CS63 Spring 2018
Recipe Ingredients

Michelle Ma and Haochen Wang

1 Introduction

For our final project in CS63: Artificial Intelligence, we decided to participate in a Kaggle machine learning contest to solve a supervised learning problem. The specific machine learning contest we entered was focused on using recipe ingredients to categorize the cuisine.

To accomplish this task, we used a neural network as well as an ensemble learning method. In deciding to use neural networks, we took into account their capability of modeling and processing non-linear relationships between inputs and outputs in parallel. (`https://www.kdnuggets.com/2016/10/artificial-intelligence-deep-learning-neural-networks-explained.html`) Ability to operate on non-linear datasets without placing restrictions on input variables is an important strength of neural networks given the many real-life, non-linear relationships between inputs and outputs. Moreover, neural networks are particularly good at generalizing data. After learning from the initial inputs and their relationships, it can infer relationships on unseen data as well, thus making the model generalize and predict on unseen data. (`https://towardsdatascience.com/introduction-to-neural-networks-advantages-and-applications-96851bd1a207`) Given that our dataset is non-linear and given the aforementioned strengths of neural networks, we thought using a neural network would be a good fit.

In addition to neural networks, we also tried the Random Forest ensemble learning method. In general, ensemble learning works by combining the output of many weak classifiers to make a strong classifier that outperforms all of its component parts. The two most common methods for ensemble learning are boosting and bagging. The key idea of boosting is to change the algorithm by restricting its complexity and/or randomizing. The key idea of bagging (bootstrap aggregating) is to change the data set by sampling with replacement. Overall, bagging fits complex models to resamples of the dataset, where each model will over-fit to its sample, and bagging takes lots of samples and votes across them to reduce the overall variance. Boosting fits simple models to the entire dataset, where each model will be under-fit to the data set, and as long as the biases are uncorrelated, voting reduces the overall bias.

The Random Forest algorithm uses bagging to overcome several problems with decision trees, such as reduction in overfitting by averaging over several trees as well as reduction in variance, by training on re-samples of the data. Furthermore, the randomness of the Random Forest Algorithm is seen in the training phase by taking a random subset of all available features, and identifying the best split feature over a finite set of iterations. Overall, the Random Forest algorithm is an

excellent classification algorithm because it classifies large datasets with accuracy. Thus, we felt like using Random Forest over a single decision tree would be the best ensemble learning method to accomplish the task of building a recipe classifier.

2 Method and Details

Our specific Kaggle competition provided a train.json, which is the training set containing recipes id, type of cuisine, and list of ingredients. It also provided a test.json, which is the test set containing recipes id, and list of ingredients. The cuisine type is removed, as it is the target variable we are trying to predict. Thus, we only used the train.json file, because we can then use the correct dish label provided to check the accuracy rates of our models.

We first parsed the train.json file to obtain an array of ingredients and an array of cuisines. Since Kaggle intentionally does not provide labels for their test set, as they use the labels for the test set to gauge accuracy of the models submitted to the contest, we split the array of ingredients and cuisines, reserving 80remaining 20set, this way we are able to gauge the accuracy of our model by checking how our model outputs compare to the given labels.

In terms of how much data we were given, the train.json file initially had approximately 38,183 dishes, so after we did the 80-20 split as explained above, the training set has 31,819 dishes to predict and the testing set has 6,364 dishes to predict. To visualize our dataset for trainIngredients and testIngredients, we wanted to create a boolean matrix where rows indicate a dish and columns represent the presence of an ingredient in that dish, out of a set of all possible ingredients across all dishes. The total number of unique ingredients across all dishes was 6,311. Thus, this means that trainIngredients has dimensions 31,819 (number of dishes to predict) x 6,311 (total number of unique ingredients across all dishes) and testIngredients has dimensions 6,364 (number of dishes to predict) x 6,311 (total number of unique ingredients across all dishes). To visualize our dataset for trainCuisine and testCuisine, we wanted a vector of numbers which indicated the category of cuisine that each dish belonged to. When we printed out the shape of trainCuisine and testCuisine, we found that their shapes were (31819, 20) and (6364, 20) respectively, which indicates to us that there are 20 unique cuisines across all possible dishes. Therefore, trainCuisine has dimensions 31,819 (number of dishes to predict) x 20 (total number of unique cuisines across all dishes) and testCuisine has dimensions 6364 (number of dishes to predict) x 20 (total number of unique cuisines across all dishes).

To accomplish this, we used the scikit-learn package, specifically the fit_transform method on a CountVectorizer object. The fit_transform method first "fits" the feature extractor itself to determine what features will base future transformations. Then, the method "transforms" the data by tokenizing the strings and producing count vectors for the data. At first we attempted to used this idea on trainIngredients and testIngredients but found out that CountVectorizer objects didn't work on duplicate vocabularies, which didn't work well for recipes with ingredients that had duplicate vocabularies (i.e. if a recipe had purple onion and green onion, the classifier would have a difficult time realizing that two different types of onions appeared in the recipe). Therefore, we had to manually create the boolean matrix for trainIngredients and testIngredients. Since cuisines

usually do not involve duplicate vocabularies, we were able to avoid the aforementioned problem and apply CountVectorizer objects to transform trainCuisine and testCuisine into boolean matrices. After this transformation, we applied the NumPy operation "argmax" with a parameter of 1 to grab the column which has a 1 in its bucket, which indicates the cuisine type.

For our model, we tried neural network as well as the Random Forest ensemble learning method. For our neural network, we added two hidden dense layers, with a dropout layer of dropout probability = 0.1 between the first and second hidden layers. For our first and second hidden layers, we had 100 nodes, using the ReLU activation function. The reason why we added a hidden layer of 100 nodes is because more nodes makes the network more powerful, as even if each node in the layer is computing a simple function, the aggregate computation of the entire layer of 100 nodes can result in the completion of a more complex function. Moreover, the reason why we chose ReLU as the activation function is because ReLU has a constant derivative. Thus, when we do backpropagation, the contribution to the change in the cost for each weight is more significant because we no longer encounter the Vanishing Gradient problem which occurs when using the sigmoid function. Our output layer consists of 20 nodes and uses the softmax activation function. The reason why we use 20 nodes is because there are 20 possible cuisines each dish can be categorized as. The reason why we chose softmax as an activation function for our layer of output nodes is because "the output of the softmax function can be used to represent a categorical distribution that is, a probability distribution over K different possible outcomes." (`https://en.wikipedia.org/wiki/Softmax_function/`). Since our output is one of 10 numbers, softmax suits this classification problem well, which is why it increases our neural network's prediction accuracy.

Moreover, the reason why we chose Adamax for our optimizer was because Adamax has the benefit of RMSProp, an algorithm that does well on online and for noisy problems, and also makes use of second moments of gradients. It is also more appropriate for problems that are large in terms of data and with sparse gradients, which describe our dataset well (`https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/`).

In terms of training our system, we first fit the neural net using boolTrainIngredients (our transformed training input, as outlined above) and boolTrainCuisine (our transformed training output) and validated our model with boolTestIngredients and boolTestCuisine to check the accuracy of our model. We did this for 10 epochs, as we noticed that the accuracy rate stagnated at that point.

We also tried using the Random Forest ensemble learning method for our model, in which we imported the RandomForestClassifier from the sklearn.ensemble package. We gave our RandomForestClassifier 500 n_estimators, which is the number of trees that are built before taking the maximum voting or averages of predictions. Having 500 n_estimators was advantageous because higher number of trees generally yields better performance and makes predictions stronger and more stable, but it wasn't too high that our processor could not handle it. Additionally, we gave our RandomForestClassifier max_features of "auto" which will simply take all the features which make sense in every tree, without placing an restrictions on the individual tree. (`https://www.analyticsvidhya.com/blog/2015/06/tuning-random-forest-model/`). Lastly, we gave our RandomForestClassifier a class_weight of "balanced", wherein classes are automatically weighted inversely proportional to how frequently they appear in the data (`https://chrisalbon.com/machine_learning/trees_and_`

forests/handle_imbalanced_classes_in_random_forests/); this is particularly useful when dealing with imbalanced classes in Random Forest, which was characteristic of our data.

3 Results

For our neural net, when we run the test sets (boolTestIngredients, boolTestCuisine) on our model, we get an accuracy rate of 93concluded that a 93model's inaccurate classifications were minor. For example, we noticed that our model classified a Vietnamese dish as Thai, but upon looking at the ingredients of the dish, we noticed that the Vietnamese dish did share a lot of the same ingredients as other Thai dishes, such as fish sauce, rice noodles, and cilantro leaves. Overall, because our inputs are ingredients and each country's cuisine usually has a few distinct ingredients that are neural network can capitalize on, it makes sense that our neural network has a relatively high accuracy rate.

For our random forest ensemble model, we scored a higher accuracy rate of 99This was surprisingly high, but since we have an ensemble that is able to aggregate the strength of many smaller trees and pick out

In this section you should show and analyze the results. Measure the performance of your system, and if possible compare your performance to other implementations. Use tables and figures to illustrate the results. If you can't fit all of the pictures that you'd like to show in the paper, you can make an accompanying web page and point the reader to it.

Even if your project is not as successful as you'd hoped, you still need to show results. This section is one of the key parts of any scientific paper. Be sure to provide adequate information so that the reader can evaluate the outcomes of your experiments.

4 Conclusions

This section should sum up what you did and what you found.