CS63 Spring 2018
Recipe Ingredients

Michelle Ma and Haochen Wang

5/15/18

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 used 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, 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. 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.

The Random Forest algorithm uses bagging to overcome several problems with simpler models like the 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 that using a 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 file, which is the training set containing recipes id, type of cuisine, and list of ingredients. It also provided a test.json file, which is the test set containing recipe id and list of ingredients. Kaggle intentionally does not provide the cuisine type, or labels, for their test set, as it is the target variable that participants are trying to predict. Additionally, the held-out correct labels for the test set will be used to gauge the accuracy of the models submitted to the contest. Thus, we only used the train.json file, because we can then use the known dish label provided to check the accuracy rates of our models.

We first parsed the train.json file to obtain an array of all ingredients and an array of all cuisines. Then, we split these two arrays, reserving 80% of the array of ingredients and cuisines as our training set (trainIngredients, trainCuisine) and leaving the remaining 20% to serve as our test set (testIngredients, testCuisine). Since we are given the correct labels for the training set, 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 38,183 dishes, so after we did the 80-20 split as explained above, the training set has 31,819 dishes for model fitting and the testing set has 6,364 dishes for prediction. 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 train) 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 train) x 20 (total number of unique cuisines across all dishes) and testCuisine has dimensions 6,364 (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 to 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 multiple same terms (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 only refer to one type, 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 a 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 20 numbers, softmax suits this classification problem well, which is why it increases our neural network's prediction accuracy.

Moreover, we chose Adamax for our optimizer because Adamax has the benefit of RMSProp, an algorithm that does well on online and 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/`)

Furthermore, we chose sparse_categorical_crossentropy as our loss function based on three main factors: number of outcomes, the type of output, and the type of target. We have more than 20 outcomes, so we were looking for a loss function that produced more than 2 outcomes, i.e. more than binary categorical crossentropy could offer, for example. We were also looking to work with probabilistic output, as the "fit_transform" method we called on trainCuisine (i.e. our yTrain dataset) produces a matrix of probabilities. Lastly, we were looking for "hard" targets, as we wanted a definite cuisine label as opposed to a "soft" target, which would have given probability statements about how likely a dish was to be a certain cuisine label. Thus, because sparse_categorical_crossentropy as a loss function accomplished these three main factors, we concluded that it would be the best fit loss function for our neural network.

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 final implementations of our neural network and ensmeble model, we achieved accuracy rates of approximately 93% and 99%, respectively. Our explanations for the accuracy of our models is provided below.

Neural Network. For our final implementation of our neural net, we used the following parameters: the Adamax optimizer, the sparse_categorical_crossentropy loss function, and two hidden dense layers with the ReLU activation function, and an output layer using a softmax activation function. For our finalized neural net, when we run the test sets (boolTestIngredients, boolTestCuisine) on our model, we get an accuracy rate of approximately 93%. Although somewhat high, we concluded that a 93% accuracy rate ultimately made sense, because a lot of our model'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 our neural network can capitalize on, it makes sense that our neural network has a relatively high accuracy rate.

In order to compare our final neural net against some other implementations, we varied the optimizer of our neural network with the number of dropout layers of our neural network architecture and compared the accuracy rates. We kept the loss function constant. Our results from these experiments are illustrated in Table 1 below:

From Table 1, we can see that Adamax performs the best against the other optimizers, RMSProp and SGD, with a 92.33% accuracy. As mentioned above, we note that this is because

	1 Dropout Layer	2 Dropout Layers
RMSProp	91.25%	90.85%
Adamax	92.33%	91.67%
SGD	73.18%	73.74%

Table 1: Optimizers and Dropout Layers

Adamax has the benefit of RMSProp, an algorithm that does well on online and noisy problems, and also makes use of second moments of gradients. Thus, because RMSProp encompasses the benefit of Adamax, it follows that its accuracy would be higher. The reason why SGD had the lowest accuracy rate, at 73.18%, is because it has a learning rate that is proportional to the inverse of the number of iterations. Thus, after a long run, it will take smaller steps and can easily get stuck in poor local minima. RMSProp is able to adjust the step size and gets iteratively updated with a running average that is on the same scale of the gradients (`http://www.erogol.com/comparison-sgd-vs-momentum-vs-rmsprop-vs-momentumrmsprop/`).

Moreover, since ensemble learning is oftentimes slow and inefficient, we thought it would be a good idea to work with dropouts layers as a more efficient alternative, since dropout layers can often mimic the effects of ensemble learning methods. However, after varying the number of dropout layers from 1 layer to 2 layers, we noticed that it didn't have as significant of an impact on the overall accuracy rate of our neural network.

Ensemble Model. For our final implementation of our random forest ensemble model, we used 100 n_estimators, "auto" max_features, and a "balanced" class_weight. With this model, we scored an accuracy rate of approximately 99%. This was very high, but since we have an ensemble that is able to aggregate the strength of many smaller trees with 100 n_estimators and select features well with the specified parameters, this result is also not too surprising.

In order to compare our final neural net against some other implementations, we varied the number of estimators and the type of max_features and compared the accuracy rates. We kept the class_weight constant. Our results from these experiments are illustrated in Table 2 below:

	Auto	Log2
5 trees	96.50%	97.06%
10 trees	99.25%	99.42%
100 trees	99.95%	99.97%

Table 2: Number of Estimators and Maximum Features

From Table 2, we note that the accuracy of our ensemble model has a positive correlation with the number of estimators, which corroborates our final ensemble model's use of 100 estimators over 5 or 10 estimators. We also note that overall, log2 performs better as a range for the number of features to consider when finding the best split. We also note that as the number of estimators increase, the difference in accuracy rates between using auto and log2 begins to minimize. Thus, we decided to use 100 estimators, our decision to use either auto or log2 for our max_features was relatively trivial.

4 Conclusions

After undergoing multiple experiments and an extensive parameter tuning process, we concluded that our neural network maximized accuracy with the following parameters: the Adamax optimizer, the sparse_categorical_crossentropy loss function. Additionally, our neural network architecture was optimized with two hidden dense layers using the ReLU activation functions with a dropout layer between the first and second hidden layers, and one output dense layer using the softmax activation function. Moreover, we concluded that our ensemble model maximized accuracy with the following parameters: 100 estimators, "auto" or "log2" max_features, and a "balanced" class_weight.

The incredibly high accuracy rates of our neural network and ensemble model was a huge point of inquiry, but we concluded that the accuracy rates were likely a correct reflection of the strength of our model. We concluded that because there is so much data in the training set so that some dishes can reappear in a pretty similar combination in the test set, therefore increasing the likelihood that our models learn the correct classification. Overall, due to its ability to work with large datasets without loss of accuracy and by virtue of constructing a multitude of decision trees, we conclude that the Random Forest Algorithm is one of the best classification algorithms to accomplish the task of building a recipe classifier.

5 Citations

https://www.kdnuggets.com/2016/10/artificial-intelligence-deep-learning-neural-networks-explai
html
https://towardsdatascience.com/introduction-to-neural-networks-advantages-and-applications-968
https://en.wikipedia.org/wiki/Softmax_function/
https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/
https://www.analyticsvidhya.com/blog/2015/06/tuning-random-forest-model/
https://chrisalbon.com/machine_learning/trees_and_forests/handle_imbalanced_classes_
in_random_forests/
http://www.erogol.com/comparison-sgd-vs-momentum-vs-rmsprop-vs-momentumrmsprop/