Recipe Search Engine
Group 15

Bence Mogyoródi, Dimitris Christodoulou, Nikolas Pilavakis, Petr Manas

Abstract

We have built a search engine for recipes from various online sources, combined with a cuisine classifier for these recipes. The project is a web application built in Python with the help of the Django Framework and several supporting libraries, set up as four composed Docker containers, and hosted on DigitalOcean. Our dataset contains about 380,000 recipes scraped from 24 popular recipe websites which have been pre-processed to extract the important information. The raw data is then indexed into a PostgreSQL database along with key information about recipes, ingredients, and assigned tags (such as the cuisine).

The user can search using a text query and a set of ingredient constraints, i.e., ingredients to include, exclude, and must-have. The text queries are handled using ranked search using the Okapi BM25 ranking function, and ingredient constraints are imposed using Boolean search. If the user mistypes their query, spelling is autocorrected up to two edit distances using the Damerau-Levenshtein distance metric. The retrieved recipes can be further filtered by their tags, which are often the cuisine they belong to. Since only a small portion of our raw data includes this information, we have trained a Logistic Regression cuisine classifier and used it to produce new cuisine labels for our recipes. Finally, we evaluated the performance and precision of the retrieved results for several queries.

Web Application URL

Our project is available on: http://138.68.178.55/

Source code is available on:
https://github.com/bmogyorodi/recipe_search

1. Technology Used & Deployment

Our project is written in Python 3.8 using the Django Framework[1] 3.1.4 and several supporting Python packages, which will be outlined in the relevant sections. Django enables rapid web development and provides us with the basic web server architecture, a templating system, as well as a simple setup auto-generated administration for our database models. It also provides a state-of-the-art Object-Relational Mapper (ORM) with support for our PostgreSQL database, which allows us to write our database tables (such as the index) as Python classes and use the plethora of database functions Django provides.

Since a team of 4 people worked on this project and the final product needed to be publicly deployed, we made the setup for our project as reproducible as possible with the help of Docker and Docker Compose[2]. Our project consists of 4 composed containers (image name in parentheses):

- **web** (*python:3.8.2*): running our Django project using gunicorn[3]
- **static** (*nginx:alpine*): web server routing network traffic to our web application and serving static files (css, javascript, ...)
- **db** (*postgres:13.1*): PostgreSQL database storing our index and other processed data
- **memcached** (*memcached:latest*): a memory object caching system used by Django in order to provide faster loads on repeated requests

Because our project has been *dockerized* from the very beginning, each team member essentially only needed to install Docker and run `docker-compose up -d --build` in order to get up and running. It also means that our deployment to DigitalOcean is nearly one-click while having the exact same configuration as our local development versions. Another benefit is simple extendability: we can add a new container or change one of our containers (e.g., use Redis instead of Memcached), and Docker does the heavy lifting on the next build while keeping consistency among our deployments.

Thanks to using our selected technology stack, our project is easily scalable from relatively low-powered servers to high-performance machines. It is possible to create the database index locally on a powerful computer and then import it to a much slower server which then only needs to deal with running information retrieval queries and serving client requests.

That can, however, be relatively slow on low-powered hardware, and in order to support multiple users at the same time, at least 2 CPU cores must be available, otherwise a single core is quickly overwhelmed. Since memory caching can deliver significant performance improvements (see section 9.2), a multi-core machine with > 4 GB RAM will perform the best. As our current deployment relies on free promo credits from DigitalOcean which only last 60 days, we decided to go with the most expensive droplet we could afford which has 4 vCPUs and 8 GB of RAM. This

[1] https://www.djangoproject.com/

[2] https://www.docker.com/

[3] https://gunicorn.org/

configuration, however, costs $40/month, which would be prohibitively too much if we had to use our own money to run it as a side project. That is why we have initially tested it on the $15/month 2 vCPU + 2 GB RAM droplet, which also provided reasonable performance.

2. Data Collection

We have collected recipes from a variety of popular recipe websites. The recipes are scraped using scrapers provided by the `recipe-scrapers`[4] python package.

For each website, before scraping its recipes we need to find the URL for each of its recipes. We have accomplished this by using each website's sitemap file. For each website we defined using a regular expression which the URLs of the recipes conform to. Therefore, we are able to retrieve all URLs from the website's sitemap and only keep those which correspond to recipes.

Some websites have a single sitemap with URLs for all of their pages, whereas some others have one root sitemap with links to multiple sitemaps. The first case is simple, and we retrieve all recipe URLs in the way described above. For the second case, we use a regular expression to retrieve the URLs for the sitemaps which contain recipe URLs (Such websites usually have a different sitemap for each page type.). Then, for each of the sitemaps, whose URL matches the regular expression we follow the process described above to scrape the recipes.

When the recipes are scraped, the main components of the recipe are separated. These are the *Title*, *Ingredients*, *Instructions*, and optionally the *Author*, *Total Time*, *Tag*, *Ratings* and nutritional information. If a page returns no title, ingredients or instructions when scraped, this suggests that either the page does not contain a recipe or the recipe is not formatted in a standard way, therefore it is discarded.

We also store the canonical URL for each recipe, as well as a link to the provided image which is displayed in search results. Apart from that, some scrapers also include information on a recipe's nutrition, yields, and total time to prepare the recipe. Where these values exist, they are stored as well, since they take up little to no space compared to the main components like instructions. Some of these values are then displayed to the user (e.g., total time) while others are ignored for now, leaving us an option to use them in the future if we deem them useful.

We have collected 380,869 recipes from 24 popular recipe websites. The number of recipes from each source, as well as the proportion of recipes from each source for which a cuisine tag is available are listed in Table 1. The recipes which have tags will be used for training the cuisine classifier (Section 8).

In order to be able to run a variety of different data analysis experiments on the collected data, we first store all scraped raw data in a bzip2 compressed pickle[5] file. Thanks to the compression, our largest raw dataset for eatsmarter.com containing about 79,000 recipes takes up only 17.3 MB, which allows us to share all our datasets in the GitHub repository[6]. All our compressed raw datasets combined take up a little over 100 MB of space, which is insignificant compared to our final index size.

Source	Recipe count	Proportion with cuisine tag
A Couple Cooks	2170	100.0 %
Allrecipes	74652	0.2 %
Amazing Ribs	383	100.0 %
Ambitious Kitchen	925	40.6 %
BBC Food	10177	0.0 %
BBC GoodFood	12168	65.5 %
Bon Appétit	12537	0.0 %
Budget Bytes	1037	0.0 %
Claudia Abril	1058	0.0 %
Cookie and Kate	774	70.8 %
Eat Smarter	79165	25.6 %
Epicurious	13169	37.3 %
Food & Wine	22276	0.0 %
Great British Chefs	5125	0.0 %
Jamie Oliver	602	16.7 %
Minimalist Baker	1121	100.0 %
MyRecipes	64647	0.0 %
NYT Cooking	20784	44.7 %
Taste	14457	24.4 %
Taste of Home	21019	20.2 %
Tasty	5639	59.7 %
Tesco Real Food	5411	99.9 %
The Happy Foodie	2707	0.0 %
The Spruce Eats	8869	0.0 %

Table 1. The sources of the recipes we collected alongside the number of recipes from each source and the proportion of those recipes for which a cuisine tag could be scraped

3. Database Model

We chose to store our index and other recipe information in a PostgreSQL 13.1 relational database within its own Docker container.
There were several reasons for this decision:

Portability The final database takes up about 4820 MB of disk space, which is hardly portable. When exporting it using `pg_dump -Fc` (custom compression) however, the database dump takes up only ~350 MB (7.3% of original), which is shareable through Microsoft Teams.

Scalability PostgreSQL can easily scale up to gigabytes, terabytes, and even petabytes in special cases, hence we had no reason to worry about our index growing large.

[4] https://github.com/hhursev/recipe-scrapers

[5] https://docs.python.org/3/library/pickle.html

[6] GitHub has a 100 MB file size limit: https://docs.github.com/en/github/managing-large-files/what-is-my-disk-quota

Accessibility Django provides an object-relational mapping (ORM) for PostgreSQL by default, hence we can take full advantage of Django's sophisticated database management and functionality without looking for 3rd-party packages.

Efficiency Instead of storing our index in several json or custom-format text files, PostgreSQL took care of table relations (e.g., between tokens and recipes), and we could be confident that the SQL queries Django produced behind the curtains were as efficient as possible.

Our entire database model is displayed as a UML diagram in figure 1. All or the most important fields for each model (table) are displayed as well as the various relations between different models.

3.1. Recipe Information

The most important model in the database is the `Recipe`, which contains all our processed information about each recipe, such as its title, URL, and its original ID at the source where we scraped it from. All other models revolve around the `Recipe` and provide additional information through relations, as well as the index itself.

Since each recipe originates from an online source, we need to know from which source so that we can refer to it, which is what the `Source` model handles. It contains the title and base URL of a recipe source website, as well as a *favicon* which is displayed in the search results next to the source's name.

3.2. Ingredient and Tag Index

Many recipes often contain the same ingredients and are classified into the same cuisines, hence both of these entities need their own models. Both the `Ingredient` and `Tag` models only store their title, as that is usually the only data we are able to scrape, and it is also the most important data when searching.

Originally, we also stored the quantity and unit of an ingredient in each recipe, if that data was available. Unfortunately, correctly parsing this information in every case proved nearly impossible (this is further discussed in section 5) and generally not beneficial, hence we decided to store only the title of an ingredient. Both of these models have a many-to-many relationship with `Recipe` since many recipes contain the same ingredients and tags, and any recipe can have multiple ingredients or tags.

3.3. Token Index

The primary portion of our indexing described in section 6 produces a set of `Token` objects, which are related to each `Recipe` through the intermediate `RecipeToken` model. The `Token` model stores only the title of the token while the actual index is constructed using `RecipeToken`s, which carry additional information. They capture the position of the token within a recipe, as well as what part of that recipe it appears in (`token_type`). This can be *title*, *author*, *instructions*, or *ingredients*, which enables us to assign different weights to tokens from different categories. For instance, a token appearing in a recipe's title is likely more relevant and important than one a part of the author's name.

4. User Interface

We built the User Interface using the Django Template Language[7]. The UI is designed in a way that makes it simple for the user to submit a query and displays the search results in a simple and straightforward format. We ensured that the website is also mobile friendly by using Bootstrap's[8] Grid System.

We made use of the Template Library's template inheritance feature to build separate blocks for different UI components which can then be nested in a parent template to construct the page.

A key component of our UI is the recipe card, an example of which is shown in Figure 2. The search results are displayed as a sequence of recipe cards. Important information about a given recipe are displayed: the title of the recipe, the name of the source website, the logo of the source, its ingredients (after they have been parsed in the way that is described in Section 5), the thumbnail picture of the recipe, the total time needed for the recipe and the average review score in the form of a star rating. If the review score or the total time are not available for a given recipe, they are omitted. If a thumbnail is not available, a placeholder image is used. The ingredients that match the given 'to include' or 'must-have' ingredients are highlighted in green. Clicking on any point on the card redirects the user to the original web page where the recipe is located, which is opened in a new tab, so the user can still browse the retrieved results.

We use a paginator to limit the number of recipes appearing in a single page which improves the browsing experience. The user can use the navigation controls at the bottom of the page to switch between pages. We set the paginator to 10 recipes per page. We also had to make sure that the page retained all the search parameters during the page change, including the search expression, and the ingredients list that need to be included or excluded from the shown recipes.

The user can enter the query parameters through the form elements at the top of the page. The text box at the top is used to enter the text query that will be used for ranked retrieval. In addition, there is an advanced search view which is collapsible and can be accessed through the button next to the search query text box. From there, the user can enter ingredients in three different constraint categories: 'must-have', 'to include' and 'to exclude' ingredients. More details about how these constraints are used are given in

[7] https://docs.djangoproject.com/en/3.1/ref/templates/language/

[8] https://getbootstrap.com/

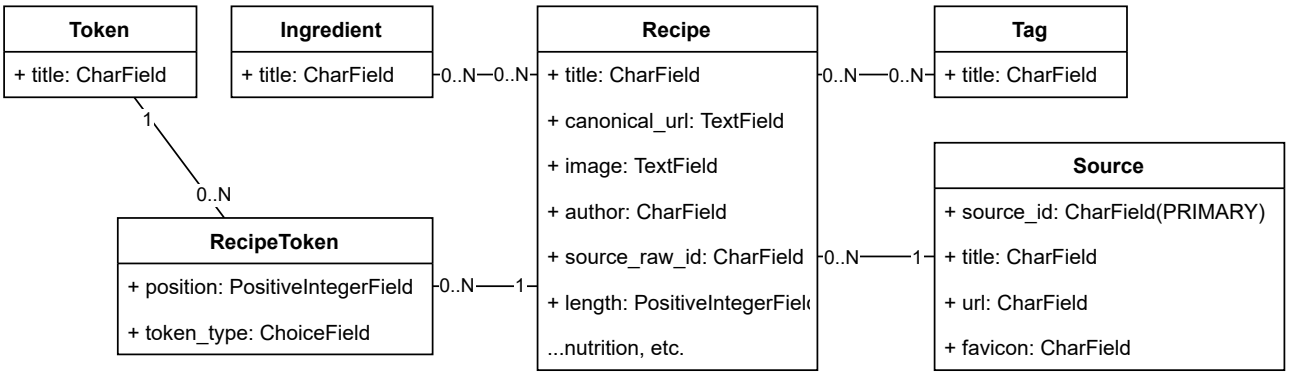

Figure 1. UML diagram of the database table structure, including relations between models.

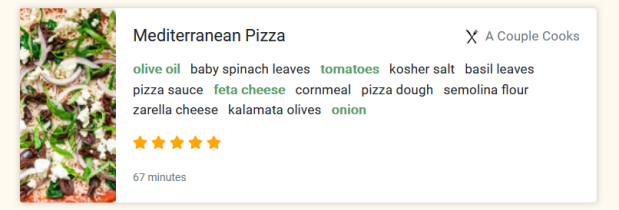

Figure 2. An example of a recipe card

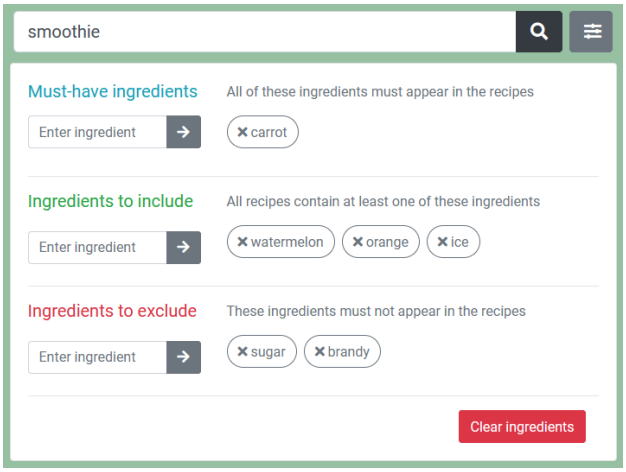

Figure 3. The search interface, including the text box for text queries and the advanced search section for choosing ingredient constraints

Section 7. The ingredient selection fields also provide autocomplete functionality.

Below the search fields, the available options to filter by cuisine are displayed. Clicking on any of the filters displays only the search results which are classified in that particular cuisine. Clicking on the selected filter removes the filter and shows all the results again.

Our website consists of two pages. The home page provides the search components and displays three recipes which are retrieved at random from the database and displayed as suggestions. These recipes must have rating at least 4 out of 5. The search page displays the results after a search request has been submitted. It provides the same search functionality as the home page, displays the results as a sequence of recipe cards and shows the cuisine filtering options.

5. Ingredient Parsing

The names of the ingredients used in the recipes had to be isolated from the quantity required and any comments. The methodology that was followed to achieve this goal is outlined in this section.

5.1. Ingredients phrase tagger

As the task of tagging ingredients was too complex to be implemented from scratch, potential alternatives were considered. The ingredients phrase tagger used by New York Times cooking[9] is a purpose-built tagger that uses a conditional random field (CRF) classifier to determine the different parts of an ingredient (quantity, unit, name, comment, other). Unfortunately, the repository is no longer maintained. A fork[10] maintained by Michael Lynch where some of the bugs of the original library were fixed was used instead. This library is actively maintained as it is used by Zestful[11], an ingredient parsing service. As Zestful is a paid service, we decided to integrate the library to our project instead. For the purposes of this project, we are only interested in the name tag of an ingredient, and any additional information can be discarded.

5.2. Ingredients sampling

For testing and optimisation purposes, the ingredients from the ten smallest datasets scraped were combined, totalling

[9] https://github.com/NYTimes/ingredient-phrase-tagger
[10] https://github.com/mtlynch/ingredient-phrase-tagger
[11] https://zestfuldata.com/

Step	Preprocessing	Ingredient example
1	Remove unicode vulgar fractions	¼ tbsp sugar
2	Decode unicode characters	chicken pâté
3	Remove any brackets and their content	(8-oz) steak [thinly cut]
4	Remove non alphanumeric characters	5% fat yogurt
5	Replace common shorthands ('tsp' , 'tbsp', 'oz') with full words	5 tbsp of sugar
6	Replace 'kg' with 'g'	1Kg self raising flour

Table 2. Preprocessing applied to ingredients before tagging

Step	Postprocessing	Ingredient example
1	Split ingredient after 'ounces'	fluid ounces cranberry juice
2	Split ingredient after 'lb'	450 grams1lb caster sugar
3	Remove leading 'g '	g caster sugar
4	Remove leading 'x'	x tomatoes
5	Remove leading 'ml' (optionally with quantity)	5 140ml milk
6	Remove leading 'cups' (optionally with quantity)	34 cups flour
7	Remove leading numbers	30 carrots
8	Split on 'and' or 'or'	salt and black pepper

Table 3. Postprocessing performed to remove additional information

24,000 ingredients. The tagger was initially tested on this dataset, without any preprocessing applied to it. An ingredient was considered mislabelled when the name tag did not include the name of the ingredient, or if unnecessary details (such as quantity or comments) were included.

5.3. Ingredients preprocessing

The initial performance of the tagger for the sample dataset was promising, but showed room for significant improvement. A variety of different preprocessing combinations was tested. Interestingly, stricter preprocessing, such as such as removal of all quantities and/or units yielded worse results. Therefore, only necessary preprocessing was applied. Priority was given to systematic classifying errors related to the structure of the mislabelled ingredient.

The preprocessing steps, accompanied with an example of an ingredient that benefits from this step, are summarised in Table 2. All entries were converted to lowercase for consistency purposes. All ingredients containing a vulgar Unicode fraction were mislabelled. Initially, decoding fractions to their corresponding number form was attempted, however, the tagger did not deal well with fractions either. As a result, they were completely removed. All other Unicode characters were decoded, as ingredients containing them are common in recipes. Any type of bracket found in an ingredient string most likely contains a quantity, an alternative measuring unit, or a comment. Hence, all brackets and their contents were removed. Ingredients containing full words instead of shorthand yielded much better results so some shorthands were replaced for full words. Finally, 'kg' was converted to 'g' for better accuracy. Since the quantity and unit are discarded post classification, this incorrect conversion did not introduce any drawbacks.

5.4. Ingredients postprocessing

After the preprocessing was applied to the ingredients, the name tag of the ingredient was extracted using the classifier and everything else was discarded. Examining the results verified that the preprocessing was effective, however, unnecessary information was retained for ingredients. Fortunately, for most cases, the extra information could be easily removed with some additional processing.

The steps that were followed to remove additional information, accompanied with an example of an ingredient that benefits from this step, are summarised in table 3. Some ingredients containing alternative measurements units, often separated by a slash or whitespace, were mislabelled. In these cases, the ordering of units was observed to be constant (e.g. grams followed by lbs, ml followed by fl ounces), which allowed for the quantity and unit to be discarded consistently. Ingredients containing unusual amounts of units were mislabelled. The quantity and unit was then discarded using regular expressions. Care was taken to ensure that the aforementioned regular expressions were flexible, by allowing optional quantity and whitespace. Subsequently, ingredients containing the words 'and or 'or' were split into two distinct ingredients. Finally, any unnecessary whitespace was removed.

5.5. Removal of uncommon ingredients

After postprocessing, the database contained over 150 thousand unique ingredients, appearing a total of about 4 million times. Thresholding was used to keep only the most common ingredients. The number of remaining unique ingredients and the percentage of total ingredients remaining for a varying threshold are outlined in figure 4. A threshold of 50 was used to remove any gibberish that survived the previous steps, perhaps by appearing in an identical or similar fashion across multiple recipes. Instead of discarding uncommon ingredients completely, they were replaced with the most common ingredient that they contain. (e.g. maple sugar was replaced by sugar). This step assigned 333994 out of 395603 (84%) ingredient occurrences that did not meet the threshold to an appropriate ingredient that did. To summarise, just over 60 thousand ingredient occurrences were discarded while reducing the amount of unique ingredients by 98%.

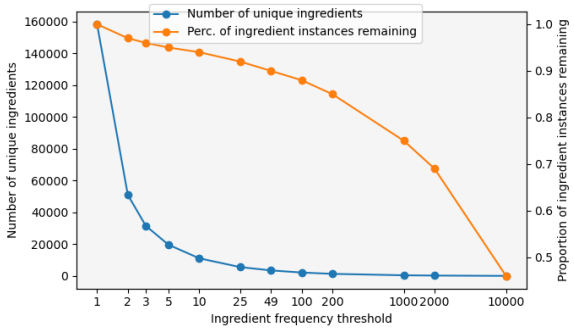

Figure 4. Effect of different thresholds on unique and total occurrences of ingredients

5.6. Assimilation of common ingredients

Examining the list of ingredients that remained, it was observed that some ingredients were found using two different spellings. This could be mainly attributed to plurals (e.g. eggs - egg) and cases where two or one word were used for an ingredient (e.g. selfraising vs self raising flour). Initially, ingredients were assimilated to the most common ingredient that is one edit distance away. However, this method was far from perfect as it sometimes matched unrelated ingredients (e.g. beer - beef or ice - rice). To tackle the issue, ingredients ending in 's' were replaced with their singular counterpart (without 's'). This method matches 'peppers' with 'pepper', which might be confused for seasoning. In all other cases, the desired result is achieved. Examining all the matched single edits between common ingredients revealed that 242 out of 358 (68%) matches were taken care of by this method.

6. Indexing

The indexing module is responsible for the construction of the inverted index which enables fast retrieval of recipes where a particular term is present.

6.1. Text Preprocessing

First, since the text is scraped from HTML documents, all named and numeric character references such as '`&`' are converted to the corresponding Unicode characters. Then, using Unidecode[12], non-ASCII characters are converted to the closest ASCII character (e.g. 'â' → 'a', 'é' → e). This is done since diacritics are not used consistently when words that have them are written in English text. For example, 'pâté' is sometimes written as 'pate'.

Then, the text is converted to lowercase and it is tokenized by splitting on non-alphabetic characters, excluding the apostrophe. Leading or trailing apostrophes, which appear when apostrophes are used as quotation marks, are removed. Apostrophes followed by an 's' are also removed if they appear at the end of a token to avoid having, for example, "butter" and "butter's" as separate tokens. Tokens which are stopwords are removed, using NLTK's list of English stopwords. Finally, the tokens are lemmatized using NLTK's WordNet Lemmatizer.

[12] https://pypi.org/project/Unidecode/

6.2. Constructing the Index

The recipes are loaded one-by-one from the compressed pickle files that were stored during scraping as described in section 2. A `Recipe` object is created with its fields populated and committed to the database. For each dataset, a separate `Source` object is stored in the database with information relating to the corresponding recipe website (Favicon URL, website name etc.).

Each of the four text components of the recipe are retrieved. These are: title, author, ingredients and author. Each component is preprocessed separately following the procedure described in 6.1. For each token that appears in the recipe, a `RecipeToken` object is added to the database which besides the Token and Recipe ID includes the position in the document where the token appears, as well as a field called 'Token type'. The token type is a field designating in which text component of the recipe the token appeared. It takes values 1, 2, 3 and 4 for the Title, Author, Instructions and Ingredients respectively.

In addition, the ingredients are preprocessed following the method described in Section 5 and a `RecipeIngredient` object is stored to the database for each ingredient occurrence. If the ingredient has not been encountered before, a new `Ingredient` object is also created.

6.3. Cleaning the Index

As with most data, Zipf's law applies to our dataset as well since the vast majority of token appearances are accounted for by a tiny portion of tokens. This applies for our ingredients and tags as well, and since the least occurring objects are the least useful for searching, we remove all objects from the database which occur less than a certain threshold after full indexing. The thresholds, percentage of objects **deleted**, and percentage of total object occurrences (e.g., relations between Token and Recipe) remaining after deletion are displayed in table 4.

Model	Occurrence threshold	Objects deleted	Occurrences remaining
Tags	100	81.7%	96.9%
Ingredients	50	98.7%	96.4%
Tokens	10	81.2%	99.7%

Table 4. Occurrence thresholds for Tags, Ingredients, and Tokens, and their effect on the index after deletion.

Although deleting recipe tokens could potentially result in some extremely rare recipes being completely inaccessible (if all of their token relations were deleted), the tokens

which appear less than 10 times across 380,000 recipes are incredibly unlikely to be used in a search. The most frequent of these include, for instance: 'degrade', 'cadillac', 'purist', or 'chocoholic'. As for Tags and Ingredients, deleting objects merely gives the user fewer options for filtering, so we could set the threshold much higher. Even with thresholds of 50 occurrences for ingredients and 100 for tags, however, over 96% of all total occurrences remain for both models.

7. Retrieval

The user can give the following search parameters:

- **Search query:** A string which is used to rank the relevant results.
- **Must-have ingredients:** Ingredients which must appear in the ingredients of the retrieved recipes.
- **Ingredients to include:** Ingredients which can appear in the ingredients of the retrieved recipes. If no must-have ingredients are given, at least one of those ingredients must appear.
- **Ingredients to exclude:** Ingredients that must not appear in the ingredients of the retrieved recipes.

First, ranked search is conducted using the search query. Then, if ingredients are given, the recipes satisfying the ingredient constraints are found and results from the ranked search are filtered to only include the recipes satisfying those constraints and are returned. If only ingredient constraints are given without a search query the recipes satisfying the constraints are returned without a particular ordering.

7.1. Spelling Correction

Before the user's query is executed, it is passed to the spell checking module which detects tokens that are not in our dictionary and attempts to correct them. We use the set of tokens that appear across all the recipes we have collected as the dictionary.

First, we find the tokens that are not in the dictionary. For each of those tokens all possible candidate tokens are found. These tokens are the tokens which are at most two edit distances away that are included the dictionary. We use the Damerau–Levenshtein distance metric which allows the following types of edits:

- **Deletion:** A character being removed
- **Transposition:** Two adjacent characters being swapped
- **Insertion:** One additional character being inserted
- **Substitution:** One character being replaced with a different character

Initially only the candidate tokens that are one edit distance away are found. If there are no such candidate tokens, the candidate tokens that are two edit distances away are retrieved. Then, the candidate token that appears the most in the recipes we have indexed is picked as the corrected token and returned. If no correct spelling can be found, the token is removed from the query. A potential issue with this approach is that if a misspelling of a word is present in our database, this could mean that the token is not corrected. This is mitigated to a great extent by the fact that rare tokens are removed from the database as mentioned in Section 6.3.

7.2. Text Ranked Search

First, the search query is preprocessed in the same way described in Section 6.1 and the spelling of the tokens is corrected as described in Section 7.1. Then, we retrieve the documents that are most relevant to the query.

The recipes/documents are ranked using the Okapi BM25 ranking function, with an additional weight depending on whether the token appears in the title of the recipe. Each document d is assigned a score $S(d, T)$ as for a query with tokens T as follows:

$$S(d,T) = \sum_{t \in T} c_{t,d} \frac{tf_{t,d}}{k\frac{L_d}{\overline{L}} + tf_{t,d} + 0.5} \log_{10}\left(\frac{N - df_t + 0.5}{df_t + 0.5}\right)$$

where k is a parameter set to $k = 1.5$, $tf_{t,d}$ is the term frequency of the term t in document d, $\overline{L}$ is the mean length of all documents in our database, L_d is the length of document d, N is the total number of documents and df_t is the document frequency of the term t. The weight $c_{t,d}$ is 5 if t appears in the title of d and 1 if it does not.

At most 100 recipes with the highest scores are returned which are sorted based on their score.

BM25 was chosen since it bounds the effect of very high values of term frequency (the limit of the term frequency component as term frequency approaches infinity is 1) and it also takes into account the length of a document, avoiding giving higher score to longer documents which are likely to have more occurrences a token, but are not necessarily more relevant compared to a shorter document with fewer occurrences of a token. In addition, we added the weight which is greater when a token appears in the title of a recipe since it is very likely that a recipe's title includes the most important tokens for the document.

7.3. Ingredient Boolean Search

The Ingredient Boolean Search functionality is responsible for retrieving the set of recipes satisfying the given ingredient constraints. For each ingredient the set of recipe IDs where the ingredient string is a substring of any of the ingredients of is retrieved. For example, in the recipes with the ingredient "flour", the recipes which have the ingredient "plain flour" are also included.

To find the recipes satisfying the constraints we follow the

steps:

1. If no 'Must-have' or 'To include' ingredients are given, the initial recipe set is the set of all Recipes. Otherwise, the initial recipe set is the empty set.
2. Iterating through the 'Must-have' ingredients, at each iteration the recipe set is the conjunction of the current recipe set and the set of recipes where the 'Must-have' ingredient appears.
3. Iterating through the 'To include' ingredients, at each iteration the recipe set is the disjunction of the current recipe set and the set of recipes where the 'To include' ingredient appears.
4. Iterating through the 'To exclude' ingredients, at each iteration the recipe set is the difference of the current recipe set minus the set of recipes where the 'To exclude' ingredient appears.

The resulting recipe set is returned.

8. Cuisine Classification

An important functionality of our system is to filter search results based on their cuisine category. However, not every scraped recipe had a cuisine tag available. To solve this issue, we built a cuisine classifier which uses the ingredients of a recipe to predict its cuisine.

This idea is supported by a study (Kalajdziski et al., 2018), where they built a Support Vector Machine classifier that classifies recipes to cuisines using their ingredients, with an accuracy of 80.9% on a Kaggle dataset[13]. We attempted to recreate this result on the same dataset and experimented with different machine learning algorithms, including Logistic Regression, Nearest Neighbours, Linear Discrimant Analysis Classifier and Linear Support Vector Classifier. We found that using Logistic Regression provides the highest accuracy out of all of them, 78.3%, which looked sufficient and close enough to the performance of the classifier in the paper.

We retrieved all the recipes from our database, alongside their corresponding list of ingredients and tag (if available). After gathering the data, labelled and unlabelled recipes are separated, and labelled recipes are used as training data. In a similar way to the paper, the data is transformed to have all possible ingredients as an attribute, with the value indicating whether a recipe contains an ingredient or not (1 if it contains the ingredient, 0 if it does not).

We also decided to filter the data, and ignore cuisines and ingredients that appear less than 100 times. This will improve the classifier as it will work with only the most meaningful attributes, and the reduction number of classes (especially the elimination of very imbalanced ones) will also improve the classifier's performance.

[13] https://www.kaggle.com/c/whats-cooking/data

Threshold	Proportion of labelled recipes (%)	Accuracy of given labels
0.45	56.3%	72.93%
0.50	47.3%	75.73%
0.51	45.4%	76.16%
0.55	38.6%	77.74%
0.60	31.0%	83.06%
0.65	23.9%	83.78%
0.70	17.33%	86.89%

Table 5. Effect of different thresholds on cuisine classification with the proportion of recipes that were labelled (out of a total of 224,422 recipes) and the proportion of those labelled recipes that were labelled correctly (accuracy of given labels) - only considering cases where the probability was high enough (above the threshold) to give a label, not considering low threshold cases when more than one label is returned.

After training the Logistic Regression classifier on the labelled training data, we can predict the cuisine for the unlabelled recipes. First, we transform them to the same format as the training data and assign to them the label for which the model gives the highest probability. For our system we wanted to prioritise precision over recall, which means that we only want to return a label, if the label probability the model gives back is sufficiently high. Therefore, we are using a threshold the label must reach in order to assign that label. The higher this threshold is, the more confident we can be that the assigned labels are correct. However if the threshold is too high, a large number of recipes will not receive any labels. To begin with, we set a threshold of 0.51 as this probability would ensure that there is a tag with a probability higher than all other tags combined (if a label is assigned).

Table 5 lists the proportion of recipes that were labelled and the proportion of the assigned labels that were correct for different thresholds. We used 90% of labelled data for training and 10% for testing the models. The higher thresholds resulted in a more accurate classification naturally, however as expected it resulted in lower amount of data receiving a label.

Overall we found that a higher number of recipes get correctly classified if we choose a lower threshold like 0.51. For the reason that our collected data has far more unlabelled recipes than labelled ones, we would benefit from using a 0.51 threshold, which would allow us to have a cuisine tag for almost half of the recipes in our database. This would mean that filtering based on tags would give the user a sufficiently high number of results to be a meaningful functionality. To summarise, our final model for cuisine classification is a multi-class Logistic Regression model that assigns a cuisine tag to a recipe only if the probability of the label is at least 0.51.

#	Query	Justifcation	Pr@1	Pr@10	Pr@20
1	roast chicken	Simple common query	1	0.90	0.90
2	vanilla ice cream	Two words often used as a single ingredient	1	1.00	1.00
3	carbonara	Specific dish	1	1.00	1.00
4	spaghetti bolognese	Specific dish and sauce	1	1.00	1.00
5	cardamom butter chicken	Combination of common and uncommon words	0	0.20	0.20
6	christmas panettone	Italian and seasonal	1	0.70	0.60
7	chicken roasted overnight on rosemary and thyme	Long specific query	0	0.40	0.25
8	pork joint sprinkled with smoked paprika	Long specific query	1	0.50	0.40
9	crème brûlée	Words with diacritics	1	1.00	1.00
10	chicken teryiaki	Common misspelling	1	1.00	1.00
11	piza	Missing letter	1	1.00	1.00
12	dakami	Two misspellings (salami)	1	1.00	1.00
13	Gordon Ramsay [include = egg, cheese]	Author name	1	0.80	0.75
14	[must-have = butter, egg]	Boolean without query	1	0.90	0.90
15	soup [include = potato, carrot]	Query with include	1	1.00	1.00
16	gammon [exclude = pineapple]	Query with exclude	1	1.00	0.95
17	pasta [include = chicken, tomato] [exclude = cheese]	Query with include and exclude	1	0.80	0.80
18	spaghetti carbonara [must-have = pancetta]	Query with must-have	1	1.00	1.00
19	[must-have = cucumber][include = tomato][exclude = cauliflower]	Combination of Boolean	1	1.00	1.00
20	ice cream sorbet [must-have=lemon, include=sugar, exclude=strawberry]	Query with complex Boolean constraints	1	0.60	0.55

Table 6. Precision of 20 queries evaluating a wide range of implemented functionality, at 3 cutoff values

9. Evaluation

9.1. Relevance of results

The most common metrics used in literature for evaluation of search engines are precision and recall (Sheela & Jayakumar, 2019; Atsaros et al., 2008). Precision is the fraction of retrieved recipes that are relevant, recall is the fraction of relevant recipes that are retrieved. Precision @k, (i.e. Having a cut-off on the ranked list at rank k before calculating precision) is perhaps the most appropriate metric for search engine evaluation, as users usually only check the first k results. Analysis of five million google search results[14] showed that the 10th search result displayed is ten times less likely to be clicked compared to the 1st. Interestingly, less than 1% of the users were interested in a result ranked 11-20. Other evaluation metrics exist but require a precomputed list of relevant results.

For the evaluation of the results of the search engine, precision for 20 different queries at three different cutoffs was calculated. This required manual assessment of the first 20 results returned for each query. We designed our test queries such that they test a variety of potentially problematic situations. The selected queries along with a short justification of the selection and the precision of the first 20 results are summarised in Table 6.

[14] https://backlinko.com/google-ctr-stats

The evaluation results are very promising. The system returns excellent results for simple queries of common ingredients (queries 1-4), parses diacritics correctly (query 9), corrects simple and double mistakes in query spelling (queries 10-12) and is flexible with Boolean queries (queries 14-19).

Most of the false positives occurred when the search query was more specific. For example, query 1 returned results containing "broccoli in roast chicken drippings", which includes the query and is not completely irrelevant. The number of irrelevant results increases as the query becomes more specific. Queries 5, 7 and 8 returned some results that matched the majority of the query, but not all of it. With the current implementation, the results of these queries could be easily improved by adding some of the ingredients in the "must have" advanced search filtering instead of the search query. Searching for a specific author also returns some false positives, as shown in query 13. These can be justified by noting that authors that share the same name or surname exist and that the author of interest is sometimes mentioned in recipes by other authors. False positives from Boolean queries are also possible, as sometimes irrelevant ingredients can contain the ingredient of interest as a substring (e.g. butter - buttermilk).

#	Query	Run 1 (s)	Average of runs 2-5 (s)
1	**pizza**	3.97	0.37
2	pineapple **pizza**	1.94	0.60
3	sweet & sour **chicken**	9.02	3.54
4	**chicken** teriyaki	1.76	1.91
5	**pizza**	0.34	0.38

Table 7. Five search queries run in order, each run 5 times, and their times recorded. Common words are in bold, and the columns show the time for the first run and an average of the next four runs.

9.2. Cache Evaluation

As mentioned in section 1, we are using memcached for automatically memory caching just about anything Django can serialize. One type of these "cacheables" are database queries which are the basis for our information retrieval. When a user searches, for instance, for "pizza", the results of the database query for "pizza" are cached for later use and do not need to be computed again if they search for "pizza margherita". Moreover, this cache is independent of the user, so if any user has searched for "pizza" recently, next time someone else searches for a query containing it, it will load significantly faster.

Table 7 shows this effect on a series of queries run in sequence, each of which is run 5 times in a row (i.e. the sequence is "11111 22222 3..."). The recorded times are from our search function itself rather than an HTTP request, as these carry extra overhead which can blur the results.

The first time "pizza" is search for it takes about 4.0 seconds, while the next four runs average at only 0.37s. The next query contains the previous one, hence "pizza" can be retrieved from the cache and only "pineapple" needs to be searched. Query #3 contains more tokens, hence it takes about 9.0 seconds to finish on the first run, but the subsequent runs are still nearly 3 times faster. Query #4 is the only one that does not see a significant improvement, which is likely because there are 13.8x more "chicken" occurrences than "pizza", hence it takes noticeable longer even just to retrieve those results from the cache. More importantly, when we search for "pizza" again at the end, all runs including the first one take only around 0.34 seconds, meaning those results are still cached.

These 5 queries and 25 search runs in total take up only a relatively small portion of the RAM (low 100s of MB in total) but result in significant performance gains. This suggests that increasing the amount of RAM on our deployed server would lead to better improvement in performance than increasing the core count. Of course, this assumes that users will tend to search for similar queries; however, this *is* a likely scenario due to the zipfian nature of our dataset, hence improvements would be expected in real life. Since disk (database) space is cheaper than RAM, another possible solution would be to cache the common queries in the database, either by setting a threshold for minimum user searches before a query is cached or simply pre-computing the search results for N most frequently occurring tokens.

10. Conclusion

Summary of Work Done

We have scraped about 380,000 recipes from 24 popular recipe websites, pre-processed and parsed them, and then indexed this data into a PostgreSQL database. Ingredients were processed the most since their format differed greatly across all recipes and both singular and plural terms (e.g., 'egg' vs 'eggs') occurred (section 5). Ranked search using the Okapi BM25 ranking function was implemented, as well as Boolean search for ingredient constraints. A Logistic Regression cuisine classifier was trained and used to label around 50% of our unlabeled recipes with the cuisine they belonged to based on their ingredients. We have also evaluated our search engine based on how relevant search results were, as well as how memory caching is improving our performance and could do so even better in the future.

Future Work

Apart from obvious improvements such as better performance and speed, we would like to make the application more user-centered and personalisable. The first step would be to allow users to create an account so they can collect their favourite recipes and so that the application can store past searches and learn what the user likes for the purpose of recommendations.

In addition, it would useful to have certain ingredients classified as relevant or prohibited for categories like "vegan", "dairy-free", or "ketogenic", in order to make it easier for users to search for recipes if they have dietary restrictions. Especially for categories like "ketogenic", where carbohydrates are severely restricted, using the recipe nutrition would be useful, and where recipe nutrition is not available, we could approximate it using ingredients and their quantities thanks to the USDA database[15]. Users could then specify their dietary preferences on their profile and searches would automatically include these preferences.

Having a large number of users would also provide us with better relevance metrics for results for specific queries, which could benefit all users, even ones without an account. These improvements have to be classified as future work, since they all require a large user-base as well as a significant amount of time and labour, for instance to classify ingredients into their respective categories. They would, however, all bring a better experience for the user and an overall superior application, if the scope of this project allowed us to implement all of them.

[15]USDA FoodData Central: https://fdc.nal.usda.gov/

11. Individual Contributions

Nikolas Pilavakis (s1623062)

My work was mainly concerned with ingredient parsing, as discussed in section 5. This includes experimentation with the ingredient tagger and analysis of the ingredients that were present in the dataset. I also experimented with different preprocessing and postprocessing techniques and ways to assimilate ingredients to reduce the number of unique ingredients that are present in the database.

I also conducted a major part of the evaluation of the results, including research of potential evaluation methods, selection of queries and summarising of results gathered.

Petr Manas (s1652610)

I have primarily worked on data collection, the database model and some of the raw data preprocessing along with Nikolas, and the initial project setup along with Docker and the final deployment.

Since I have worked on several Django projects in the past and present, some of which were adapted to use Docker, I created the initial setup and configuration for our project and guided the rest of the team on how to install and run the Docker containers.

Within data collection, I have written the bulk of our generic recipe scraper classes and adapted Dimitris' implementation for the sitemap scraper into a generic form. I have then implemented several of the scrapers for our specific source websites and collected data using them along with Dimitris.

Then, I have created the database models for our raw data to be indexed to, while ensuring consistency using Django's `Model.clean()` method and several preprocessing utility functions. Some of these were written by Dimitris while others by myself, often with Nikolas' help and testing.

When our application was nearly finished, I created a Droplet (server instance) using Dimitris' DigitalOcean account (in order to be able to use the free promo credits) and deployed our project. Throughout development, I've run several of our database modifications and shared the data dumps with the rest of our team. I have used the final of these dumps to initialize the deployed database. Thanks to the initial work I had done on configuring Docker for our project, deployment was quite simple and only took a few minutes.

Dimitris Christodoulou (s1738739)

My areas of work were in data collection, building the User Interface, indexing and retrieval.

Initially, I worked on data collection. I built upon a generic scraper built primarily by Peter to create web scrapers for several different websites, as our scraper needed to be adapted for each website.

For the User Interface, I improved upon an early version built by Bence. I improved the appearance of the interface and restructured the code to make use of the Django Template Library's template inheritance feature. I designed the recipe cards to have the look and information they do in the final version, and built the current advanced search interface.

Then, I built the indexing module. It loads the scraped data, preprocesses them, stores the recipe information and the source information in the database and constructs the inverted index.

Finally, I implemented the recipe retrieval module which has two main parts. The first one is Boolean Search based on the given ingredient constraints by the user. The Second is ranked retrieval based on the text query. I experimented with several ranking functions and settled on the Okapi BM25 ranking function. In addition, I built the spelling correction functionality by comparing the search terms with tokens in our database using the Damerau-Levenshtein distance.

Bence Mogyoródi (s1655560)

I mainly worked on the Cuisine classifier algorithm and the user interface.

The goal of the cuisine classifier was to improve efficiency of the search engine by labelling recipes with cuisines they most likely belong to, based on the list of ingredients needed to prepare them. Based on the study (Kalajdziski et al., 2018) and Kaggle database mentioned above, I built a classifier which was able to classify recipes with 78% accuracy (based on cross-validation test). Then I build a compact class called Cuisine classifier, which takes as input the recipe list with ID, ingredients, and cuisine labels. The classifier during initialization takes care of the preprocessing, fitting a logistic regression model on the training data, and initializing a Label Encoder, so the labels are returned in a readable form when exported to the database. The classifier then labels the unlabeled recipes. During the development of the algorithm I experimented with different classifiers, filtering conditions, and thresholds for prediction probability.

My other area of work was creating the user interface that allows the user to communicate with the website. My work on the user interface included pagination of search result, which divides the returned result into 10 per page. I also created the main view, along with the partial view (recipe card, which details how the retrieved recipe is supposed to show up on screen). I've also worked on creating the form that sends in the search terms through a get request, with focus on retrieving the standard search term, and the selection of ingredients that are supposed to be included or excluded. I also added the functionality to the include and exclude ingredients field to appear as a dropdown field with all possible ingredients, with auto-complete.

References

Atsaros, Georgios, Spinellis, D., and Louridas, P. Site-specific versus general purpose web search engines: A comparative evaluation. *2008 Panhellenic Conference on Informatics*, pp. 44–48, 2008.

Kalajdziski, S., Radevski, G., Ivanoska, I., Trivodaliev, K., and Stojkoska, B. R. Cuisine classification using recipe's ingredients. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 1074–1079, 2018. doi: 10.23919/MIPRO.2018.8400196.

Sheela, A. C. S. and Jayakumar, C. Comparative study of syntactic search engine and semantic search engine: A survey. *2019 Fifth International Conference on Science Technology Engineering and Mathematics (ICONSTEM)*, 1:1–4, 2019.