Recommender System
Food Recipe Search Engine

This is a brief overview of the recommender system we developed for our recipe search engine. It was not integrated into the app due to time constraints and some of the difficulties of working in the containerized environment.

Overview

We implemented a content-based recommender system for our food recipes. Essentially this is a cosine similarity scheme in which unseen recipes that are closest to the recipes in the user's profile would be recommended up to the user.

Preprocessing steps and design choices

For each recipe, the title and ingredient fields were concatenated into a single string, then preprocessed to lowercase and remove non-alphabetical characters. The rationale here is that title and ingredients would provide the best sense of the similarity between recipes and culinary preferences more generally. Since the ingredients field contains a lot of noise and would also have been much more expensive in terms of compute and space overhead, this was a relatively obvious design choice for us.

Following concatenation of the title and ingredients field, we tokenized each word, and performed stop-word removal. The stop-words were adapted from NLTK's stop word list, which covers very frequent English words. We added 32 words that we found commonly in our dataset that were obviously non-informative for distinguishing food recipes. These include words like, "tablespoon", "cups","advertisement", etc.

The next step was forming a corpus vocabulary from the preprocessed recipe documents, where every unique word in the vocabulary was stored in an index with the associated number of documents in which each word occurs. Concurrent with this step, each recipe document was converted from a list of words to an index of unique words with an associated word count for each unique word in the document. The average document length of the corpus was also computed during the vocabulary build process.

The final step was performing a sub-linear transformation on the document vectors, converting the unique word counts in each document into a score for each term that captured the TF-IDF weighting for each term in each document. The BM25 formula was a natural fit for this transformation.

To save on compute resources, this collection preprocessing step was done once with the results written to file in JSON format for easy retrieval during the recommendation step.

Recommendation step

For the recommendation step, a collection of recipe IDs are provided as the user profile input as well as the number of recommendations that the user would like returned. Then, the corpus is split into “seen” and “unseen” documents, where the “seen” documents are the list of recipes in the user profile.

For each seen document, the dot product of the weighted recipe term vectors is computed for that document with each document in the unseen documents collection. The documents that return the overall k-highest scores are stored and returned as the algorithms output.

Test Profile

'California Club Turkey Sandwich'
"Slammin' Salmon"
'Molasses Cookies'
'Asian Chicken Salad'
'Holiday Chicken Salad'
'Jamaican Turkey Sandwich'
'Perfect Pumpkin Pie'
'Best Guacamole'
'American Lasagna'
'Cooky Cookies’

Test Recommendations

'Easiest, Amazing Guacamole'
'Party Guacamole'
'Simple Pumpkin Pie'
'California Guacamole Hummus'
'Baked Pumpkin Custard ‘

Analysis

These results are somewhat to be expected, since guacamole and avocado and pumpkin are relatively uncommon ingredients and would score quite highly, particularly in if they were short recipes. We suspect that several performance improvements could be achieved with some BM25 parameter turning, expansion of the stop-words list and perhaps even introducing smoothing into the scoring algorithm. Overall, though, we were pleased to complete a working recommender that returned reasonable results.

Further use and testing

As we did not design this recommender system to be integrated with our application, this was included as an “extra’. if you wish to explore this module on your own, you will need to make some minor changes to the code. They are as follows:

1) You need to make copies of the datasets with the “.” Removed from each recipe ID field. This is because the raw dataset we used is fed into Elasticsearch with the periods removed dynamically during ingest without updating our datasets. Batch -> “es_data_loader.py” should give you a nice template for how to complete this step.

2) Update filepaths:
    batch -> recommend_preprocessor.py
        ln 27
        ln 125
    batch -> term_vector_rec.py
        ln 87
        ln 115
        ln 116

3) Run recommend_preprocessor.py

4) Use test recipes for identical output

5) For a customized output, find the recipe IDs associated with your search results in the output of your Docker shell window. Just copy and paste the recipe IDs that you want to test into the term_vector_rec.py (lines 69-78)

6) Run term_vector_rec.py

7) Steps 1-3 are one-time activities, repeat steps 5 and 6 to run more tests with different IDs