Kaggle - What's Cooking?
CS 145 Project 1

Team Rocket

Team Member	UID	Role
Akila Welihinda	404268205	Naive Bayesian Classifier, Stochastic Gradient Descent, Decision Tree
Arvin Nguyen	304300135	Fuzzy Approximation, Statistics, Majority Algorithm
Lawrence Yuan	604274601	One vs All, Multinomial Naive Bayes, Statistics
Saad Syed	204281364	Random Forest, One vs One, Logistic Regression, Word Lemmatization
Vir Thakor	304273900	Unique Ingredients, Suffix-Optimization, Statistics

Introduction

Our group attempted several solutions to the “What’s Cooking” competition posted on Kaggle. A brief description of the project problem would state:

> Given a training file of recipes with their corresponding cuisine type (ethnicity) and ingredients of each cuisine, predict the cuisine type based solely on the ingredients in each recipe of a test file.

We follow the basic principles of supervised learning in which we use training data to create some association between the input vector of data (ingredients) and the output value (cuisine). From this association we aim to create a potential function that when given the input data, will output the correct cuisine as accurately as possible. This formulated function must be able to make inferences on new data and adequately solve for the output. The specific instance of supervised learning we’re interested in is classification. We need to develop classifiers to identify the correct output based on the known observations.

In our implementations of the multi-class classification algorithms, we used Python and several assistive scientific computing libraries such as pandas and scikit-learn. Pandas is a software library used for data manipulation and data analysis. Pandas provides tools to assist in tasks related to data structures, which are heavily utilized in examining the training and test data. Scikit-learn is a machine learning library for Python we relied on heavily that provides helpful tools we can use for supervised learning or more specifically classification. We also chose Python because it’s a good general-purpose programming language, it’s very malleable, it comes with many open source libraries that can be utilized for different tasks, a majority of our group members are familiar with the language, and it was a recommended language on the Kaggle forums.

For each of our classification algorithms, we first follow similar approaches in “learning” or recording the training data. From the provided train.json file we collect a list of all cuisines for each recipe and a list of all ingredients for each recipe. From the test.json file we collected the list of ingredients for each recipe.

Data Optimization

Suffix-Optimization

We realized that we can apply an optimization to the train.json and test.json data that we received. The optimization consists of replacing each ingredient in the JSON files with the last word each ingredient. For example let us say that train.json contains the following:

```
[
		{
			“id”: 1,
			“cuisine”: “greek”,
			“ingredients”:
				[
					“lettuce”,
					“olives”,
					“black pepper”,
					“purple onion”
				]
		},
		{
			“id”: 2,
			“cuisine”: “greek”,
			“ingredients”:
				[
					“Romanian lettuce”,
					“black olives”,
					“pepper”,
					“onion”
				]
		}
	]
```

If we apply our optimization to the data above, we receive the following data:

```
	[
		{
			“id”: 1,
			“cuisine”: “greek”,
			“ingredients”:
				[
					“lettuce”,
					“olives”,
					“pepper”,
					“onion”
				]
		},
		{
			“id”: 2,
			“cuisine”: “greek”,
			“ingredients”:
				[
					“lettuce”,
					“olives”,
					“pepper”,
					“onion”
				]
		}
	]
```

This optimization throws out the preceding adjectives in a given ingredient because, occasionally, these adjectives are not useful to us.

This optimization is useful for several reasons. First of all, the optimization causes our algorithms to use less space and time when processing the data because we have compressed the data to contain the relevant portion of each ingredient. However, more importantly, this optimization allows ingredients with varying prefixes to be classified as the same ingredient. For example, we would want to consider “olives” and “black olives” as the same ingredient because there is a small difference between the two. We can view our optimization as a method of “cleaning” or “filtering” the noisy data into simpler, and typically more accurate data.

We realize that this optimization is not perfect and has downsides. For example, applying the optimization will result in our algorithms thinking “garlic powder” and “curry powder” as the same ingredient. This optimization yields better results for relatively simpler algorithms and yields worse results for more complex algorithms. We will use this optimization appropriately to boost our prediction rate and mention when this optimization was used to improve our prediction rate.

Word Lemmatizer

Another data optimization that helped significantly was using a word lemmatizer. A word lemmatizer converts different forms of the same word into one fixed form. We used the WordNetLemmatizer from the Natural Language Tool Kit Python library in order to create a more uniform set of words in our ingredients list. When we pass the word “onions” to the word lemmatizer, it would change it to the word “onion”, thus reducing the noise and discrepancy in our ingredient list. We performed this word lemmatization operation on all the ingredient words in our testing data and training data. We would then take all the lemmatized words in each recipe and convert it into a bag of words, and pass it this to our algorithms later on. Furthermore, we also discarded any non-alphabetical characters in exchange for a space in ingredients, to allow us to create a better bag of words ingredients list. An example of this is “curry-powder” would be changed into two individual words, “curry” and “powder”.

This optimization allowed for more direct comparisons between recipes that used slightly different words for the same ingredients. Using this optimization, our data was much cleaner and organized than the original noisy input that was given to us in train.json and test.json. In comparison to the suffix-optimization, this saw much greater improvements in the accuracy of our algorithms. We decided to use this data optimization for the rest of our algorithms.

Unique Ingredients

Another heuristic that we used was taking advantage of unique ingredients that only appeared under one cuisine classification. If a specific ingredient appeared only in one type of cuisine C, we would automatically classify all recipes which contain this specific ingredient as having a cuisine type of C.

Note that this optimization will not help probability based algorithms (such as the Naive Bayesian Classifier). However, it is relatively simple to implement and could potentially increase our prediction rate, so we decided to include it on our algorithms.

Algorithms

Library Functions

We made extensive use of scikit-learn in our Python programs. A lot of classifier algorithms that we use are modules defined in this convenient library. Many of the algorithms we use take training data feature-vectors as input in order to build the classifier. We then pass the testing data feature-vectors into the generated classifier in order to receive predictions from the classifier. The size of our feature vector is equivalent to how many unique ingredients we have in our entire training data. All of the training data has 6714 unique ingredients. Each element in the feature vector is either 0 or 1; 0 if the current recipe doesn't contain the current ingredient and 1 otherwise. Since we have 39,774 different recipes in our training data, that means that we have 39,774 different feature vectors. Representing our data as feature vectors alone forces us to use 1.1 GB of space (assuming integers are 4 bytes). This large memory requirement can lead to lots of page-swapping in the operating system, which could lead to slow performance in some algorithms.

Naïve-Bayesian Classifier

The Naïve-Bayesian Classifier should be one of the first algorithms we should try when attempting to solve a classification problem. We implemented this first because the Naïve-Bayesian Classifier is very simple to implement (~70 lines in Python) and runs relatively fast (under 5 seconds). We implemented the Naïve-Bayesian Classifier ourselves because we learned exactly how it worked in class and it has a relatively simple implementation.

We ran the Naïve-Bayesian Classifier on train.json in order to first build the classifier. We then ran the Naïve-Bayesian Classifier on the same train.json used to create the classifier, and we got an accuracy of 83%. On the testing data, the Naïve-Bayesian Classifier had a prediction rate of 64%. An accuracy of 64% is decent considering the simplicity and speed of this algorithm.

When the Naïve-Bayesian Classifier was applied to the suffix-optimized data, we received an accuracy boost of 0.3% in our prediction rate. This is not a substantial increase, but is still worth mentioning.

The Naïve-Bayesian Classifier is not very accurate compared to other algorithms. The main reason behind this is because the algorithm assumes all of the presence of features (ingredients) are independent from each other for a given class. This is not always the case in our data. For example, the presence of “eggs” can drastically change meaning of a recipe that has “rice” as an ingredient.

Overall, the Naïve-Bayesian Classifier is useful to us because it is very quick to implement/run and can give us a good baseline to judge the rest of our algorithms. However, the downside is that it is not very accurate. We will explore more accurate algorithms later in this paper.

<u>Multinomial Naive Bayes Classifier</u>

In addition to the classic naive Bayes classifier, we also implemented a multinomial Naive Bayes classifier. This classifier takes into account that the frequency of the words will be in a multinomial distribution. The downside to this algorithm is that it can only take in integer values, but this works well with the word count integer that we are using. The multinomial Naive Bayes classifier was designed specifically for text-based applications, which is the case for this project. This can help explain the big prediction rate increase that we receive when using the multinomial version of the Naive Bayes Classifier.

To implement this classifier, we used the Sci-kit Learn MultinomialNB class. We first optimized the data using the WordNetLemmatizer as well as the non-alphabet sanitizer, then we generate the feature vectors and pass it into the Sci-kit Learn class. The data optimizer, combined with the multinomial distribution assumption, allowed us to increase our accuracy from the basic Naive Bayes classifier by a reasonable amount. With this implementation, our accuracy increased to 72% for multinomial Naive Bayes, from 65% from the normal Naive Bayes. This is a testament to both the combination of our data optimization as well as the better assumption of the word data distribution.

<u>Decision Tree Classifier</u>

We used Scikit-learn DecisionTreeClassifier module in order to build a decision tree from the training data. Using this decision tree, we were able to classify the test data.

Decision tree learning involves creating a decision tree in which classifications of ingredients are mapped to cuisines. This limited method of classification has the potential of leading to many misclassifications for data points in the test file that weren’t present in the training file. The decision tree classifier proved to be very inefficient. Building the decision tree from the test data and classifying the test data took at around 6 minutes. The decision tree approach was also relatively inaccurate. Its prediction percentage on the test data was only 60%. The decision tree approach is both considerably slower and less accurate than the Naïve-Bayesian classifier.

In our implementation, the features of the decision tree are the unique ingredients in the training data. The reason that our decision tree approach is inaccurate is because we have 6714 features in our decision tree. It is known that decision trees overfit data with an excessive amount of features. This explains the slow performance and inaccurate predictions we receive via the decision tree classifier.

Overall, using decision trees for this data set has no benefits. It is both slow and inaccurate and not a well suited classifier for this problem. There are simply too many features for the decision tree classifier to be of any use.

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a simple but efficient way to create a linear classifier. It makes use of Support Vector Machines (SVM) in its computation. The SGD classifier works by minimizing a multivariable objective function by using the “gradient descent” strategy. The “gradient descent” strategy is a method to finding the local minimum of a function by following the negative gradient of a function at a given point until a local minimum is achieved.

We made use of SGD in our Python programs by importing the SGDClassifier module. We converted our training data into feature vectors and then passed the SGDClassifier a set of feature vectors to build the classifier. We then converted our testing data into feature vectors and pass it to the classifier in order to receive predictions.

SGD is a good algorithm for this problem. It runs decently fast considering that it needs to convert all the training data into feature vectors first. SGD runs in about 70 seconds and yields a 77% prediction rate on the testing data. This is a good prediction rate considering the relatively high speed of the algorithm.

SGD requires a lot of training data (more than 1 million samples) in order to achieve good prediction results. Note that although we only have around 40,000 samples of training data, we are still able to get good results with SGD. If we had more training samples, then SGD might be the most optimal algorithm for this application given its prediction rate increase and its decent speed.

SGD is widely used successfully in the industry for text-classification and Natural Language Processing (NLP). Since our classification problem is somewhat related to both of the applications, it can help explain why SGD produces good results in this case.

Random Forest Classifier

The Random Forest Classifier is a fairly successful albeit quite slow classification method. Our implementation finishes creating the solution csv file in around 8 minutes with an accuracy of approximately 76.6%.

Before classifying the test data, we began by first analyzing the training data and the ingredients in the test data. We create a vector for all cuisines encountered by the training data and all id's encountered in the test data. We transform multi word ingredients in either data set into one word by adding the "-" character instead of spaces. Then, we create a space separated string of all ingredients for both the training data and the test data. This simulates a 'Bag of Ingredients' instead of 'Bag of Words' model.

In our classification procedure, we make use of the scikit-learn modules LabelEncoder, CountVectorizer, and RandomForestClassifier. We use LabelEncoder to create a numerical encoding for each of the cuisines present in the training data. We then use CountVectorizer to create a term-document matrix based on the ingredients of the training data, and create a document-term matrix based on the ingredients of the test data. A term-document/ document-term matrix describes the frequency of the training and test ingredients. Finally, we use a RandomForestClassifier to build multiple decision tree classifiers based on the term-document matrix of training ingredients and the encoding of all training cuisines. Using this random forest, we can predict the class for each recipe based on class with the highest mean probability estimate between the decision trees.

In our implementation, we set the number of trees in the implementation to 500 using the n_estimators parameter. Using the default number of trees of 10 results in a lower prediction accuracy of 71.128% but creates the csv file in under 30 seconds on the same computer. By changing the number of trees used we open the possibility of reaching a greater accuracy because more trees are being compared; however, there is a tradeoff in that more memory is necessary for more trees and more time is necessary to build the trees and compare them. Although, there is in fact an upper limit on accuracy which we find to be approximately 76.569% because if we continue to increase the number of trees, accuracy does not increase.

Random Forest Classifier is a decent algorithm; however it is less accurate that other approaches which require less time/memory.

One vs One

We can implement the one vs one classifier using the Scikit-learn OneVsOneClassifier module using the LinearSVC module, a linear support vector classification, as the estimator. This implementation results in an accuracy of approximately 76%.

The one vs one classifier is built by creating N(N-1)/2 binary classifiers for a N-way multi-class problem, where N is the number of classes (or cuisines). Each binary classifier receives two classes from train.json so it can distinguish between these two classes. When creating a prediction, all N(N-1)/2 classifiers are used on each test class and the trained classifier. In order to construct vectors holding the ingredients in each recipe for both training and test data, we used the WordNetLemmatizer module and re.sub() to get the "main" word in each ingredient.

The one vs one strategy is generally slower than the one vs all strategy because of its time complexity of O(N^2) = O((# of classes)^2). One vs one is generally more applicable to situations that don't scale well to number of sample data points because each binary classifier is trained on a small subset of the data. However, the algorithm still performs fairly well despite this inefficiency because of the large number of binary classifiers used on each test case.

One Vs All

Our group also implemented a version of the One vs. All classifier. The One vs. All classifier, also known as the One vs. Rest classifier, creates N different classifiers for the N possible classes. In this case, the classes are the 20 different cuisines. Using the training data, we can train 20 different classifiers for each possible cuisine and then select the one with the highest probability or classification score after running the ingredients for a particular recipe through every single classifier. In practice, this is done by applying the class in sklearn called OneVsRestClassifier.

As compared to the One vs. One classifier, although we are creating 20 different classifiers and running each set of ingredients through each classifier, the One vs. Rest algorithm is actually faster due to not having to create n^2 classifiers to compare each possible cuisine. Our classifier using this multi-class classifier was around 77%, which proved to be more accurate than the One vs. One algorithm. The One Vs All algorithm actually performed the best[1]. It had both the highest percentage accuracy, and also ran reasonably fast.

In general, aside from simplicity, another advantage of One vs All is its interpretability. Since each class is only represented by only one classifier (as opposed to many with One vs One), information about that class can be derived from one specified point.

[1] Not including logistic regression, which makes use of One vs All as a subroutine.

Fuzzy Approximation

We realized that given a recipe, there is most likely another recipe that is very similar to it with just a slight variation. For example, two mexican dishes might consist of cheese, beans, and tortilla, but use different sauces. Thus if we view the training dataset as a knowledge base, for each recipe in the test dataset we can search for the closest matching recipe and assume that they will be for the same cuisine. We find a fuzzy approximation to the ingredients.

We used the ElasticSearch search engine to index all of the recipes and to query the index. The first thing we tried was to index and query the recipes without any configurations to the search engine. The closest matching recipe is the one with the most ingredients shared. This yielded a result of 55%. One optimization we used was ElasticSearch's language analyzer tools that will tokenize the ingredients, similar to the lemmatization strategy used previously. This converts the ingredient “5 Egg Yolks” to “egg yolk”. The reason we want to do this is because we want to consider these ingredients to be effectively the same. This improved the accuracy to 70%.

Lastly, we examined the output and determined that while ElasticSearch might return one cuisine that is the most similar to the input, the next couple of most similar cuisines might all belong to a single cuisine which is different from the first one. So unless the first search result is strongly similar to the input, we take the most common out of top three results. This had marginal improvements, with the accuracy increasing to 73%.

The obvious downside to this approach is the assumption that there is a similar recipe for the input data. This algorithm produces very inaccurate results if the recipe is too novel compared to what exists in the database. Another pitfall is the assumption that all similar recipes are from the same cuisine and emphasizes on quantity of matched ingredients over quality. If a recipe contains an ingredient exclusive to a cuisine (such as a local spice), it’ll unfortunately be matched more closely to one that shares a lot of the basic ingredient (rice, salt, etc.)

Logistic Regression

Another approach we implemented involved classification through logistic regression. This model constructs the probabilities of trial outcomes as logistic functions. Logistic functions are ‘S’ shape, or sigmoid, functions that arise often in a wide array of fields such as neural networks, biology, and economics among many others.

The basic idea involves generalizing linear regression to solve a series of subproblems which compute conditional probabilities used to predict the classification of an item. The fact that logistic regression extends linear regression, a traditional mathematical procedure for data fitting, is only one of many reasons why it is a popular classifier. Logistic regression tends to show up naturally in many cases involving exponential probabilistic distributions. Such distributions relate the probability of some event to an exponential function on the data gathered. These distributions are fairly common within statistics and sciences such as physics, leading to natural use cases for logistic regression.

This classifier gave us our best result of any individual algorithm at 78.6%, while also having a faster run time than most of the algorithms except for the Naive Bayes Clasifier. In the default case, for multi-class classification, the library for logistic regression that we used trains the data with a One vs All strategy. Its speed and accuracy can then to some degree be attributed to the aforementioned properties of One vs All.

Majority Algorithm

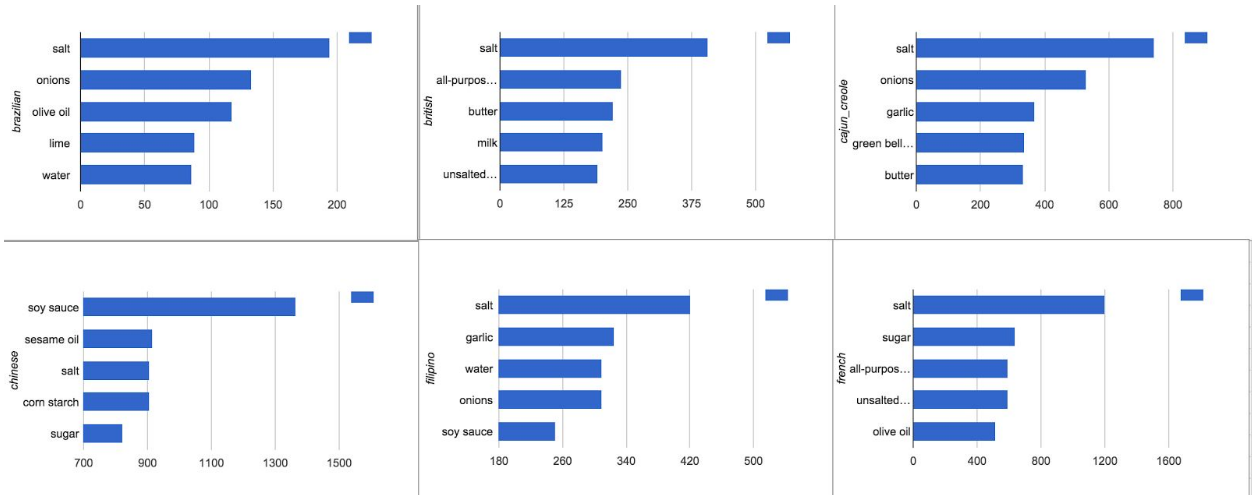
We split the 39,774 recipes in Kaggle’s given training data into a new 30,000 recipe training and test dataset. Thus we know the genuine classification of the items in the test dataset. This allowed us to create our own grading script on and see the actual recipes that were incorrectly classified.

For our best performing algorithms, we used the Unix diff tool to compare the outputs of each one. We noticed that sometimes if one algorithm misclassified a certain recipe, several other algorithms happened to classify the same recipe correctly. Using this discovery, we tried to leverage our multiple algorithms to compensate for one another's weaknesses. For each recipe, we simply classify it using the cuisine that a majority of the algorithms classified it as. If all of our algorithms disagreed on the classification, we used the best performing algorithm's result. The resulting accuracy of using the majority of all of our algorithms was 79.47%, better than any individual algorithm. The algorithms we compared within this approach were One vs All, One vs One, Random Forest, Logistic Regression, Stochastic Gradient Descent, and Multinomial Naïve-Bayesian Classifier

A limitation to this method is that if only one algorithm has the correct classification and it is not the algorithm with the highest percentage correct, then the correct classification will not be chosen as it is not the majority cuisine. However, this method allows for our different algorithms to work together in order to achieve a higher prediction rate.

Further Statistics

In addition to doing the classification algorithms, we also attempted to do some data science and analysis in order to figure out the pain points of our majority algorithm. One approach we employed was figuring out the most common ingredients in each type of recipe, and trying to use that to decide our optimal search algorithm. We compiled a chart to make this clear.



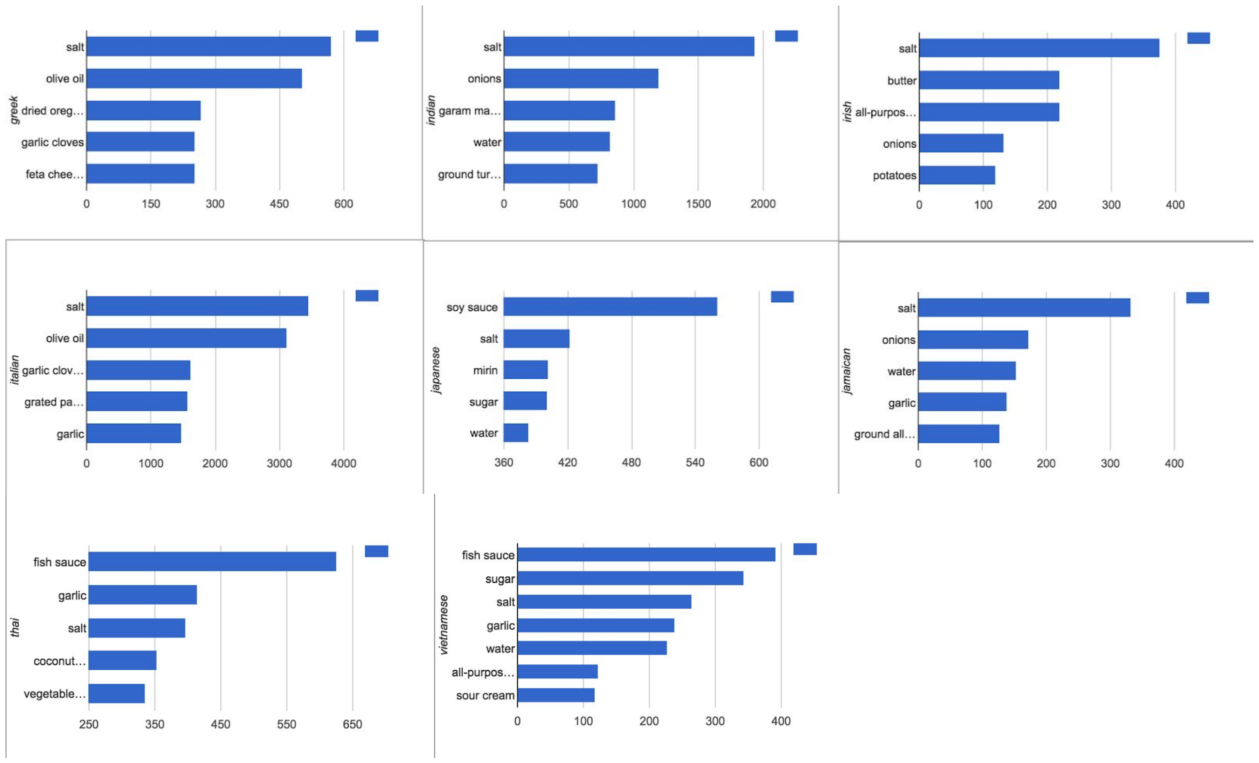


Figure 1: Each cuisine and the five most common ingredients in each.

According to this chart, almost all of the cuisines had salt as a top 5 most frequent ingredient. This was not conclusive. However, there were standout ingredients in each that were not as frequently used in the others. One example of this is the Irish cuisine, which had potatoes in the top 5 ingredients which none of the other cuisines did. Overall, the takeaways from the top 5 ingredients were not incredibly conclusive, which led us to believe that a Naive Bayes' independent variable approach would not improve our results as much as taking an entirely different approach that attempts to factor in combinations of ingredients.

We also attempted to see which cuisines we were misclassifying most often and which cuisines we were confusing them with. We ran our algorithm on our training data, and then ran it on a separate set of test data. The top three cuisines we were predicting incorrectly were Italian (16% wrong), Mexican (15% wrong), and Southern US (10% wrong). Every other cuisine had significantly less error rate ranging from 1%-7%. Below is a table for the top three cuisines we mistook each of the above for.

	Mexican	Southern US	Indian
Error Count	324	219	142
Percentage of Total Italian Food Errors	21%	14%	9%

Figure 2: Top 3 cuisines we mistook Italian food for.

	Italian	Southern US	Indian
Error Count	315	173	123
Percentage of Total Mexican Food Errors	23%	13%	9%

Figure 3: Top 3 cuisines we mistook Mexican food for.

	Italian	Mexican	Chinese
Error Count	224	176	88
Percentage of Total Southern US Food Errors	24%	19%	9%

Figure 4: Top 3 cuisines we mistook Southern US food for.

As shown above, we mistook Italian, Mexican, and Southern US food for each other regularly. This was the main contributor for our errors. If we were given more time, we could take these results and try to specifically find out why each of these cuisines were predicted incorrectly much more often than the other ones, and change our algorithm appropriately.

However, the incorrect classifications were usually culturally and ethnically close to the correct cuisine.

Conclusion

In conclusion, our group went through many classification methods to attempt to sort new recipes into their predicted cuisines. We went through many routes to both optimize the data we got, as well as optimize each algorithm individually and build another majority layer on top of that. In the end, we managed to receive an accuracy of 79.47% on Kaggle, which at the time of submission, was enough to place 133rd.

If we were to seriously further pursue this project, we would take a number of steps. We already analyzed the faults of our algorithm - which cuisines were commonly mistaken for which other cuisines. Based on this data, we could fine tune our algorithms manually to have better sensitivity between the Italian, Mexican, and Southern US cuisines.

Furthermore, given the constraints of the project, we were only allowed data based on the ingredients list for every recipe. However, this data may not have been enough to classify all recipes. An example of one recipe we got wrong contained the ingredients list: {bread, milk, butter, gravy} and was British. This is a pretty confusing recipe as only looking at the ingredients list, it is very hard to tell the origin is British. Another set of variables that would help would be the name of the recipe as well as the preparation methods. The name of the recipe could give us clues to the cuisine based on the language roots of the words. Certain cultures also favor certain preparation methods - an obvious one is that the Chinese cuisine steam and stir-fry very commonly.