toMEto: a Networks-based Approach to Recipe Recommendation

CS 145

Albert Ge
California Institute of Technology
Pasadena, CA
age@caltech.edu

Matthew Jin
California Institute of Technology
Pasadena, CA
mjin@caltech.edu

Jonathan Joo
California Institute of Technology
Pasadena, CA
jjoo@caltech.edu

Boyu (Charlie) Tong
California Institute of Technology
Pasadena, CA
bttong@caltech.edu

ABSTRACT
toMEto is a web-based application that suggests ingredients to supplement existing food recipes. We develop and evaluate several network based algorithms, packaging the suggestions under an intuitive and friendly user interface.

1. INTRODUCTION
Traditionally, cooking involves a combination of technical skills and creativity. While technical skills can only be improved through time and practice, we believed that the creative aspect of cooking could be improved through utilizing the wealth of existing recipes available online. There are a variety of web and mobile applications that mostly function as a database for recipes, and allow users to quickly search to recipes pertaining to their desires. There are also more advanced applications that devote specifically to *food pairings*; websites such as `foodpairing.com` allow users to experiment with different ingredients and rate their synergy by examining the flavor network. On another hand, `yummly.com` will recommend new recipes based on a variety of selection filters, such as allergies or diets.

The purpose of this project is to be able to blend the two and suggest individual ingredients, as opposed to entire recipes, using the concept of food pairings found through an ingredient network. Thus, through construction of a network using this information, toMEto offers unique suggestions to create interesting new twists on already successful recipes. Having these suggestions available alongside a recipe allows toMEto to inspire new forms of cooking styles, and improves the creative cooking experience for beginners and advanced chefs alike. With a simple, intuitive user interface, along with a rich database of recipe information, toMEto allows for a streamlined recipe lookup experience, with the added functionality of optimized suggestions.

The rest of the paper is organized as follows. Section 2 contains details on data scraping and other preprocessing steps. Section 3 describes the various algorithms considered. Section 4 covers the app's web design while Section 5 covers the backend. Section 6 concludes our paper.

1.1 Related Work
There have been a select number of papers discussing analysis of food and ingredient networks. In the literature, there are two common food networks that subject to network analysis. The first is a *co-occurrence network*, which relates ingredients by how often they appear together in user-defined recipes. The other is a *flavor compound network*, which relates ingredients by how similar their chemical composition is. Throughout this paper, we will refer to the co-occurrence network as the *ingredient network*, and the flavor compounds network as the *flavor network.* Teng et al. [4] describes the goal of being able to recommend entire recipes by examining communities of a co-occurrence network and predicting the success of a particular recipe's rating.

Another paper by Ahn et al. [1] uses a flavor-compounds network to compare the differences in cuisine and make general statements about the co-occurrence of ingredients by their chemical composition. They discovered that Asian recipes tend to have ingredients with different compounds, whereas Western cuisines have ingredients with like compounds.

Still other research is ongoing, some building upon the relevance of flavor compounds in regional cuisines [5], [3], and others attempt to implement these ideas to help users make suggestions about healthy food choices [2].

Within this web of research, we discovered a potential niche for recommending individual *ingredients*, as opposed to entire recipes. This combines both the practical application of current research on food networks, as well as the theory behind finding ingredients that complement each other.

2. DATA PROCESSING
2.1 Data source
Our application's source of data are the online recipe websites New York Times (NYT) Cooking and AllRecipes.com. We gained over 13,000 and 52,000 recipes by crawling and scraping these two websites respectively. This was done by a script written in `scraper.py`.

2.2 Initial parsing

Our application only requires the title, ingredients, and body of each recipe, so we extracted this data from each webpage using a simple script written in `parser.py`.

However, this extracted data was not in a clean format that our algorithms could easily use. For example, ingredients were often combined with quantifiers and other adjectives:

- *medium-size russet potato, about 10 ounces, peeled and diced*
- *shrimp, shelled and cut into bite-sized pieces*
- *kale, stemmed, rinsed and coarsely chopped to make 6 cups*

These ingredients would only occur maybe once or twice in the entire dataset, resulting in many "rare" ingredients. In the NYT dataset (12402 unique ingredients), approximately 70% of the unique ingredients occurred only once, and 90% occurred less than 10 times. We realized this resulted in suboptimal performance, as these ingredients would not be counted correctly in the ingredient network and many edges would not exist in the network.

2.3 Mapping

To fix this, we developed a script in `nyt_mapper.py` to find these "rare" ingredients and map them to their root ingredient. For the example ingredients from above, we mapped:

- *medium-size russet potato, about 10 ounces, peeled and diced* $\rightarrow$ *potato*
- *shrimp, shelled and cut into bite-sized pieces* $\rightarrow$ *shrimps*
- *kale, stemmed, rinsed and coarsely chopped to make 6 cups* $\rightarrow$ *kale*

The script attempts to extract the root ingredient for each entry by combining several strategies. For example, some strategies that we use are:

- Removing parenthetical tokens
  - *vanilla extract (to taste)* $\rightarrow$ *vanilla extract*
- Removing comma separated descriptors
  - *honey, preferably wildflower* $\rightarrow$ *honey*
- Picking the best of two ingredients separated by 'or'
  - *sriracha or other hot sauce* $\rightarrow$ *sriracha*
- Finding a sole 'top' ingredient name
  - *coarsely grated carrot* $\rightarrow$ *carrots*

Using the mappings generated by `nyt_mapper.py`, we reduced the unique ingredient count by 75%. This resulted in a much more densely connected network, and better results in our ingredient recommendations.

2.4 Supplementing with AllRecipes data

We initially focused our application on the NYT dataset due to the relatively cleaner data and ingredients list. Once our basic algorithms were up and running on this dataset, we moved to supplement it with the much larger AllRecipes dataset (52,000 recipes). We reasoned that this would further improve the ingredient network by providing a more accurate count of ingredient frequencies. We also expected it to introduce new connections (and thus new, creative suggestions), since there may be recipes and foods that are not found in the NYT dataset.

As our application does recommendations only for NYT recipes, we modified our approach to processing the AllRecipes data accordingly. Again, we wrote a mapping script in `allrecipes_mapper.py`, but instead of mapping to only the root ingredients, the script also attempted to map to the top ingredients in the NYT dataset. This was important because our goal ultimately was to develop an ingredient network for the NYT dataset, and we did not want the ingredients of the AllRecipes dataset to be disjoint.

The AllRecipes mapper employed many of the same techniques as the NYT mapper, with some additional strategies. The recipes from the AllRecipes websites list the ingredients with numbers and quantifiers, for example:

- *1 pound ground beef*
- *1/2 teaspoon black pepper*
- *1 (25 ounce) package frozen cheese ravioli*

Thus, we added filters that removed numerical digits from the raw ingredient name and also searched for and removed quantifiers such as "pound", "teaspoon", or "ounce". To map the ingredients to the NYT ingredients, we took a set of the most common NYT ingredients and searched for corresponding tokens in the AllRecipes ingredients. If there was a 1-to-1 matching, then we considered it a mapping.

Combining both datasets resulted in a network with 7339 unique ingredients. Of these, we selected the top 1000 which approximately corresponded to the set of ingredients with at least 10 occurrences. For several of our algorithms, we restricted the recommendations to be from within this "top set" in order to have meaningful suggestions.

3. ALGORITHM DESIGN AND ANALYSIS

3.1 Algorithm Design

The primary application of algorithm design resides in ingredient recommendation to the user. These algorithms were implemented in a collection of files with the pattern `analyzer_*.py`, and each file corresponds to a different algorithm (as we enumerate below). We reserve the majority of this section to understanding the different frameworks we tried for the *complement* method, which is a method that takes in a particular recipe ID, and suggests up to ten ingredients that the user could potentially add.

Before we do so, however, we first introduce some common terminology to be used throughout the text:

1. *w(x,y)* describes the weight of the edge between two ingredients. We define the weight of an edge as the co-occurrence of two ingredients; that is, how many times the two ingredients appear in the same recipe.

2. *graph* is the entire ingredient network.

3. *peripheral ingredient* describes those ingredients that are not part of a recipe, but have at least one edge (with nonzero weight) to an ingredient in the recipe (Fig. 1). Formally, this is described as

$$\{k : \text{k} \in \text{adjacent neighbors of recipe} \wedge k \notin \text{recipe}\}$$

4. *compatibility score* is a score computed (using different metrics) that is used to rate each ingredient suggestion. An ingredient with high "compatibility" with another suggests that the pair are commonly used together, and therefore would be a good fit for the recipe.

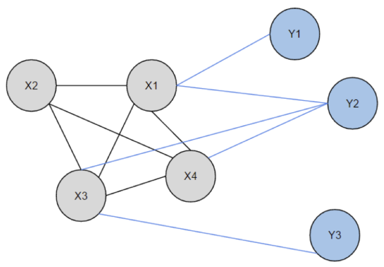


Figure 1: Peripheral ingredients. Nodes denoted in gray are part of recipe X, while ingredients in blue are peripheral ingredients with connections to some of the recipe ingredients.

There are two primary schools of thought when designing an appropriate algorithm for recipe recommendation - using degree centrality measures, and using pointwise mutual information (PMI). Here, we enumerate both of these algorithms in detail.

First, we discuss the degree centrality algorithm. We apply degree centrality toward compatibility scores of ingredients:

1. Degree Centrality (Algorithm 1). The key aspect of this algorithm was to compute a compatibility score of an ingredient by summing over all its connections with ingredients in the recipe. For two ingredients (a, b), we compute the weight of the connection $w(a, b)$ by counting the number of occurrences in which a, b appear together in the same recipe. Each connection is normalized by dividing by the highest weighted number of connections of the two ingredients; this helps prevent other high-frequency ingredients from dominating the recommendation. To compute the the weighted number of connections of an ingredient a, we sum the weights over all the connections with its neighbors: $\sum_{b \in N(a)} w(a, b)$.
Formally, we write, for a pair of ingredients (i, k) with i being a recipe ingredient and k a peripheral ingredient,

$$score = \sum_{i \in recipe} \min(\frac{w(i,k)}{degree(i)}, \frac{w(i,k)}{degree(k)})$$

This algorithm was the first variant that we attempted on ingredient recommendation. First, we obtain a list of all peripheral ingredients of a particular recipe. Here, we exclude the top ten ingredients as part of our search. The idea was to exclude potentially obvious or very similar ingredients from our search - we wanted to suggest something that would introduce variability to our suggestions, rather than recommend the same high-frequency ingredients. Additionally, we must check if ingredients exist in the network, as our data processing eliminates certain ingredients from the graph but doesn't eliminate them from the recipes themselves.

Algorithm 1 Degree Centrality algorithm

```
1: function NAIVE($recipeID$)
2:     Let $recipe$ = recipe with ID $recipeID$
3:     Let $top10$ = ten ingredients with highest weighted number of connections
4:     Let d = dictionary to contain all ingredients and their compatibility score
5:     for $i$ in ingredients of $recipe$ do
6:         if $i$ in graph and $i$ not in $top10$ then
7:             for peripheral ingredient $k$ do
8:                 $d[k] = d[k] + \min(\frac{w(i,k)}{degree(i)}, \frac{w(i,k)}{degree(k)})$
       return 10 ingredients from $d$ with highest compatibility score
```

Since this algorithm takes an arbitrary standard of excluding the top ten ingredients from recommendation, this was initially seen as a naive algorithm for recipe recommendation. Hence, we turned to a much more extensive analysis of using PMI as the basis of our algorithms. The general formula of PMI, between two ingredients a, b is given by:

$$\log \frac{p(a,b)}{p(a)p(b)}$$

where $p(a)$ is defined as $\frac{\text{number of occurrences of a}}{\|graph\|}$. Intuitively, this value between two ingredients describes how likely two ingredients are to appear together in the graph. Two ingredients with very few connections could still have a high value according to this metric, if they appear together more often than they do not. Thus it was thought that this metric captures ingredient pairings much better, as now there isn't any need to remove the most frequent ingredients.

As in degree centrality, we use PMI mainly for computing compatibility scores of ingredients.

Here we enumerate the algorithms using PMI:

1. Normalized PMI. After computing a score using PMI, normalize the value by the weighted number of connections. This algorithm is the exact same as in Algorithm 1, only the weight $w(a, b)$ is subsituted for $PMI(a, b)$. This was mostly meant to compare directly with the the degree centrality algorithm.

2. Generalized PMI (Algorithm 2). This is an extension to the pairwise PMI score, and generalizes it PMI between more than 2 ingredients. For an n-tuple of ingredients in the recipe, we compute sum of all their weighted connections with each other $\sum_{a,a' \in tuple} w(a, a')$. We then find the tuple whose sum of weights is the

greatest - this captures the ingredients that are seen together most often in recipes, and gives us a sense of the “essential” ingredients in the recipe. Then, using these essential ingredients, we can compute a generalized PMI score for each peripheral ingredient that has connections to all of these essential ingredients, which is given by

$$PMI(tuple) = \log \frac{p(x \in tuple)}{\prod_{x \in tuple} p(x)}$$

In practice, we found that letting $n = 3$ for determining the size of our tuple was best, as it as large as we could get without too many division by zero errors.

3. Weighted PMI (Algorithm 3). This weights an ingredient’s PMI score by the number of edges that exist between itself and ingredients of the recipe. Formally, for a peripheral ingredient b we describe this as

$$(\sum_{a \in recipe} \mathbb{1}_{(a,b) \in graph})(\sum_{a \in recipe} PMI(a, b))$$

Thus, for each connection to an ingredient in the recipe, we increase the weight factor by 1. Intuitively, this assigns more importance to peripheral ingredients that pairs well with many of the recipe’s ingredients.

4. Minimax PMI. This algorithm was meant to consider safe suggestions; that is, recommending peripheral ingredients that would be compatible with all recipe ingredients, instead of ingredients that would work very well with some and not at all with others in the recipe. To do this, for a peripheral ingredient b we computed $\min_{a \in recipe} PMI(a, b)$, and then returned the ingredients with the highest min values. This way, we could rule out ingredients that would never pair well with at least one recipe ingredient.

Algorithm 2 Generalized PMI algorithm

```
function genPMI($recipeID$)
    Let $recipe$ = recipe with ID $recipeID$
    Let $top10$ = ten ingredients with highest weighted number of connections
    Let d = dictionary to contain all ingredients and their compatibility score
    $bestMatch = \max_{(a,b,c) \in recipe} w(a,b) + w(b,c) + w(a,c)$
    $tuple = \text{argmax}_{(a,b,c) \in recipe} w(a,b) + w(b,c) + w(a,c)$
    for $i$ in ingredients of $tuple$ do
        for peripheral ingredient $k$ and $a, b, c \in N(k)$ do
            $d[k] = PMI(k, tuple)$
    return 10 ingredients from $d$ with highest compatibility score
```

3.2 Analysis

Furthermore, we compared these algorithms using an objective test. First, we separated the dataset into two parts. The first part, used for “learning”, consisted of 70% of the dataset, and the second part, used for testing, consisted of 30% of the dataset. The separation of this dataset was done randomly to ensure that there was no bias in the learning and testing sets. (For example, we wanted to avoid a case where all the baked goods ended up in the learning set.) Then, we applied our algorithms on the the learning set, and tested their recommendations on the testing set, with each recipe having one ingredient removed. We restricted the removed ingredients to be from the top 1000 ingredient set because our recommendations are drawn only from that set. The algorithms were evaluated on their ability to recover the missing ingredient. This gave us an objective and fair test between the algorithms, and we found that the *weighted PMI algorithm* was the most successful.

Algorithm 3 Weighted PMI algorithm

```
function wPMI($recipeID$)
    Let $recipe$ = recipe with ID $recipeID$
    Let $top10$ = ten ingredients with highest weighted number of connections
    Let d = dictionary to contain all ingredients and their compatibility score
    for $i$ in ingredients of $recipe$ do
        if $i$ in graph then
            for peripheral ingredient $k$ do
                $d[k] = d[k] + \min(\frac{PMI(a,b)}{degree(b)}, \frac{PMI(a,b)}{degree(a)})$
                $count[k] = count[k] + 1$
    for ingredient in $d$ do $d[k] = d[k] \cdot count[k]$
    return 10 ingredients from $d$ with highest compatibility score
```

However, it was only able to recover the missing ingredient about 20% of the time. While this indicates some failure with our algorithms, it is really a misalignment of objectives. Our algorithms try to recommend a novel ingredient to spice up a dish, whereas the algorithm analyzer awards points for recovering a missing ingredient. These use cases, albeit similar, are clearly different. Thus, through the construction of this alanlyzer, we observed the difficulty of implementing an objective test that evaluates the accuracy of our algorithms. Improvements to such algorithm analyzing techniques may be of interest to study in future work.

4. WEB DESIGN

Web design was done using a combination of HTML, CSS, JavaScript, and Python. HTML was used to create the content for display on the page, and CSS was used to style this content. JavaScript was also used for animations and general User Interface tweaks to make browsing the site seamless and intuitive. Finally, Python was utilized as an interface between the back-end and the front-end. Through the usage of Python’s Flask framework, it was possible to integrate the back-end algorithms with displaying the relevant computed information on the front-end. In other words, our module app.py imported modules from the back-end, while also using this information to fill in the HTML templates based on search queries, etc. provided by a user in the front end.

The primary focus on the front end was to develop a site that is intuitive and self-explanatory, while also featuring only the information that is needed the most. Thus, the landing page has the following design:

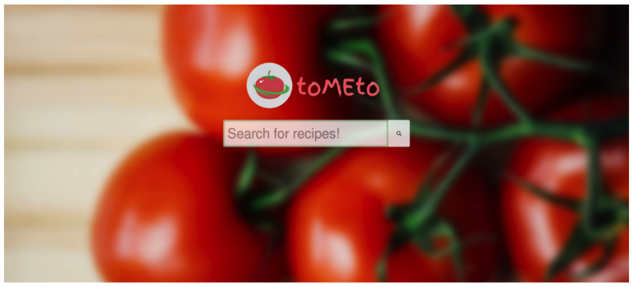


Figure 2: Landing page for toMEto.

Upon searching for a recipe, a loading bar appears, and once recipe information is obtained, tiles fade in. These tiles offer an image of the recipe to be prepared, with the recipe title overlaid on top. This can be seen below:

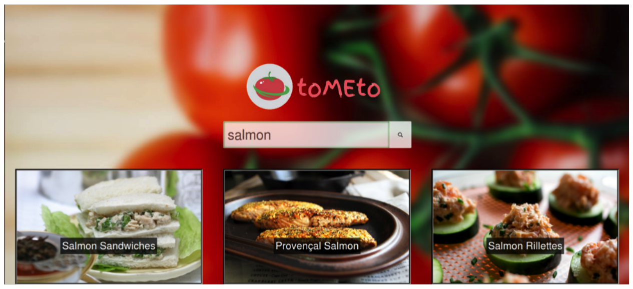
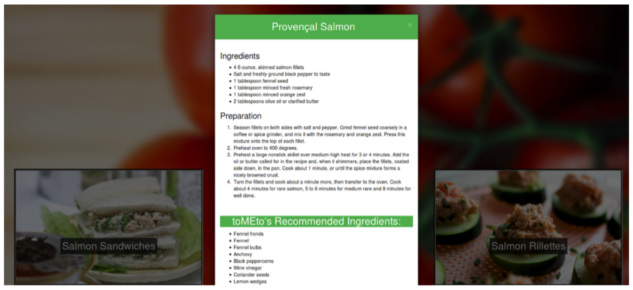


Figure 3: Tiles, each with a recipe related to the search input.

Upon clicking a recipe, a modal popup appears, which lists both the recipe itself as well as toMEto's recommended ingredients, as can be seen in the below image:



Provençal Salmon

Ingredients

- 4 6-ounce, skinned salmon fillets
- Salt and freshly ground black pepper to taste
- 1 tablespoon fennel seed
- 1 tablespoon minced fresh rosemary
- 1 tablespoon minced orange zest
- 2 tablespoons olive oil or clarified butter

Preparation

1. Season fillets on both sides with salt and pepper. Grind fennel seed coarsely in a coffee or spice grinder, and mix it with the rosemary and orange zest. Press this mixture onto the top of each fillet.
2. Preheat oven to 400 degrees.
3. Preheat a large nonstick skillet over medium-high heat for 3 or 4 minutes. Add the oil or butter called for in the recipe and, when it shimmers, place the fillets, coated side down, in the pan. Cook about 1 minute, or until the spice mixture forms a nicely browned crust.
4. Turn the fillets and cook about a minute more, then transfer to the oven. Cook about 4 minutes for rare salmon, 5 to 6 minutes for medium rare and 8 minutes for well done.

toMEto's Recommended Ingredients:

- Fennel fronds
- Fennel
- Fennel bulbs
- Anchovy
- Black peppercorns
- Wine vinegar
- Coriander seeds
- Lemon wedges

Figure 4: The resulting popup from clicking on an image, with the recipe and our recommended ingredients.

Much of the front-end was designed to be easily updated and maintained, and thus small UI tweaks, such as background images, color schemes, etc. are easily changeable by editing the HTML and/or CSS files. The front-end utilizes mainly three different HTML templates:

- `simplesearch.html`
- `simplesearch_searched.html`
- `no_results.html`

`simplesearch.html` is simply the html for the landing page, before any queries are entered. Then, once a search query is entered, app.py directs this information to the backend, which then generates information which is supplied to the `simplesearch_searched.html` template. The user is also redirected to this template, which includes the tiles, modal popup information, etc. Furthermore, if a search is entered into `simplesearch_searched.html`, this also refreshes the `simplesearch_searched.html` template, utilizing the new information. Finally, `no_results.html` is used as a template to be redirected to when the query entered into `simplesearch.html` or `simplesearch_searched.html` does not contain any results. As a note, a user will be redirected to `simplesearch.html` upon clicking the toMEto logo.

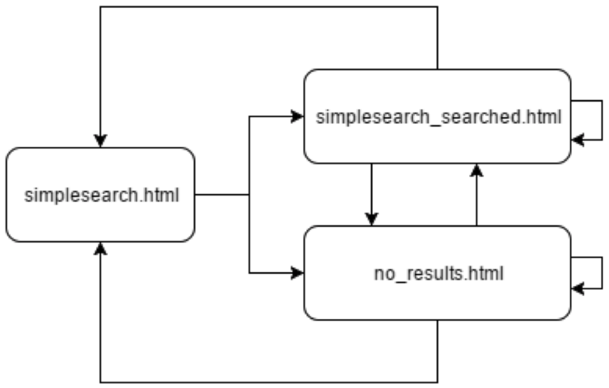


Figure 5: Diagram of front-end architecture. `simplesearch.html` **is the landing page of the website, and after a search query, the webpage will then transition into either** `simplesearch_searched.html` **if there are search results, or** `no_results.html` **if there are no results. Additional search queries will just redirect into the corresponding HTML page.**

5. BACKEND ARCHITECTURE

5.1 Integration with Frontend

A diagram of our backend architecture is described in Figure 6. We first launch the service by calling `app.py`, as well as invoking the appropriate `analyzer_*.py` program whose algorithm we wish to display. Shortly after, the landing page for toMEto is brought up. The user then can enter a search query, which gets passed through `app.py`, and then `search.py` in the backend. The relevant recipes are returned back through `app.py`, which then looks through a text file `analyzer_*.txt` to find our recommended ingredients for the returned recipes. Finally, these are displayed to the user as a webpage.

5.2 Backend processing

As described in Section 2 of this paper, we use the following modules to extract and obtain relevant recipe information, to be fed to `app.py`.

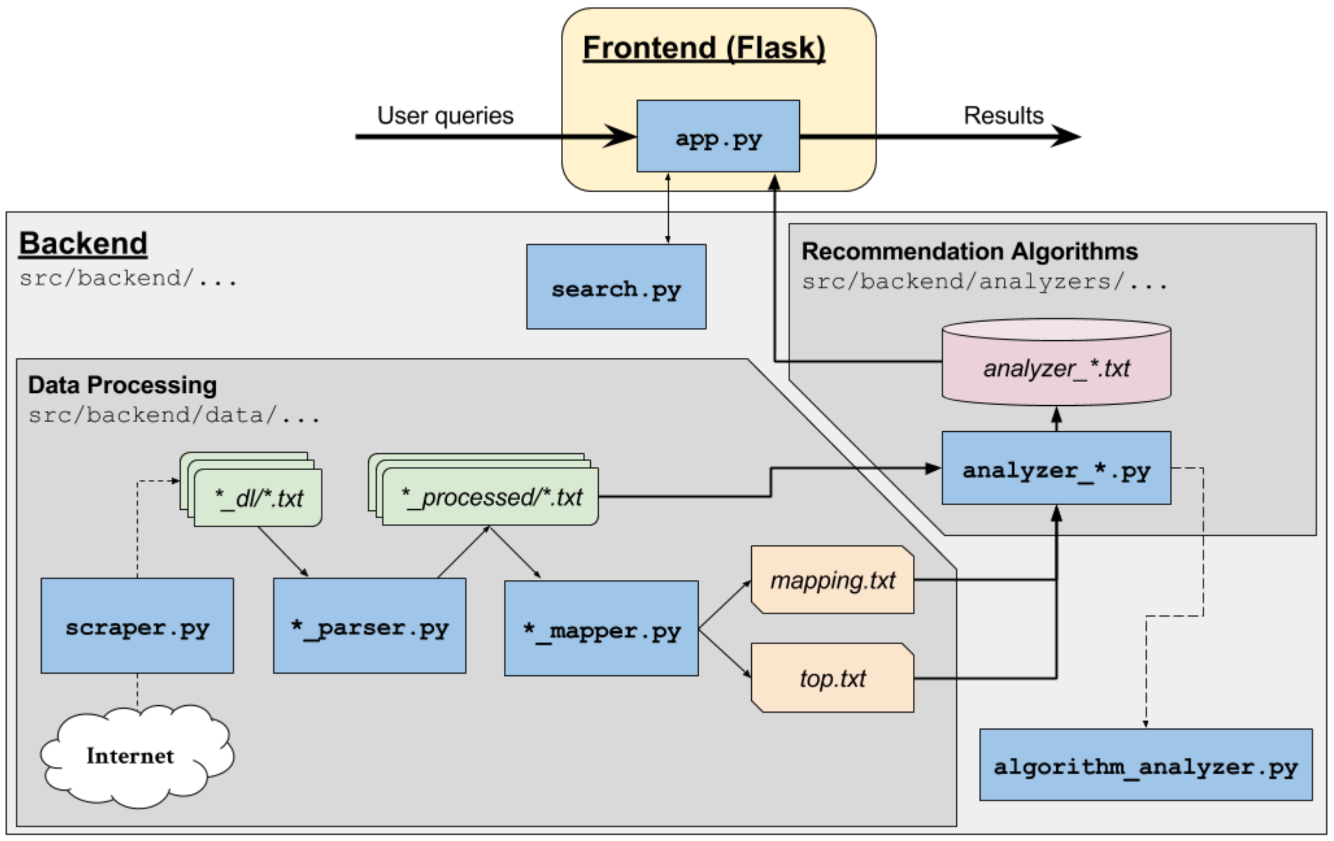


Figure 6: Schematic of the architectural design.

1. First, use module `scraper.py` to scrape recipes from the Internet and store raw data in a folder `dl/`.

2. Then, we employ `parser.py` to parse out the relevant recipe data and store the recipes in a folder `processed/`.

3. `mapper.py` is run to create two files: `mapping.txt`, which maps similar ingredients to the same ingredient, and `top.txt`, which lists the 1000 most common ingredients.

4. These are fed into the `analyzer_*.py` files, which produce the ingredient networks and run the relevant algorithms described in Section 3. These functions will run the algorithm on each recipe obtained from NYTimes Cooking, and save all the results in the corresponding `analyzer_*.txt` files.

6. CONCLUSIONS AND FUTURE WORK

The final result of our project is a web based application that suggests supplementary ingredients to existing recipes. The novel part of our application lies in our algorithms, where we use purely network metrics to rank ingredient suggestions. The generalized PMI algorithms had the most success in providing solid, consistent recommendations.

We also attempted to relax the standard for determining if suggested ingredients were close enough in similarity to the actual removed ingredient. Here, we defined a *minimum edit distance*, which is the minimum number of insertions, deletions or substitutions of individual characters between two strings (Algorithm 4). We write the code for this in `backend/levenshtein.py`, which computes the minimum edit distance between all distinct pairs of the 1000 most common ingredients.

Algorithm 4 Minimum edit distance algorithm

```
function LEVENSHTEIN($a[1 \dots m], b[1 \dots n]$)
    delete = ($Levenshtein(a[1 \dots m-1], b[1 \dots n]) + 1$)
    insert = ($Levenshtein(a[1 \dots m], b[1 \dots n-1]) + 1$)
    substitute = ($Levenshtein(a[1 \dots m-1], b[1 \dots n-1]) + \mathbb{1}_{a_m \neq b_n}$)
    return min($delete, insert, substitute$)
```

The idea behind minimum edit distance is to use dynamic programming to determine the shortest number of single-edits required. This is done by recursively solving the subproblems which arise from deleting a single character from one string, or a single character from both strings.

However, this analysis also didn't appear that fruitful either, because this requires an assumption about similar ingredients being also lexicographically similar. However, for instance, c̈hickenänd b̈eefäre likely to be easily substituted, but do not share many characters in common.

There are also several features that could be implemented to further improve toMEto. At the moment, recipes are found using a database. Allowing the user to specify his own recipe with a set of ingredients and then recommending supplementary ingredients would be a very useful feature. This would not be very difficult to implement with our current architecture. The one challenge here is restricting or preprocessing user inputs to match those in the existing ingredient network.

We could also parse the dataset and classify entries, allowing users to get suggestions for a certain type of recipe. For example, some classifications could be how the recipe is cooked (baked, fried, etc.) or the type of cuisine (Asian, Italian, etc.). Then, suggestions that appear will be restricted to the type of recipe the user is interested in.

Furthermore, we could allow user personalization. By allowing the user to upvote or downvote suggestions, we may be able to utilize machine learning techniques to determine what kind of additions the user likes. This would require new algorithms to supplement our existing network algorithms, as well as a more complex user/login-based infastructure.

These additions may offer a more featureful and useful application. However, the work covered in this paper offers a solid foundation for accomplishing our initial objectives, and aids the cooking experience through suggestion of creative, relevant ingredients.

7. REFERENCES

[1] Y.-Y. Ahn, S. E. Ahnert, J. P. Bagrow, and A.-L. Barabási. Flavor network and the principles of food pairing. *Scientific Reports*, 1, 12 2011.

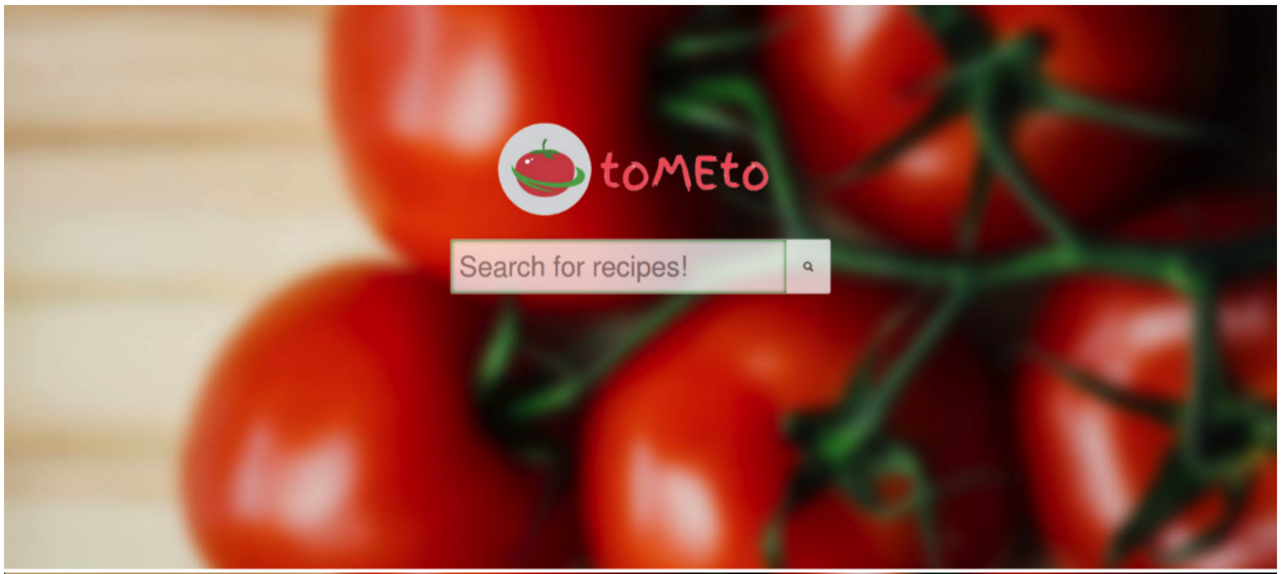
[2] G. Geleijnse, P. Nachtigall, P. van Kaam, and L. Wijgergangs. A personalized recipe advice system to promote healthful choices. In *Proceedings of the 16th International Conference on Intelligent User Interfaces*, IUI '11, pages 437–438, New York, NY, USA, 2011. ACM.

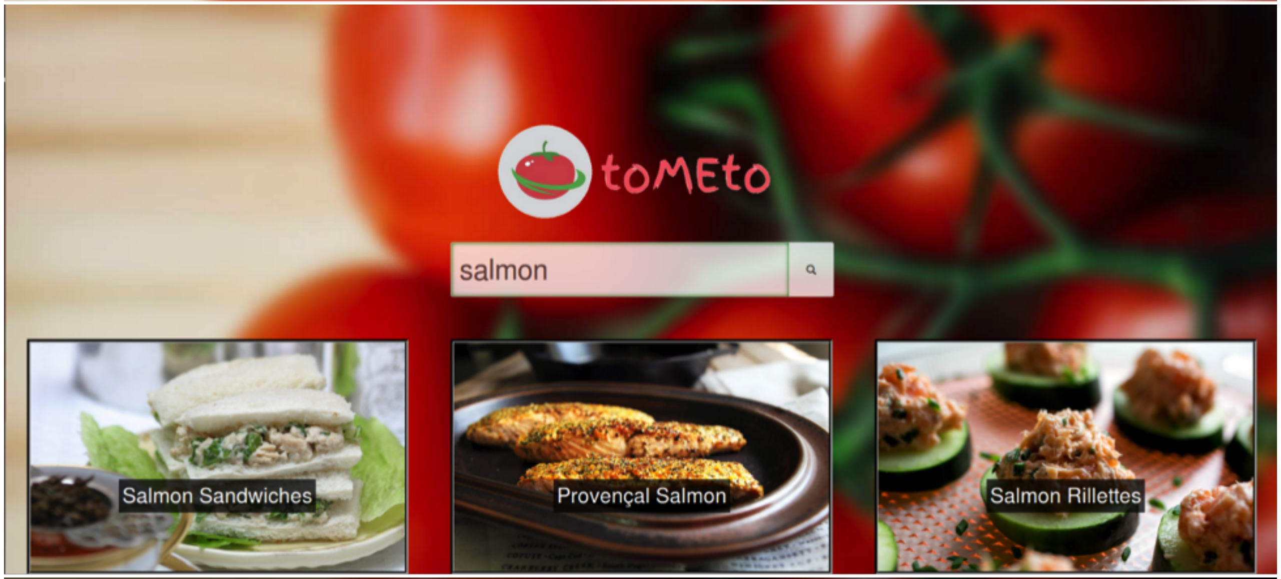
[3] A. Jain, N. Rakhi, and G. Bagler. Analysis of food pairing in regional cuisines of india. *PloS one*, 10(10), 2015.

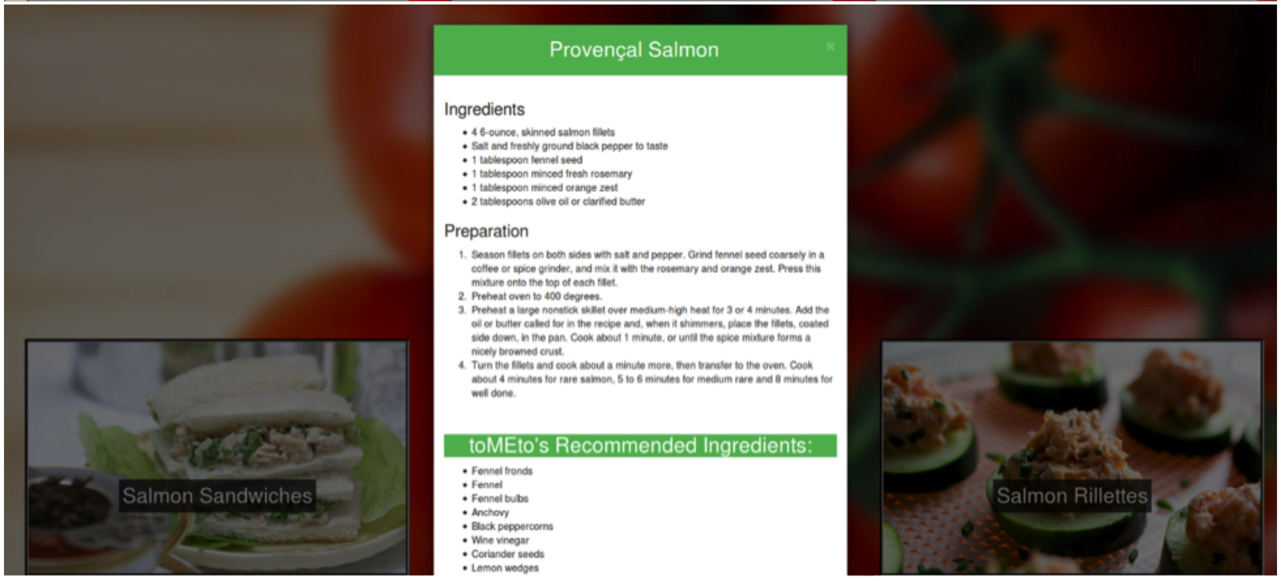
[4] C. Teng, Y. Lin, and L. A. Adamic. Recipe recommendation using ingredient networks. *CoRR*, abs/1111.3919, 2011.

[5] Y.-X. Zhu, J. Huang, Z.-K. Zhang, Q.-M. Zhang, T. Zhou, and Y.-Y. Ahn. Geography and similarity of regional cuisines in china. *PloS one*, 8(11), 2013.

Appendix A: Larger versions of Figures 2-4







Provençal Salmon

Ingredients

- 4 6-ounce, skinned salmon fillets
- Salt and freshly ground black pepper to taste
- 1 tablespoon fennel seed
- 1 tablespoon minced fresh rosemary
- 1 tablespoon minced orange zest
- 2 tablespoons olive oil or clarified butter

Preparation

1. Season fillets on both sides with salt and pepper. Grind fennel seed coarsely in a coffee or spice grinder, and mix it with the rosemary and orange zest. Press this mixture onto the top of each fillet.
2. Preheat oven to 400 degrees.
3. Preheat a large nonstick skillet over medium-high heat for 3 or 4 minutes. Add the oil or butter called for in the recipe and, when it shimmers, place the fillets, coated side down, in the pan. Cook about 1 minute, or until the spice mixture forms a nicely browned crust.
4. Turn the fillets and cook about a minute more, then transfer to the oven. Cook about 4 minutes for rare salmon, 5 to 6 minutes for medium rare and 8 minutes for well done.

toMEto's Recommended Ingredients:

- Fennel fronds
- Fennel
- Fennel bulbs
- Anchovy
- Black peppercorns
- Wine vinegar
- Coriander seeds
- Lemon wedges

Appendix B: Software Architecture

Directory	Contains	Notable files
`src/`	Root of toMEto source code	`app.py`
`src/static`	Images, CSS, JavaScript used by the frontend (Flask)	
`src/templates`	HTML templates used by the frontend (Flask)	
`src/backend`	Backend scripts	`algorithm_analyzer.py` `search.py`
`src/backend/data`	Data processing	`allrecipes_mapper.py` `allrecipes_parser.py` `nyt_mapper.py` `nyt_parser.py` `scraper.py`
`src/backend/analyzers`	Different recommendation algorithms	`analyzer_*.py` `utils.py`

Table 1: Source code organization

Module Name	Language	Author	Lines of Code	Description/Functionality
`data/scraper.py`	Python	Charlie	100	Scrapes for recipes off NYT and Allrecipes websites
`data/allrecipes_parser.py`, `data/nyt_parser.py`	Python	Charlie	150	Extracts relevant information (title, ingredients, body) from scraped recipes
`data/allrecipes_mapper.py`, `data/nyt_mapper.py`	Python	Charlie	400	Finds top ingredients and maps uncommon ingredients to real ones (improves data quality)
`analyzers/analyzer_*.py`	Python	Charlie, Matthew, Albert	600	Different algorithms for recommendations
`analyzers/utils.py`	Python	Charlie, Matthew, Albert	200	Utility functions used by different files
`backend/algorithm_analyzer.py`	Python	Matthew, Albert	200	Objectively tests differences between algorithms
`backend/search.py`	Python	Matthew	20	Handles user search queries (forward to NYT search)
`src/app.py` and CSS/HTML	Python/Flask, CSS, HTML, Javascript	Jonathan	600	Entry point to the program, used to start up the toMEto service (built using Flask Framework)

Table 2: List of modules used in toMEto.