitable for our purposes, such as duplicate, irrelevant, or low quality images, were then manually filtered out. All images were resized to 224x224 pixels to be compatible with our network.

Classifier Training and Optimization

To start, the model was trained using an SGD optimizer and a StepLR scheduler, which gave a resulting accuracy of 87%.

Next, we experimented with different optimizers, schedulers, and values for learning rate (lr), weight_decay, step size, and gamma. Each test was run for 10 epochs and the results are summarized below:

Optimizer Test Results

Each test was run with the StepLR scheduler and no image transformations besides resizing. All Adam and AdamW optimizers were run with betas=(0.9, 0.999), amsgrad=False and eps=1e-08, combined with varying lr and weight_decay parameters.

Optimizer:	Best Resulting Accuracy
SGD(lr=0.001, momentum=0.9)	87%
AdamW((lr=0.0001, weight_decay=0.01)	86%
AdamW(lr=0.0001, weight_decay=0.02)	88%
AdamW(lr=0.0001, weight_decay=0.03)	87%
AdamW(lr=0.001, weight_decay=0.02)	77%
Adam(lr=0.0001, weight_decay=0.02)	86%

Scheduler Test Results

StepLR and OneCycleLR Schedulers were run with AdamW(lr=0.0001, weight_decay=0.02) optimizer since it was found to give the highest accuracy from the previous tests.

Scheduler	Best Resulting Accuracy
StepLR(optimizer, step_size=5, gamma=0.1)	88%
OneCycleLR(max_lr=0.01, steps_per_epoch=5, epochs=10)	61%

After determining that the StepLR scheduler gave better results, additional testing was done with varying values for step size and gamma.

Step Size & Gamma Test Results

All tests were run for 10 epochs with the StepLR scheduler and AdamW (lr=0.0001, weight_decay=0.02) optimizer. StepSize tests were run with default gamma = 0.1. Gamma tests were then run with the best step size (5).

Step Size	Best Accuracy
4	71%
5	88%
6	70%

Gamma	Best Accuracy
0.08	62%
0.1	88%
0.11	63%
0.12	55%

Finally, we tried data augmentation to improve our accuracy. The following transformations were applied to our dataset to create more samples:

Data Augmentation Results

Pytorch Transformation	Reason for Testing Transformation	Best Resulting Accuracy
RandomHorizontalFlip	Reasonable for ingredients to be facing the other way	80%
Grayscale	Trying to train model to recognize features of ingredients other than color - there may be different colors of ingredients not included in our datasets (ex. bell peppers)	88%
ColorJitter (brightness=0.5, contrast=0.5, saturation=0.5,hue=0)	Images of ingredients may be taken under different lighting, with different quality of cameras	88%

Since data augmentation did not improve accuracy, we chose not to use any transformations to reduce the amount of operations to be performed by the model.

Thus, after experimenting, the best version of our classifier model used an AdamW(lr=0.0001, weight_decay=0.02) optimizer and StepLR(step_size=5, gamma=0.1) scheduler and achieved an accuracy of 88%.

Recommender

We built two recommenders using different approaches. The deep learning recommender model is based on the collaborative filtering model with matrix factorization introduced in class. Another recommender is based on the Rating Frequencies Recommender Algorithm. The deep learning model is trained with an SGD optimizer and an OneCycleLR scheduler to predict users' preferences.

Dataset Creation

The dataset is based on the Food.com Recipes and Interactions dataset from Kaggle, with some filtering and modification. First, the detailed ingredients used in the recipes are simplified to match the list of ingredients (e.g. "Spaghetti" and "Rotini" should both map to "Pasta). Then, based on our selected ingredient set, all the recipes that contain none of the selected ingredients are removed. Next, the recipes with no ratings are removed. User ratings that contained recipes that were removed in the previous steps were also removed. Finally, the remaining recipes, interactions, and ingredients list are pickled for further use.

Optimize Test Results

During optimization, multiple combinations of optimizers with different parameters were tested for 5 epochs. First, Adamax, ASGD, AdamW, and SGD were tested with OneCycleLR and SGD was found to have the best efficiency and least loss. Next, we adjusted the parameters in both the optimizer and scheduler. Finally, we reached the least overall loss by setting lr to 1e-4 and

max_lr to 0.4. Using these parameters, the recommender model using PyTorch produced a training loss of 0.27 and a test loss of 1.40 after 5 epochs.

RF Recommender

Experimentally, the Rating Frequencies Recommender model is much quicker than matrix factorization when running with fewer ingredients. It is suggested to be a (relatively) efficient and effective algorithm for creating predictions on sparse datasets like the recipe interactions dataset. We implemented this not to replace the matrix factorization, as it has not been tested extensively and is not machine learning, but merely as a “bell and whistle” to make using the web app quicker. The algorithm is calculated as (where the items are recipes):

$$\hat{r}_{u,i} = \arg\max_{v \in R} \; (freq_{user}(u,v) + 1 + \mathbb{I}_{user}(u,v)) * (freq_{item}(i,v) + 1 + \mathbb{I}_{item}(i,v))$$

Where $\hat{r}_{u,i}$ is the predicted rating for a given user and recipe. $freq_{user}(u,v)$ and $freq_{item}(i,v)$ are the frequencies of rating value v for a given user or item, respectively. $\mathbb{I}_{user}(u,v)$ and $\mathbb{I}_{item}(i,v)$ are indicator functions that return a 1 if the given rating v is equal to the rounded mean rating for a given user or item, respectively. R is the set of all possible ratings (in our case {0,1,2,3,4,5}).

Recipe Filtering

A user only wants to see recipes they actually have ingredients to make. As such, recipes must be filtered out so the user only sees recipes that contain the ingredients they specify, and none of the supported ingredients they did not specify. Once the matrix factorization recommender is done training, we run the recipes that meet the criteria through the model. In the case of the RF recommender, training is not needed so we only calculate predicted ratings on recipes that meet the criteria. That is why this recommender performs well, especially when run on a few selected ingredients. Once the prediction is done, the recipes are sorted in non-increasing order and delivered to the user.

Website

The recipe recommender website was created using Flask with Python for the backend and HTML,CSS, and JavaScript for the frontend.

Frontend

Frontend for the recipe recommender website was created with HTML and CSS, with JavaScript to incorporate some logic functions. Dynamically generated pages were created using Jinja templating (included with Flask).

The frontend of the website involves an HTML form accepting multiple image uploads. This allows the user to upload images of ingredients, which will be labelled by the ingredient classifier. Once all uploaded images have been labelled, the next page displays the compiled list of available ingredients, along with an HTML form for the user to rate 5 randomized recipes.

After rating, the user is brought to an HTML page showing the real-time training progress. When the training is finished, the page will contain links to recipes made with some or all of the user's available ingredients. Clicking on a recipe will bring you to a more detailed description of the recipe.

Backend

First, images uploaded to the HTML page are sent to the server in a POST request and accessed with Flask's `request.files` dictionary. Flask's `secure_filename()` is used to secure the uploaded files, which are then saved to a folder on the base directory called `image-uploads/sample/`.

To avoid having to retrain every time a user uploads images, the ingredient classifier was trained and pickled. When the website is run, the classifier is loaded and all images in image-uploads are identified with the trained ingredient classifier. The identified labels for each classified ingredient are mapped to the appropriate ingredient name using a dictionary and sent to the next page to be displayed for the user. This list is passed to the next HTML form as a parameter in `render_template` for use when filtering recipes.

Next, when users rate a few random recipes and select the method they want to use, they will see a real time progress indicator while the model trains in the backend. The progress is sent asynchronously to the client via SocketIO. Once training is complete, SocketIO will send the top recommended recipes which will appear seamlessly in the browser.

Contributions

- Alina Shabanova (alinas2000)
    - Image collection for dataset
    - Implemented, optimized, and trained classification model
    - Integrated classifier with website
    - Initial integration of classifier website with recommender website
- Bowei Ren (boweiren)
    - Image collection for dataset
    - Responsible for test and optimization of recommender model
    - Built a recommender website using Flask
    - Worked on the style optimization and CSS file of the final web app
- Harrison Mitgang (hmitgang)
    - Responsible for skeleton code for matrix factorization recommender and recommender dataset creations

    - Implemented RF-Rec
    - Integrated recommenders with website
    - Ensured seamless integration of classification and recommendation components
- Jane Wu (jwu611)
    - Collected images for ingredients dataset
    - Implemented, optimized, and trained classification model
    - Integrated classifier with website
    - Built frontend for image uploads and backend for saving uploads