The Clown Shoes Collective
Balatero, Brajkovic, Chu, Otero

CSE 454: Advanced Internet Systems
Weld

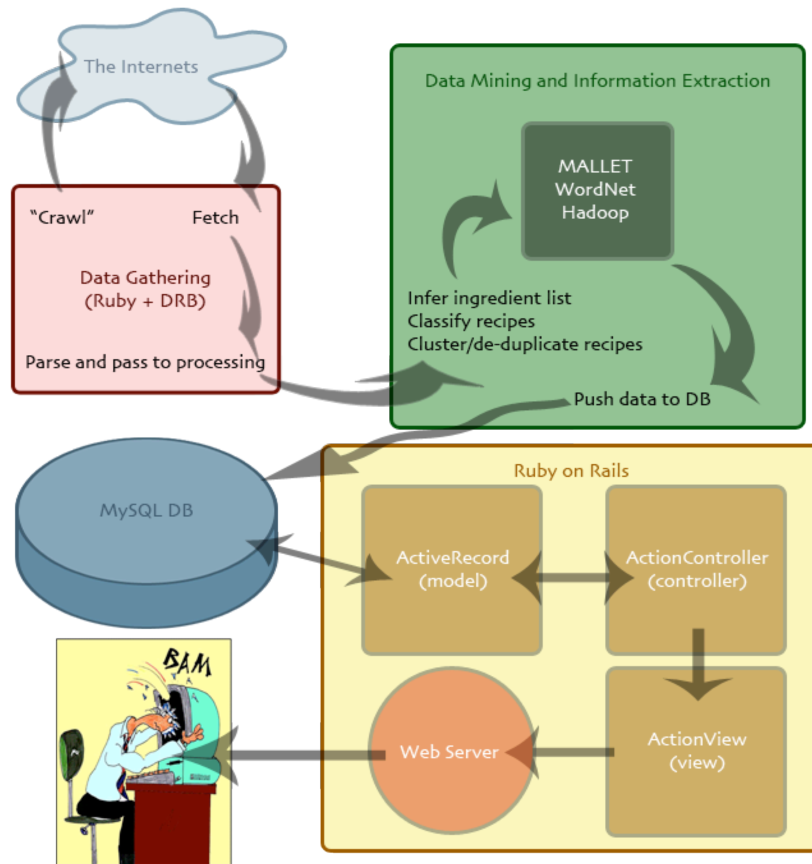
SpaceWalr.us

Intuitive Culinary Help

When it comes to cooks, there are two types. There are those that cook out of passion for food and creativity. Chances are, however, that you are not one of these. Most find themselves in the kitchen in order to satisfy the unfortunate human need to ingest food on a regular basis. As a result, they tend to be inexperienced, and able to prepare very few things. They tend to eat the same limited set of meals without variation, and even if they wanted to branch out, they wouldn't know where to begin. With nothing more than large recipe books and obtuse online recipe databases to browse, how can a busy fledgling cook find a recipe that meets dietary needs and personal tastes, and to be blunt, doesn't suck?

Our goal was precisely to solve this problem: an easy-to-use web application as gateway to thousands of recipes. In order to provide users with answers to their common questions, we would need to allow users to browse these recipes in non-standard, slightly more semantic ways. For instance, a user should be able to search for recipes that included chicken, onion, and garlic. Similarly, users who dislike certain ingredients should be able to quickly exclude them from all searches. This would require more than keyword recognition, as it is surprisingly common for the text of a recipe to include names of ingredients that aren't actually called for. Like most sites in the Web 2.0 world, users should be able to rate recipes that they have tried, and use other users' ratings as a guide when trying new recipes. Most importantly, we wanted to be able to provide recommendations to a user based on their rating history.

We quickly realized that this would require far more knowledge of what went into a given recipe than was available from any source. It would be easy to get our hands on large quantities of recipes in plain text form. We could simply crawl existing recipe sites for all of their recipes. To acquire structured data about the ingredients each recipe contained, cook times and prep times, yield, and so on would require some work. Some fields would be easy to extract, as they'd often be clearly marked by standard blocks of HTML, but more complex fields like ingredients couldn't be simply parsed out. For these we'd need some bigger guns: natural language processing and classification. Finally, in order to recommend recipes based on user preferences, we'd need a user rating data set and some way to compare users' preferences to each other in order to glean recommendation ideas. Theoretically, we could crawl the same sites for their non-anonymous user ratings and comments to create a test data set. Then using some form of clustering, we could group users by similarity of preference, and recommend recipes based on recipes liked by others in a particular user's cluster. This lead us to envision a system architecture like that depicted in the following figure.



Crawling and fetching recipes was fairly straightforward. We used a Distributed Ruby (DRB) framework to parallelize the downloading of hundreds of thousands of recipes. Because each site would need slightly different handling, and a different approach when parsed, we abstracted common logic into a crawling framework, implementing specialized features only when necessary. Most of the specialized code was written to handle the widely varying HTML layouts of each site. Once downloaded, each recipe was assigned a unique MD5 hash (based on URL) and then parsed into a standard plain-text format (using delimiters between fields) and stored on a single line of a recipe data file, along with many other recipes. This format was ideal for use in our scripted Hadoop jobs, which, one after the other, would handle the remainder of the data processing pipeline.

Information extraction would aid us in attempted to learn something about all the ingredients in every recipe we collected. If we wanted our users to be able to search and filter by ingredients on our web front-end, we'd need concrete information about each ingredient used in a particular recipe. Because of human inconsistency in authoring recipes, and even the huge variation in possible expressions of a single ingredient requirement, this data could not be satisfactorily parsed by simple means (e.g. text processing or regular expressions).

To make further discussion more transparent, let us define a few terms. All recipes call for certain ingredients. In a recipe, these requirements are often given one line each. We call each

of these an *ingredient line item*. For example, "1 cup flour" and "3 tablespoons of light cream cheese" are both ingredient line items. When we say *ingredient line* or *line item* we are also referring to ingredient line items. When we say *ingredient*, we refer only to the ingredient called for (i.e. “flour” and “cream cheese” in the previous examples), not the whole line item.

To provide users with all the features we wanted, we’d need to know for each line item:

1) how much of the ingredient was called for, and subsequently what unit was being used,
2) what the ingredient was, and finally
3) what non-ingredient text would help us differentiate between instances of the same ingredient (i.e. “light whipping cream” and “heavy whipping cream” are fundamentally the same ingredient, but must be differentiated).

To abstract this, we chose to break down each ingredient line into 5 parts: quantity, unit type, “prefix,” ingredient, and “suffix.”

1 1/2 cups of fresh green beans, neatly chopped

quantity: 1.5
unit: Volume.CUP
prefix: “of fresh”
ingredient: “green bean”
suffix: “, neatly chopped”

Splitting up each ingredient line was done in two parts: quantity and unit extraction, and ingredient extraction. By first removing quantity and unit information, we would make the latter task somewhat easier. Extracting both quantity and unit turned out to be pretty easy. We did regular expression matching for extracting quantities from the ingredient line. The regular expression was able to handle integers, as well as identify fractional numbers, which are very common in recipes (e.g. “3/4,” or “1 1/2”). We could then convert fractions to decimal equivalents (e.g. 0.75 and 1.5, respectively). Once we matched a number, we would then check the next tokens for a known unit from a hash table of units that were predetermined by us (after looking through thousands of recipes for common types). If we didn't match a quantity or a unit, we just assumed that these values were not given in the string. Consider the ingredient line "sugar to taste," which conveys neither a standard unit or a fixed quantity. We did, however, also track non-standard units often used in cooking to help us better understand the recipes. To make this somewhat academic, we looked quickly at an alphabetical listing of ingredient lines sans quantity to find common non-standard “units” (e.g. “pinch,”, “bunch,” “can,” etc.). While this was somewhat unnecessary, we did it so that such a “unit” wouldn’t be lumped in with the words used to classify ingredients (those in the prefix and suffix). We had original hoped for at least 70% correctness in our quantity/unit extraction, yet we observed 100% correctness in our evaluation.

Once we extracted quantity and unit from an ingredient line, we attempted to extract the ingredient from the remainder of the string. Any remaining words before and after the ingredient would be put in the prefix and suffix, respectively. However, this turns out to be a fairly non-trivial problem to solve. Not only can ingredients have varying amounts of words describing

them, they can also be put in different places in a sentence. In addition, it is very difficult to know which of the words modifying the final noun portion of the ingredient are intrinsically part of the ingredient, or are additional modifiers. For instance, how do you know that "teriyaki" in "teriyaki sauce" is part of the ingredient, but the "chopped" in "chopped carrots" is not?

We initially tried a naive approach to ingredient extraction: given the remaining sentence, choose the last token before a comma or the end of the line as the ingredient, anything before it as the prefix, and anything after as the suffix.

Given: "2 cups of cream cheese, light"

1. Extract quantity and units:

"of cream cheese, light"

2. Remove from first comma onwards:

"of cream cheese"

3. Take the last token as the ingredient:

of cream		cheese		, light
prefix	ingredient	suffix

While it proves to be 100% accurate for one-token ingredients, the is a substantial percentage of ingredient lines with two-token ingredients (and even a tiny percentage with three). This lowers the accuracy of the overall extraction to about 60%, where we define "accurate" extraction such that a cook would accurately be able to recreate the original recipe with our version of the ingredient. Recall, of course, is 100%, as we don't have the flexibility to eliminate ingredients which we can't accurately extract. The problem with this approach is that it doesn't capture crucial information about the ingredient. In the above case, simply classifying the ingredient as "cheese" would not be specific enough, as types of cheeses are generally not at all interchangeable. In this case, it is important to know the actual ingredient in question is, in fact, "cream cheese." Sometimes, as in the case of "sauce," the final token is so uninformative that even a cook wouldn't be able to make a good guess as to what previous tokens should be.

To combat this issue, we started looking at part of speech taggers. We first tried OpenNLP, but later switched over to a Java API for the WordNet database (http://wordnet.princeton.edu). We were able to use a part-of-speech heuristic to extract multi-word ingredients from line items. We search the string for the right-most noun as the most likely candidate to be an ingredient. Then, we check the words to the left of the noun to see if they could be possible modifiers to the noun. This method works much better for extracting full ingredient phrases. When employing the part-of-speech technique, we are able to extract strings such as "cream cheese" for the ingredient, therefore getting the most meaning out of it. In very rare cases this fails and includes too many tokens, but nonetheless we were able to achieve a 95.2% accuracy with this improved method. Our notion of "accuracy" involves some subjectivity, though we could do little else than randomly sample about one thousand of the resulting extracted ingredients and evaluate them.

Our next task was to put recipes into a uniform set of categories. A category hierarchy would make browsing easier for users, helping them to more quickly identify recipes they were interested in. However, the bulk of our data set was unlabeled, categorically speaking. What we managed to do was find a website, Epicurious.com, that included the category of each of its recipes in a canonical format in each recipe's URL. When fetching these recipes, we simply used regular expressions to extract category labels, and then built a hierarchically organized directory structure and fed the labeled set to a pair of classifiers.

Our first attempt at classification utilized MALLET's Naive-Bayes Classifier trained on the plain HTML-stripped text of the labeled recipes. This was somewhat successful, but not nearly accurate enough for us to use in a user-facing application. Ultimately, the advantage of using Naive-Bayes was speed of training and classification. We came across Maximum Entropy classification as an alternative, and while far slower (by about an order of magnitude), we were able to label recipes much more accurately than with Naive-Bayes. In addition, because there are many different axes of similarity between recipes, we wanted to be able to classify based the results of multiple classifiers. For instance, there is information about the similarity of two recipes in their titles, but also in what ingredients they contain. However, to compare their ingredients in any meaningful way, you can't train a classifier on the same data as you would with the plain text of the title. Thus we implemented a multi-classifier framework, allowing us to aggregate with different weights the results of independently trained classifiers, each working on different elements of a recipe. With this final implementation improvement, we were able to achieve precision of about 91.3% in classifying recipes. Recall, again, was 100%, as we didn't want to throw any recipes away.

As a result of crawling numerous sources, the problem of duplicate recipes was inevitable. Even though a good solution was not apparent at first, we discovered a simple yet powerful technique to de-duplicate our data set. Using a generic Near Duplicate algorithm, we were able to efficiently remove about 7% from our original set of recipes, all deemed duplicates. This type of algorithm is essentially an $O(n^2)$ comparison not unlike many clustering methods, in which a distance metric is computed between each pair of objects in a data space. The difference lies in that, rather than grouping items which are sufficient close, you are marking them for removal. To determine the similarity of any two recipes, we use the following relationship:

$$P(R_i, R_j) = R_{title}(R_i, R_j) + (1 - R_{title}(R_i, R_j))R_{ingredients}(R_i, R_j)$$

In other words, the probability that two recipes, R_i and R_j, are duplicates is a composite of the resemblance of their titles and the resemblance of their ingredients. Below is a formal definition of the resemblance function, $R(i, j)$, where $S(R_i)$ denotes the "lexicon" of either title or ingredients in recipe R_i:

$$R(R_i, R_j) = \frac{|S(R_i) \cap S(R_j)|}{|S(R_i) \cup S(R_j)|}$$

As mentioned, we removed about 7% of our original data set in this stage of the pipeline. (around 20,000 recipes). Testing these results, however, was a bit harder than we'd expected. It

was fairly easy to test the accuracy of removal (92%), as we only needed to look at the recipes removed, versus the recipes they were considered to be similar to. However, testing the accuracy of non-removal (that is to say, which recipes should have been removed that weren't) would be nearly impossible without a labeled data set to work with. Given more time, we may have been able to better validate our de-duplication mechanism, but with the time constraints that we faced, it was not the highest priority.

The final portion of the large-scale data processing pipeline was occupied by user clustering for the purpose of recipe recommendation. If a given user *u* has a known rating history, we can compare that user to another user and compute how similar their tastes are. This is done using a standard sort of vector-space approach, where each user represents a rating vector. Each dimension of the vector is a different recipe, and the value at each dimension is the particular user's rating of that recipe. Once we form clusters of similar users, we could recommend recipes to *u* from the set of recipes highly rated by other members of *u*'s cluster(s), but not rated by *u*.

We faced a fundamental problem in attempting to add this feature. We had no users. We had no rating histories. Designing a user clustering system would be useless to the initial users of our application as there would be no existing users to fuel the recommendation process. We solved this problem by re-crawling the same sites that we had used previously. This time we aggregated the rating data for each recipe, correlating common users across recipes in order to build up a database of existing users and their rating histories. With this problem solved, we began forming clusters of like-minded users.

In order to resolve the problem of clustering users efficiently, we tackled three main goals. First, given the scale of the data set involved, our algorithm needed to be parallelizable. This also implied that the data structures used had to be very fine-grained. More importantly, still, we needed to be able to efficiently re-cluster based on new users or ratings as they became available. This would allow our recommendation system to be much more dynamic and responsive to changes.

The technique developed proceeds as follows. Let us assume there are N users to be clustered. Having M existing user groups in the pipeline, where M approaches N, would produce a greater level of parallelism and increase the accuracy, but would also increase the total complexity of computation. On the other hand, having some M which approaches a constant c, where $N > c > 1$, will definitely reduce the complexity at the cost of LP (level of parallelism) and accuracy. One can see the disadvantages of the latter approach by simply introducing number of Hadoop cluster nodes K_{nodes}, such that K_{nodes} is greater then the number of total user groups. From a utilization perspective this would be very inefficient.

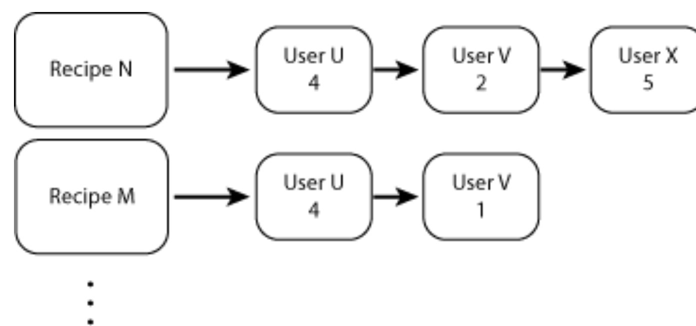
Yet another limitation that was potentially problematic was the size of memory. In our early experiments it was made clear that relying on disk reads or writes during large-scale clustering operations would be prohibitively expensive. We were therefore determined to keep all data memory. There were two approaches that we considered to handle excess data:

1) Have a user cluster grow and periodically check the size. If the size is bound to exceed the limit, break it into N smaller clusters using some heuristic.

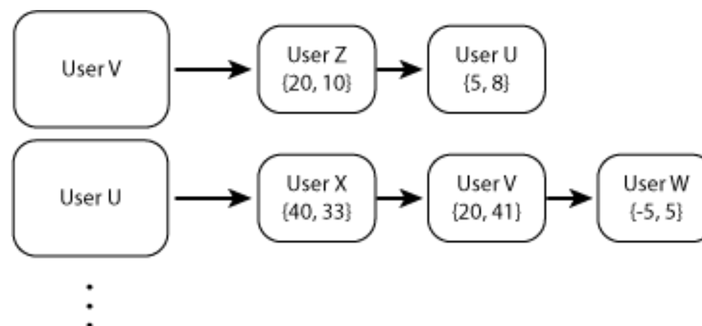
2) Keep the cluster size small with a constant limit. Limiting could be achieved by filtering out recommendations below a certain similarity threshold.

As mentioned, the determining factor was ease of incorporating new data. Clustering data from scratch each time could not be even be considered a solution. Traditional clustering algorithms such as k-Means or any form of Agglomerative Hierarchical Clustering fail to meet this need, and thus we needed a new method of grouping users. It is important to understand that none of the mentioned solutions could be optimal in all cases. We had to carefully choose which implementation to use depending on size, time and complexity.

First, we will talk about the relevant data structures we used for this pipeline. We need to keep track of each rated recipe. This is done through key-value pairs of <Recipe, List<Rating>>. A Rating object consists of two fields:



1) The name of the user who has rated that particular recipe and the associated value, as shown in (2).



2) We also needed another data structure which would keep track of user similarity. We accomplished this via a HashTable<User, SortedList<UserMatch>> where UserMatch contains the name of the user, the score and the number of common rated recipes. The list is sorted based upon the score. Users with higher number of common recipes and low score can purged from the list. With high probability it can be assumed that those users will never yield good recommendations. Each entry of the above mentioned data structures can be stored in a hash table or as a separate files, depending upon the size of the structure.

Once we have a user similarity list, we look at the recipes which have been rated by users with high scores relative to a single user, and recommend their highest-rated recipes.

After obtaining the recipe rating, we populate <Recipe, List<Rating>> pairs. We accomplish this with a MapReduce job that does the following:

1) It computes a similarity metric between all users solely based on a given recipe and outputs two tuples: (User X, [User Y, Score]) and (User Y, [User X, Score]).
2) The reducer iterates through all of the scores for particular users, removes the duplicates by adding up the scores, and finally populates the user similarity data structure ensuring that the list is in sorted order.
3) Finally, both the mapper and the reducer dump the data structures to the filesystem, so that later clusterings can use previous cached results.

Assuming we have established both of the data structures described in the previous section, we need to be able to incorporate new ratings. To accomplish this, we used yet another MapReduce job for the initial clustering along with a new one.

1) Compute and update scores for all new ratings using the existing MapReduce job.
2) Next, for each new rating, we compare it to all of the old ratings for that particular recipe, which produces new scores between users. Using the same algorithm as our first MapReduce job, we can update the user similarity data structure, making sure our list is still in sorted order.

The results of these computations do us little good in SequenceFileFormat, so we wrote an extra script to dump the user cluster data to our web application database. This would allow us to calculate recommendations in real time, and, at the same time, quickly incorporate new data into the recommendation logic as it became available.

Finally, we had the task of implementing a web application to give purpose to all of this data. We chose Ruby on Rails (RoR) as the development platform due to its robustness and implied programmer efficiency. The application was built on top of a MySQL database full of our various data sets created during the data processing pipeline. To ensure that our application would be feature-rich and meet all of our goals, we parallelized our design process, spending many hours prototyping and creating specifications while simultaneously working on the data processing pipeline. This, in combination with our use of Test-Driven development (using a Ruby library called rSpec), allowed us to hit the ground at full speed the moment we had data to work with. By writing test code for almost every aspect of the application prior to coding the application itself, we made development much more straightforward and deterministic.

As a side note, it's worth mentioning that most of the following screenshots were captured using a local instance of our application (and subsequently, only a series of test fixtures populating our database). Hence, there are only three recipes, and a small number of available ingredients.

Upon arrival at our site, the user is greeted by a typical home portal:

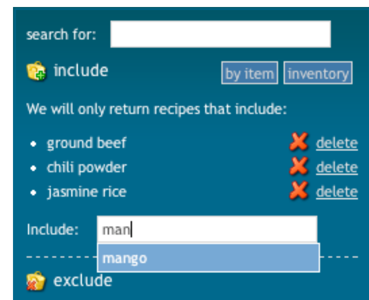
The core search features of the site are available without logging in. Users can browse by category or search more discerningly. To take full advantage of the site, users must create accounts. Were a user to log in initially, they would be given a recommendation rather than a list of top-rated recipes. Recommendations are also offered on a user's profile page.

Note that a personalized menu also appears in the place of the login form. This gives the user access to the app's user-only features.

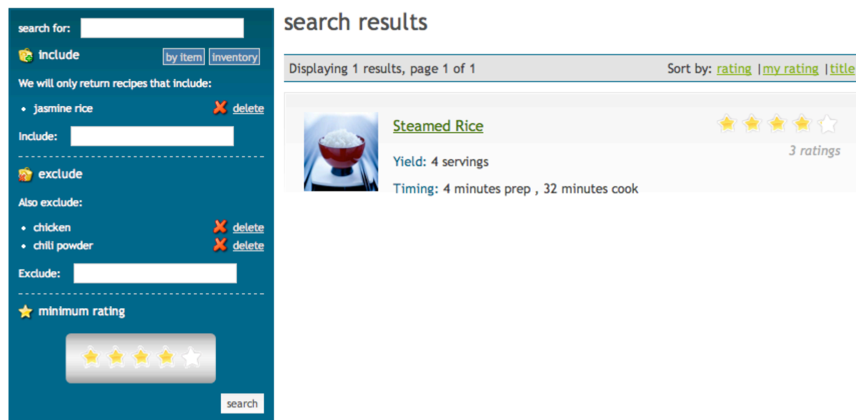
Search is very straightforward. The user can do a standard keyword-only query, which will scan the text of the recipes. The true power of our search engine comes from being able to

search by ingredient (not by keyword for an ingredient). At the most basic level, users can add ingredients to their query that they either want to include or exclude. In both the case of exclusion and inclusion, there are use cases in which users would want to have persistent lists of ingredients ready to go for each search.

Users may often want to know what they can cook using only ingredients they have on-hand. Therefore they can maintain their "inventory," a list of ingredients they currently stock, and with a single click, search only for recipes that are composed of some subset of their inventory. It is equally easy to instead search for recipes that contain specific ingredients. The user can specify ingredients by typing into the "include" text input, and as they type, possible completions will appear. When they select one of these completions, it will be added to the list of ingredients to include. This same AJAX auto-complete ingredient list creation is used all over the site. In fact, the exact same mechanism operates the exclude ingredient feature as well. However, exclusion also sports the use of a user's "dislikes" list, which again is a list of ingredients maintained by the user. Dislikes are automatically excluded from every search, making it easy to avoid allergens and hated foods. Finally, it is also possible to narrow search results by specifying a minimum rating.



When the user receives a set of search results, they can also further refine their query, or sort the results alphabetically, by average rating, or when available, by the user's own rating. Search results will be delivered in paginated form (i.e. 10 per page).



To meet our goal, we wanted to eclipse existing recipe sites. Thus we also provide a standard recipe viewing feature. This page is somewhat important, as it is also the gateway for users to rate recipes, and to add them to their list of favorites. We also implemented a batching system, leveraging our knowledge of required ingredient quantities.

spacewalr.us
cause clowns have big shoes to fill

home search browse categories about contact

You are logged in as ohboyotero. profile update profile favorites inventory dislikes logout

chicken with mango and eggs

Rate it:
2 ratings
Add to favorites

Yield: 3 servings
Timing: 5 minutes prep , 22 minutes cook
Ingredients:
1. 2 fresh mango, diced neatly
2. 10 breasts of chicken, sliced into 1/2 inch wide strips
3. 4 large eggs
4. 1 pint well-washed jasmine rice

Give me batches. Go!

Directions:
1. In a large skillet, brown the chicken breasts in oil till crispy and juices run clear. Add the red onion to the chicken.

Among the most surprising things we found was that it was possible to meet our goals and do so with a high degree of accuracy. Most of us were shocked when we started extracting multi-token ingredients that were almost always correct. Initially we were unsure that there was enough information in a recipe that would allow classification of that recipe into some category of food, but this again turned out to be incorrect. Classification worked very well, though there is much that could be done, from an end-user point of view, to improve (more on this later). In addition, there was a great deal of nervousness when it came to the feasibility of implementation of our web application, and even here we surprised ourself.

This is not to say that all went ideally or according to plan. We were incredibly lucky in that we were able to come across or develop techniques to accomplish the tasks set before us, all the while doing so within our required timeframe. It took us weeks to develop accurate ingredient extraction, and we came very close to giving up. The sheer inconsistency present in recipes was hard to believe, and a daunting obstacle to overcome. Vjeko, for instance, was convinced that natural language processing would help us, and that part of speech would be the key to solving the problem. Yet some were skeptical that the computational complexity added by NLP and POS tagging, and the time lost in researching it, learning to use required libraries, and implementing experimental techniques, would make it prohibitively expensive in all senses. In the end he prevailed (and thanks for that!), but we came very close to missing this mark.

A similar problem arose when we came to user clustering. Some of us had previous experience with clustering data, and were ready with scalable approaches that we thought would serve us well. However, we soon realized that the scalability of an initial clustering meant little if the entire computation needed to be repeated each time a user rated a new recipe. Because of the dimensions that we were dealing with (almost 300,000 recipes and an unbounded number of potential users), it would be impossible to use a traditional clustering method without having sacrifice two key criteria.

First, we would not be able to provide up-to-date recommendations as it would take far to long to re-cluster. Our initial user clustering implementation took about eight hours to execute, and that even with a relatively small user set. This would prohibit us from re-clustering much

more than twice daily, or even less as our data set continued to grow. Second, we would sacrifice the sustainability of the application once separated from CSE's Hadoop cluster. Since we wouldn't necessarily have access to a large cluster once the course ended, we would be unable to provide the services we promised to users, were we to take the site live. The approach that we finally came up with sacrificed some theoretical elegance in order to allow incorporation of new user data with very little computation.

A final pleasant surprise was the development of SpaceWalr.us. David and Daniel have both worked on a number of web applications in the past, some also implemented in Ruby on Rails. Each time we've figured out how to improve the development process so as to be efficient, yet minimize the number of problems we run into. Our last project, for instance, while it only occupied three week's time, was thrown together for the most part in a furious 72 hour coding marathon. Because we began planning SpaceWalr.us far in advance of actually creating it, we could both keep in mind new developments or realizations that came about as a result of our data processing work, and go about its development in a far more structured and agile way. While there are certainly snags that come from rushed development and lack of time, we feel the application is, bugs aside, very well written and organized in a flexible and maintainable way. This is a first!

There are a multitude of ways in which we could have improved our final product. These range from planning to implementation. We all agree that our single biggest source of problems was writing code to do data processing early, but not actually processing the data until it was needed. This created all kinds of problems for a number of reasons:

1) Not all of our code worked quite as well as we'd hoped (shocking!)
2) Some data formats hadn't been sufficiently standardized, creating problems down the pipeline
3) The elements of our pipeline were often in need of serious tweaking due to their production of unacceptable output
4) The task of parsing and loading large amounts of text-based MapReduce output into a structured database was not a trivial task, and was unfortunately left until the last minute.

While 95% accuracy in ingredient extraction sounds impressive, it is not nearly sufficient for a user-facing web application. With ~300K recipes come a huge number of ingredients, and 5% of a huge number is an unacceptable margin of inaccuracy. What this means is that users of our application will run into many irregularities when they work with our actual data set, though they are in fact dealing with a relatively tiny subset of the overall data. There are a number of common irregularities in ingredient lines that we didn't deal with for lack of time (i.e. "3-5 tbsp of sugar"), that in reality wouldn't be prohibitively difficult to address. In addition, there is no doubt that with NLP at our disposal, we could further refine the extraction of modifying words along with their respective ingredients. At the moment our tools have a hard time distinguishing between additional words which add needed levels of specificity to an ingredient and adjectives which are not intrinsic to the ingredient itself. We sometimes err on the side of too many tokens.

In the realm of further work, there is also much to be done. A major feature that we had to sacrifice was the ability of users to add their own recipes. We initially cut this feature due to the inaccuracy of our ingredient extraction process, and while we significantly improved our extraction, the improvement came too late in the project to reintegrate the missing feature. We also left it out because of the complications introduced by user-controlled data. This wouldn't have been terribly difficult to implement in a naive way, but to do it with a compelling UI would be very difficult.

Once this feature was implemented, we'd be able to try some really interesting classification experiments. For instance, given our user rating set and our existing recipes, as well as our knowledge of ingredients, we could approach the problem of trying to pre-rate new user-created recipes. This could involve a number of different approaches, including clustering the new recipe and looking at the ratings of other recipes in its cluster. One potentially worthwhile alternative would be to try to find patterns in the "compatibility" of ingredients. In other words, across the whole set of recipes, are there ingredients that often appear together in highly rated recipes? Perhaps there is a correlation between the worth of a recipe and the ingredients it combines? As with most other problems we faced when dealing with food, there are always a large number of uncertainties that arise from the many factors that determine "deliciousness." Method of preparation, for instance, a topic left unexplored, has a tremendous power over the product of a recipe. While two ingredients might often coexist in popular recipes (e.g. bread and cheese), it is conceivable that those same two ingredients could be prepared together in a manner that is found atrocious to the tongue (e.g. bread and cheese soaked in sour cream at room temperature until moldy).

Another enhancement that would probably improve the user experience (and our accuracy in several data-processing steps) would be to implement multiple-categorization. The categories from our labeled data set overlap significantly. It would only make sense that recipes falling into multiple categories would be classified as such. Again, this falls into the category things that were perfectly feasible to implement, simply not in the alloted time. Due to the way in which we classify, rather than simply taking the single best match, we could take all matches above a threshold, for instance. While there could be a better approach, this one would certainly work well and would be easy to put into practice.

In retrospect, we're very happy with the results of our efforts. We've all done many group projects, and we know how many ways a perfectly diligent group can fail. We feel we set the bar high from the start, and honestly, were pretty convinced that something would have to be sacrificed in order to meet our various deadlines. In the original project goals statement we projected two different levels of achievement for our data processing. The first was realistic, and the latter was ideal. We blew the realistic numbers out of the water, and with the exception of ingredient extraction, also far exceeded the ideal goals (ingredient extraction was ideally 100%). The only dimension which we can't accurately quantify is the accuracy of our recommendations. The additional testing computations required to realistically evaluate these numbers didn't fit in the time allowed. The web application was created feature complete, and while it works well for the most part, there may be a few bugs that come out of the woodwork. In any case, by any measure, we consider this endeavor to be a resounding success. What remains to be seen is how

much of an opportunity we have to continue working on this project. With everything that we've learned, it would be interesting to see how it could evolve.

Appendices

a. Distribution of work

NOTE: for a far more fine-grained look at everything done by each person, as well as our group wiki and our timeline, check out http://trac.spacewalr.us.

Overall architecture design: Top-level design of architecture pipeline, interfaces between pieces, broad algorithmic approaches. *David, Daniel*

Crawling for recipes: Selection of recipe sites from the list we aggregated, coding of crawlers and parsers. *David, Vjeko*

Crawling for user ratings: Re-crawling the same sites, downloading their user rating data, parsing it out and storing it. *David, Felix*

Preprocessing of data: Considerable amount of computation to get data into the formats required by extraction, classification, and clustering jobs. *David, Vjeko, Felix, Daniel*

Quantity and unit extraction from ingredient lines: Discerning quantity and unit of a required ingredient. *Felix, Daniel*

Ingredient extraction: Both design of our interface and implementation of the extraction method. *David, Vjeko, Daniel*

Classification of recipes: Download of labeled data, design of classifiers. *David, Vjeko*

De-duplication of recipes: Research of de-duplication algorithmic possibilities, implementation of algorithms. *Vjeko, Daniel*

User clustering for recommendations: Design and implementation of any of the tested user clustering algorithms. *David, Vjeko, Daniel*

Web application design: Specification and design of features for the web front-end. *David, Daniel*

Usability and feature work: Conducting usability surveys, aggregating and evaluating results, helping gather user input on desired features. *Felix*

Web application implementation: Development of Ruby on Rails web app. *David, Daniel*

Project management, administrivia: scheduling, wiki creation and maintenance, task distribution, problem resolution, report writing, etc. *Daniel*

Sleep: A restorative activity often performed by living creatures in order to maintain energy and recuperative from exertion. *Nobody*

b. External libraries used

- Rails library for Ruby (web application framework)
- rSpec Behavior/Test-driven development package for Ruby
- MALLET (classification)
- Java API to WordNet NLP library and database

c. Instructions for code acquisition and application use

Attached separately.