What's Cooking?

Classifying Cuisine by Recipe Ingredients

Andrew Gao, Jayd Hill, & Seth Plunkett

ABSTRACT

The act of classifying items into different categories based on a small number of features is applicable to many important fields. In this paper, we use a Kaggle competition on classifying recipes based on ingredients as a testing ground for three learning algorithms. We evaluate a random forest, neural network, and Support Vector Machine (SVM) for accuracy. Our results show that an implementation of the SVM produced the highest performance.

Keywords: classification, recipes, Kaggle, neural network, linear kernel

1. INTRODUCTION

Food and culture are closely tied all around the world, and in this century we are lucky enough to have access to recipes for many of these dishes online. Different cultures show themes of ingredients used and in what combination, often based on what has traditionally been available from the surrounding environment, and these styles and flavors make up regional cuisines as we know them today.

The Kaggle competition "What's Cooking?" poses the interesting question of if these ingredient patterns are enough to classify the cuisine of a specific recipe. Using a dataset of real recipes provided by Yummly - with no cooking times or methods known - we want to accurately predict the cuisines of an unseen set of recipes from Kaggle.

Although this project is intended as an exercise for learning and practice, digitization of records (e.g. old cooking magazines) could benefit from an algorithm that sorted by cuisine or something similar. More generally, algorithms that can effectively classify a set of text attributes obviously have a number of significant applications from marketing to science to health care.

In developing a strategy for this project, we first looked at the kernels and discussions on Kaggle, as well as looking online to see how similar classification problems have been approached. This work seeks to apply a variety of known machine learning algorithms and techniques to achieve the highest accuracy possible. First we will do some basic cleaning and pre-processing, although not much is required for this data set. For our predictions, we have elected to implement three different algorithms: A random forest, a neural network, and a Support Vector Machine (SVM).

The rest of this work is organized as follows: Section 2 provides the necessary background to understand this paper, supplying a detailed description of the issue we are trying to solve; Section 3 briefly outlines our approach, emphasizing why we chose the algorithms and stemming techniques we did; Section 4 details our implementation of these methods; and in Section 5 we discuss our conclusions.

2. BACKGROUND

This section describes the Kaggle challenge in greater detail, describing the given dataset in 2.1, followed by our decision to approach this as a classification problem, and finally an explanation of how and why we chose the features we did.

2.1. The Data

The data Kaggle provides is available in JSON, which is convenient in that even though it is a somewhat non-standard dataset, it is easy for both humans and programs to read. Also, it allows us to easily process objects of different sizes. There were 39,774 recipes in the training set and 9,944 in the unlabeled test set. We found 20 different cuisines and 6,714 "unique" ingredients (keep in mind that this number counts items like "greek yogurt" and "low-fat greek yogurt" as distinct ingredients).

Italian was the most common cuisine, making up nearly 20% of all of the training data [see Figure 1 for number of recipes per cuisine].

2.2. Formulating Challenge as a Machine Learning Problem

The goal of the Kaggle competition is to correctly predict the cuisine of an unclassified recipe given a training set. As such, we formulated the problem as a supervised learning classification problem, and later extended this by adding a probability estimation with two of our algorithms.

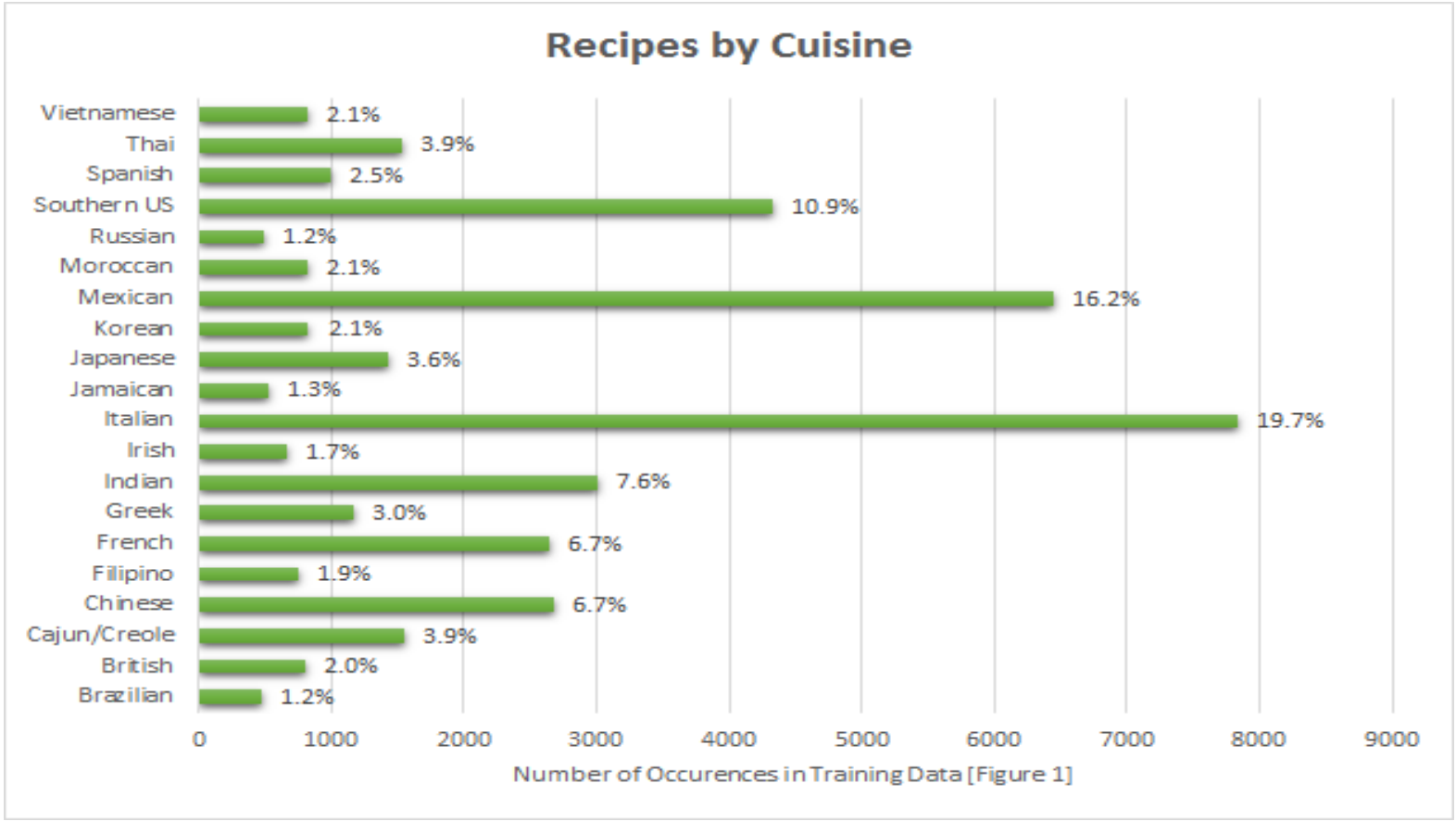
2.3. Feature Selection

Approximately 40,000 recipes, 6700 unique ingredients, and 20 different cuisines means a lot of different possible combinations. To simplify this a little bit, we selected features that were neither too common nor too rare, as described in the preprocessing section below. This allowed us to dramatically decrease the number of features while also avoiding overfitting.

3. METHODS

3.1. Preprocessing

Given the size and state of the data set, our first step was preprocessing the data. The primary challenge was to the huge number of attributes (ingredients), many of which were multiple words or slight variations on one word. We experimented with the following methods to address these issues:



A. Stemming

While stemming seemed to be a logical choice, given that the data includes ingredient names that may essentially describe the same ingredient (such as onion and onions), this method was problematic given then the list of attributes was so large and each word would need to be individually stemmed. Performing a global stem was ineffective in that it caused words to be incorrectly stemmed (for example leaves stemmed to leave instead of leaf or lea). After testing a global stem function, we concluded that it was best to not stem at all.

B. Multi-word Expressions

Many ingredients in this data set are composed of multiple words. Often a word or words composing the ingredient name could correctly identify the word, which would allow us to group ingredients together. For example, "light sour cream" and "reduced fat sour cream" could be grouped with "sour cream". This would reduce the number of attributes and increase the number of examples, both of which could increase our accuracy. Ultimately, this method proved to be unhelpful because it was impossible to determine programmatically which word or words in the ingredient were the "key" word that should define the attribute.

C. Sparse Term Modification

This proved to be the most useful and cost-effective method. We chose to remove ingredients with three or less total occurrences on the basis that they were so infrequent as to have little to no predictive power and could cause overfitting. We also used term frequency - inverse document frequency (tf-idf) to remove the most frequent ingredients (such as "salt") which were so prevalent as to be not very useful for identification. This method noticeably improved our accuracy - as long as we did not remove too many words. Specifically, applying the sparse term removal function at .98 gave us the optimum prediction rate. If this was tuned down, our prediction accuracy dropped quickly, which revealed how important it was to remove the most general terms, but not take this method too far, as we could easily lose valuable attributes.

D. Tokenization

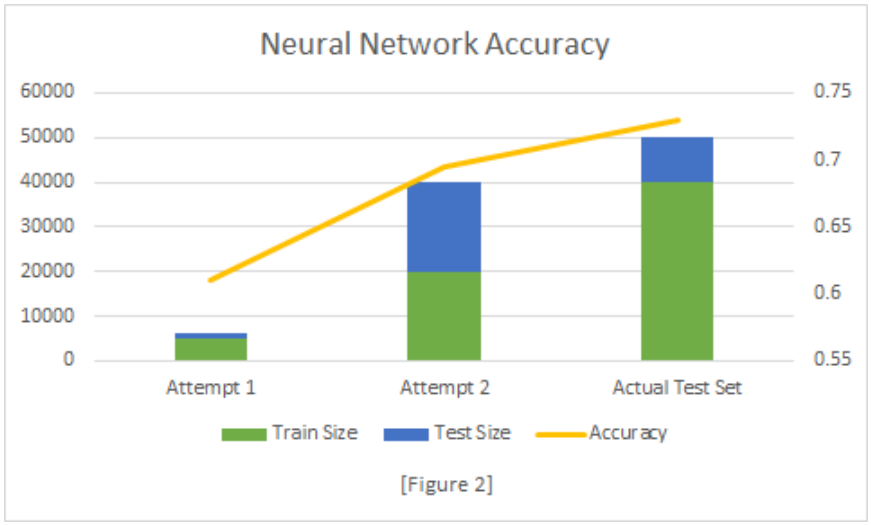
Tokenization, or breaking a string into tokens (in this case, separating out our ingredients), was automatically incorporated into our code by the fact that the file was read and processed line by line so each line was read as a single ingredient. It was therefore unnecessary to use another tokenization method.

E. Lowercase

Although initially planned on, we decided not to apply a lowercase method to the dataset in every case because the ingredients are almost uniformly lowercase. We do not believe it would have caused a statistically significant change in results.

3.2. Algorithm Selection

Now that we had classified this problem as one of classification, it was time to choose our algorithms. We decided to compare three different algorithms based on their accuracy. Kaggle uses predicting "Italian," the most common cuisine, for all recipes as the baseline. Italian food made up 19.7 percent of the training set, and scored a 19.15 percent on the validation set. So, we used 20 percent as our initial benchmark to beat. On the other side, the Kaggle leaderboard shows a high score of 83%. Anything close to that accuracy is a definite success.



[Figure 2]

A. Random Forest

A random forest is an ensemble of decision trees that constructs a large number of trees at training time, then selects the most common class as its output. We chose this as it is a fast, flexible approach that resists overfitting. This technique gave us a score of 68% on the test data - much better than random, but not as good as our other algorithm applications.

B. Neural Network

On the positive side, neural networks are high accuracy, provide nice theoretical guarantees regarding overfitting, and with an appropriate kernel they can work well even if our data is not linearly separable in the base feature space. The downside is that neural networks are memory-intensive, can be hard to interpret, and can be challenging to tune. After a few iterations [see Figure 2], we got our neural network up to 73% accuracy on the test set. Additionally, for the sake of our own interest, we added a probability estimation [see Figure 3 for an example of this output]. Upon running, our program displayed any cuisine with a 10% chance or higher, along with the percent probability that it was that cuisine, ranked from most to least probable. This was an interesting way to see how distinct but similar cuisines were scoring with our algorithm. For example, with this method we saw some cuisines from


```
{
    "probabilities": {
        "italian": "0.83",
        "french": "0.17"
    },
    "cuisine": "italian",
    "id": 24631,
    "ingredient": [
        "olive oil",
        "garlic",
        "dried sage",
        "salt",
        "ground black pepper",
        "wine vinegar",
        "trout fillet",
        "dried rosemary"
    ]
}
```


[Figure 3]

similar regions (e.g. Italian and French) were frequently grouped together.

C. SVM

Finally, we chose a Support Vector Machine - specifically, a linear kernel/ linear SVC (Support Vector Classification). This method is frequently used for text categorization, and is good for a large number of attributes like we have here.

4. EXPERIMENTAL STUDY

In the next section, we walk step-by-step through the algorithm experiments we conducted, and detail the process of designing our neural network. Following is an analysis of our results.

4.1. Algorithm Implementation and Design

A. Frequent and Sparse Term Modification

We loaded the train and test attribute lists into a Corpus, which is a collection of text documents, and then a Document Term Matrix which allowed us to get the frequency of each attribute in the dataset.

To eliminate the features that were too sparse, we deleted features that appeared less than 3 times in the corpus. To eliminate those that

were too common, such as the ingredient, salt, we eliminated features with a idf (inverse document frequency) of over 5. The equation for idf that we used was:

$$IDF = log(\frac{N}{1+n})$$

where N is total number of recipes and n is the number of recipes containing the term that we are finding the idf of. We used removeSparseTerms(), which we set to .98 (on a scale of 0 to 1), meaning only the most common words would be omitted from the dataset.

It was notable how the accuracy of our predictions was substantially increased by tuning the parameters that determined our attributes.

B. Random Forest

This algorithm was implemented in the R programming language using the randomForest package. In the preprocessing section of the program, we applied the tolower() function to make all the features consistent. We then applied the gsub() function to replace all occasions of dashes or blank spaces between ingredient words with underscores. This allows the features to be vectorized.

Each attribute was then converted to a column in a data frame. The intersect function was then applied to the data frame to return a sorted row vector of common values of vectors from the training and test sets.

The train vectors were then encoded as factors or categorical variables. We were now ready to initialize the random forest. We called the predict method on the random forest, which returns the proportion of votes of the trees in the ensemble. The predictions were finally outputted to JSON and CSV files for testing.

It was interesting to note how quickly the algorithm ran in comparison to the linear SVC and especially the neural network.

C. Neural Network

Our neural network was written in python using the scikit-learn library. First, we pre-processed the data by eliminating features that were either too common or too sparse. The data was then binarized with scikit-learn's binarizer so features for each recipe became a binary vector. From here, the training data was trained on a neural network using scikit-learn's multi-layer perceptron classifier class. The probability for each output label (a cuisine in this case) was calculated and the one with the highest probability was chosen as the predicted label. Our neural network achieved an accuracy of 73%.

D. Linear SVM

The linear SVM was applied using the scikit-learn Linear Support Vector Classification class. It implements a linear kernel and has good flexibility in the choice of penalties and loss functions, while scaling well to the large sample size.

First we loaded the ingredients into a sparse matrix. The we loaded an estimator and called its fit() function which fit the model according to the given training data. Finally, the predict method predicts class labels.

We kept this algorithm pretty basic, but it was actually our most accurate. Had we more time, we could have experimented with different kernels and using the tune() which tunes hyperparameters using a grid search over supplied parameter ranges.

4.2. Analysis of Best Performance

Our highest-performing algorithm was the Linear SVC. We theorized that since text classification is known to be handled well by linear SVMs for the following reasons:

1) Text is inherently discrete
2) Creates a high dimensional input space
3) Sparse document vectors
4) Linear separability (often)

It has also been discussed that a linear kernel is beneficial for text classification when the text snippets are small, such as in our features. It also performs well with large data sets. A linear kernel also happens to be fast and easy to optimize.

For these reasons, the linear SVM was a good choice, possibly only second to a Gradient Boosting Machine that has been winning so many of the Kaggle competitions.

5. CONCLUSIONS

In this work, we compare the accuracy and overall performance of three different learning algorithms on the task of categorizing recipes by cuisine. While the neural network scored relatively high on accuracy, the steadily increasing run-time with larger training sets make this impractical for larger datasets. The SVM, however, performed highest on accuracy as well as running relatively quickly. The random forest, meanwhile, was speedy but far from the most effective at classifying these data.

Given our limited timeframe with this project - and the fact that we did not beat the high score on Kaggle - we conclude that there is more work that could be done in this area. It seems likely that an ensemble, specifically something like gradient boosting, could produce a higher accuracy. In the end, though, we expect that it is not possible to reach 100% accuracy, and probably nothing close to that. Considering the nature of this dataset (pulled from user-submitted recipes), there are bound to be examples that are miscategorized, as well as recipes with too few or too common ingredients to predict correctly. For now, though, we are glad to have successfully implemented several learning algorithms to something close to a real-world problem.

REFERENCES

Python Programming Tutorials. (n.d.). Retrieved March 19, 2017, from https://pythonprogramming.net/linear-svc-example-scikit-learn-svm-python/

Feature Selection in Python with Scikit-Learn. (2016, September 21). Retrieved March 19, 2017, from http://machinelearningmastery.com/feature-selection-in-python-with-scikit-learn/

Linear Kernel: Why is it recommended for text classification ? (2017, March 20). Retrieved March 21, 2017, from http://www.svm-tutorial.com/2014/10/svm-linear-kernel-good-text-classification/

What's Cooking? | Kaggle. (n.d.). Retrieved March 19, 2017, from https://www.kaggle.com/c/whats-cooking