What's Cooking?

Harnessing the Power of Machine Learning in the Kitchen

Christopher Henderson
University of Oregon
chenders@uoregon.edu

Cathy Webster
University of Oregon
cwebster@uoregon.edu

Kathryn Lovett
University of Oregon
lovett2@uoregon.edu

***Abstract* — Exploring new recipes found online or through friends can be an enjoyable pastime. However, often cooks will find themselves at a loss as to the type of cuisine they are concocting. Recognizing the dilemmas posed by recipes with lists of ingredients and no specific type of cuisine, we experimented with different machine learning algorithms to develop an accurate cuisine classifier. With twenty different cuisine labels, from Chinese to Italian, we experimented with a decision tree model, a neural network model, and a linear SGD model. Ultimately, we found that the decision tree could not attain high accuracy due to the high number of labels. Utilizing a neural network increased our accuracy to about 74%. We attained our highest accuracy with a linear SGD model at 78%. Creating a highly accurate cuisine classifier proved a difficult task. We obtained a fair level of accuracy, but much more could be explored as far as different methods and algorithms that could improve the accuracy of a cuisine classifier.**

1. INTRODUCTION

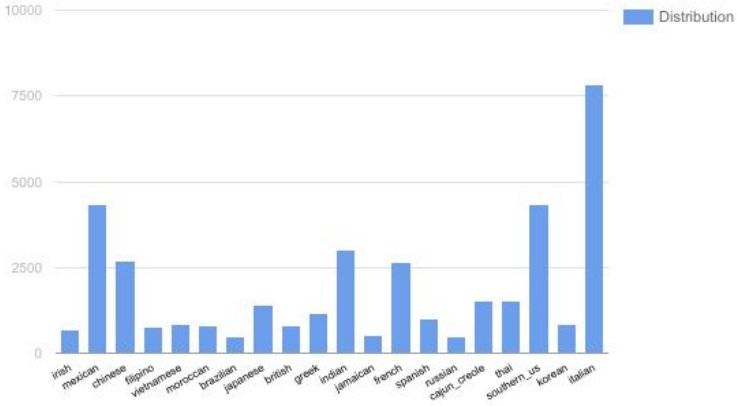
The everyday cook has a dilemma as recipes can be developed from word of mouth or from online resources. The cook will have specific ingredients but not know what category of food they are creating. We used a large data set of recipes that included categorized recipes with specific food ingredients. We will show the type of cuisine being made based on the ingredients that are being used. We will be basing our evaluation in a decision tree and cross referencing its evaluations with a neural network algorithm, as well as a neural network that implemented stochastic gradient descent (SGD). In addition, we will explore a linear SGD model. We gathered our data sets from https://www.kaggle.com/c/whats-cooking and used their guidelines to create our product.

2. BACKGROUND

We are given a dataset with examples and each example is a recipe with a list of ingredients and its type of cuisine. We are also given a second dataset that contains a list of ingredients, but no type of cuisine. Our goal is to use the first dataset to train a model and use that to predict the type of cuisine on the second dataset. There are 20 different types of cuisines listed in the training data, which means that there are 20 different possibilities for the output of each example. Figure 1 below shows the distribution of examples by cuisine. There are over 6,000 different ingredients, or features, listed in the data. Most examples use no more than 10 ingredients, so it's very likely that most of the ingredients are only in a small number of examples or even only in one example. Due to this, our chosen features are boolean, our

outputs are discrete, and the problem is a multi-class classification problem. Given the number of features and possible outputs, we can assume that this data is not linear. Thus, using an algorithm that only works for linearly separable data will not work well with this problem.

Figure 1: Distribution of cuisines in training data



The problem of recipe classification was supplied to us through a previous Kaggle challenge. The training data involved was pre-assembled into a json file of 41,423 entries. A similarly sized test data file was supplied as well. When intaking the data, we converted the examples into a 2D array, where each ingredient or feature to be considered was a binary value of zero or one, depending on whether that example contained that ingredient. We also assembled a list of the possible labels, and added the index for each label to the end of each example. We counted the occurrences of the ingredients or features, and selected the most common ones. To use all of the features would be inefficient. The ultimate goal of our problem is to classify the type of cuisine of a recipe based on its ingredients. To this end, we aimed to create a multiclass classifier and used the most common ingredients as our features, and made them binary values of zero or one based on their presence in a recipe. We used the ID3 and neural network algorithms, as well as linear SGD.

3. METHODS

For ID3, we first used the training data to come up with a model to use on the test data. We used our implementation of id3.py from the first programming assignment. Taking in the training data, we chose an attribute with the maximum gain, split the data based on that attribute, then recursively built the left and right subtrees until all the class labels were the same, there were no more attributes to choose from, or the maximum gain was less than zero. Due to our test set not having any labels, we partitioned off half of our training set as validation data. After training the first half of the data, we ran the validation data against the model to measure accuracy. The accuracy is the number of correct predictions divided by the total number of predictions made. Because our feature set is so large, we sorted the features by how many examples it appears in, then greedily picked the most common features. We started with choosing the top 30 features, 60, 150, etc., and gradually increased the number of features we utilized. The more features there were, the longer it took for the algorithm to run; it would take between 5 to 30 minutes to run.

On top of using the ID3 algorithm, we used *DecisionTreeClassifer* from the Sklearn library. With this, we read in the data file and created one 2d vector for the features and one

vector for the labels where each index *i* corresponds to the the *i*th example. The method requires numpy arrays, so we used *np.array()* to make the vectors into numpy arrays. Sklearn provides a method *train_test_split(x, y)* to split the data *x* and labels *y* array into training and testing sets. Because our test data provided by Kaggle does not have labels for each example, we used some of the training data as a validation set. Similarly with the ID3 above, we ran the algorithm adjusting the number of features. The *DecisionTreeClassifer* easily allows adjustment of the maximum depth of the tree, so we also tuned this parameter to see if it changed the accuracy.

For the neural network, we used the python library Sklearn. More specifically, we used their multi-layer perceptron method that has the ability to learn non-linear models. The Sklearn method for creating the model is *fit(X, y)* which takes in data matrix *X* and target *y* and both *X* and *y* should be numpy arrays. After reading in the data and creating data and target arrays, we used numpy method *np.array* to transform the data and target arrays into numpy array. Sklearn has method *train_test_split(x, y)* that splits the data and target arrays into training and testing data arrays. This is helpful for us because our test set provided by Kaggle does not have any labels.

To start building the neural network, we first declare a *MLPClassifer* and set its parameters. Parameters such as the size of hidden layers, solver for weight optimization, and learning rate can be tuned to best suit our dataset. Initially, we started by using the 1000 most common features for our feature set and set 3 hidden layers with 20 nodes in each to represent each label. We then further experimented by changing the number of hidden layers and other parameters to see how tuning the parameters affected the accuracy on the set. Once *fit(X, y)* finishes, we used *predictions(X_test)* to make predictions on the test set, then calculated accuracy by printing *classification_report(y_test, prediction)*. The algorithm never took more than 25 minutes to run. Increasing the number of hidden layers was directly related to how long it took to run.

Considering the size of this problem, we also used a linear SGD (Stochastic Gradient Descent) classifier, using Sklearn *linear_model.SGDClassifer*. Linear SGD will calculate the gradient of the loss for each sample and the model is updated along the way depending on that loss. Similarly to how we read in the data for the neural network method, we created numpy arrays of data *x* and labels *y*, then separated the data into training and validation sets using *train_test_split(x, y)*.

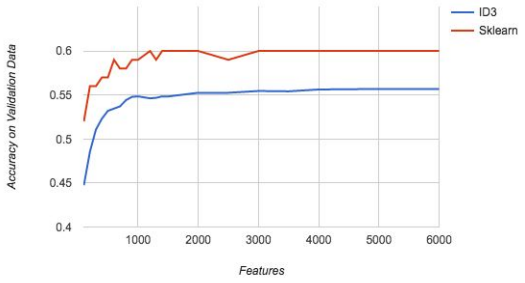
As we were running the algorithm and calculating the accuracy, we adjusted parameters for the loss function, regularization term or penalty, and learning rate. We found that hinge loss and squared hinge loss are the best loss functions to use because it gives us a linear SVM. And since L2 regularization is used for linear SVM, we used L2 for the penalty. As for the learning rate, the default is optimal where eta = 1.0 / (alpha * (t + t0)). We adjusted and trained models using learning rates ranging from 0.1 to 0.0001 to see how it changed our accuracy.

4. EXPERIMENTS

Decision Tree

We initially tested our ID3 algorithm on a small number of examples, about 10. This helped test the functionality of our code. Due to our test set not having any labels, we partitioned the training data: the first three fourths as the training set and last fourth as validation set. Initially, we ran the algorithm by picking the 100 most commonly used ingredients as our feature set. This gave us a very low accuracy of 19%. We increased the features to contain 200 most commonly used ingredients which gave us 45% accuracy. By continually increasing the number of features used, we were slowly increasing our accuracy. However, we found that the increase is exponential. Even if we used over 2000 features, the accuracy was at 54% which is just as good as a coin flip. Figure 2 below shows how the accuracy grew significantly from using 0~1000 features, but as the number of features increases beyond 1000 there is no significant growth.

Figure 2: Accuracy on ID3 and SkLearn.Tree



The negative results were disappointing and we investigated what could have been the reason for it. We modified the algorithm to a binary tree to predict if the recipe is cuisine *A* or not cuisine *A*. First, we chose the labels to be {0, 1} where 1 means the recipe is italian and 0 means it is not. When we ran the algorithm and predicted on the validation data, we got 95% accuracy. We continued to run the algorithm, using different labels to make a binary tree. This led us to cross out the possibility that our ID3 code is incorrect and it had to be other factors. Since the decision tree worked so well with binary labels, we wanted to see how well it performs with a smaller multi-class tree. We modified the algorithm so there are n number of labels. With 3-, 4-, and 5-class trees, the accuracy never fell below 70%. From these results, we learned that the more classes there are, the more difficult it is to achieve high accuracy on a decision tree.

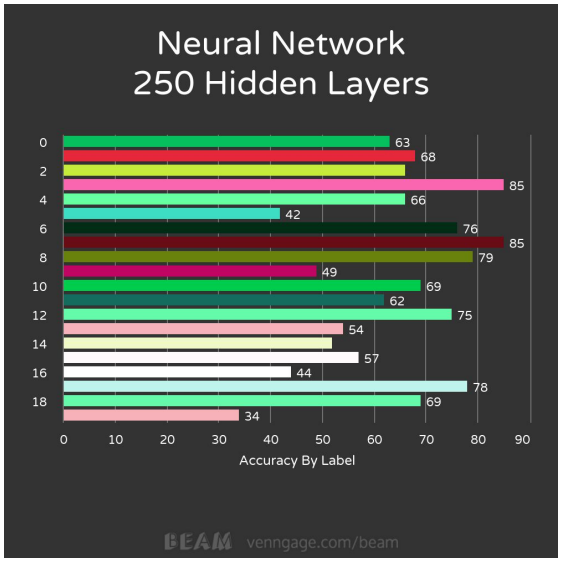
The decision tree algorithm's poor performance can fall back to one other thing: poor feature selection. By greedily picking the most common features, we left out features that are less common but could have had significant impact on splitting data. An example would be features that appear in only one type of cuisine. Even if a feature appears in a few of the examples, it could have only been in examples with one type of cuisine. When it comes down to splitting on that feature, the algorithm would choose that one type of cuisine this feature is only found in. In this case, we didn't get good accuracy because we chose "bad" features to classify the recipes. By choosing "bad" features, we have a lot of noise and it led to the model learning insignificant features. One way to perform better feature selection would be to determine which features would best differentiate each class from the others and keep the number of

features small. Choosing the best features will always be somewhat greedy, but we can make a better feature set by looking for the most unique features in each type of cuisine. A feature that is common in one cuisine can be common in another so it will not help classify an example from one cuisine or the other. However, if we can find unique features that only appear in one type of cuisine, the tree can better classify an example.

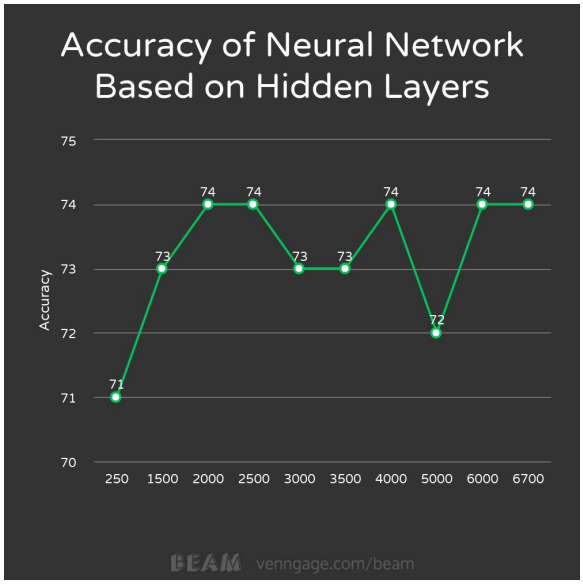
Neural Network

Due to our bad results with the decision tree algorithm, we decided to experiment with the neural network algorithm. Initially, we found example code for a multi-class classifier online and worked to adapt that code to our data. Issues we encountered include the fact that the code was designed for a three-class classifier and we have twenty classes, as well as the fact that it uses numerous numpy functions. After struggling with morphing that code to our needs, Cathy was able to get SciKit working, and we proceeded with its neural network algorithm.

Each occurrence of that Neural Network would give appropriate information about each of the associated categories and how they fared in the following results. The following chart shows that not all categories received the same accuracy as each other. What also needs to be understood about this chart is that there is not an equal amount of examples in each category in both the training and the test data.



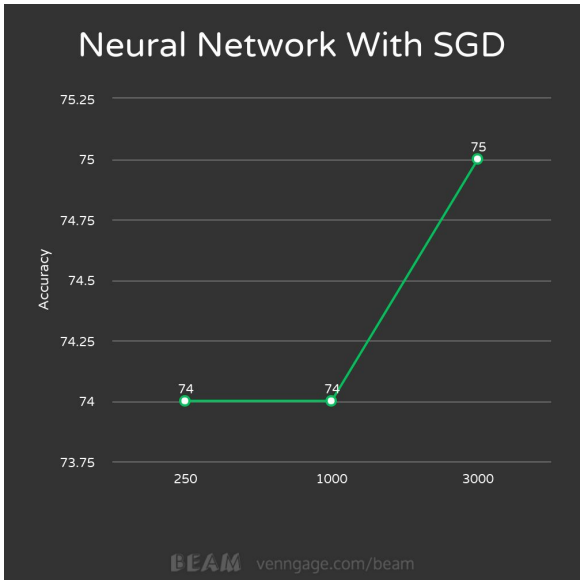
We did testing with different parameters. The previous chart depicts the accuracy by category of the generic neural network algorithm with 250 layers. The next chart shows the average accuracy for the neural network algorithm varying by the number of hidden layers. We attempted to add additional attributes to further optimize our data, but due to time restraints there was only a limited amount of tests that we could perform.



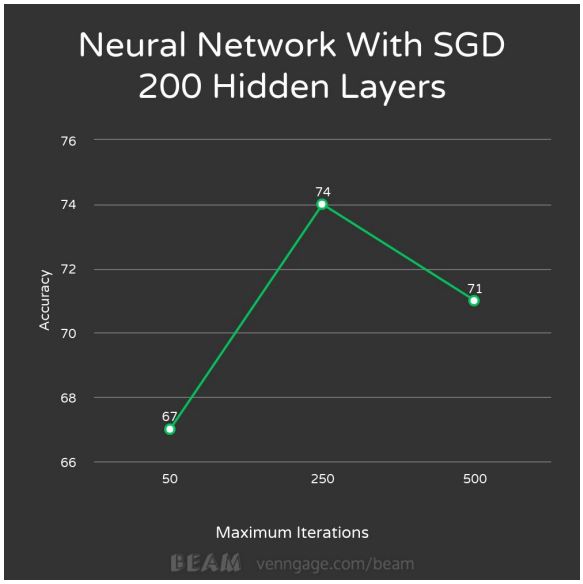
We also added additional optimization of our Neural Network by using a SGD optimizer for the training solutions. The SGD allows for optimization using some of the Sklearn predefined algorithms and it uses the following formula for the SGD adaptive learning:

$$w \leftarrow w - \eta(\alpha \frac{\partial R(w)}{\partial w} + \frac{\partial Loss}{\partial w}) \text{ [1]}$$

We initially attempted to find an appropriate selection for testing additional attributes. The following graph shows three data points of 250, 1000, and 3000 hidden layers. 250 was designated by default but that is only due to the amount of time that is required to run the algorithms. The 3000 layer SGD adaptive learning took approximately 9 hours with our fastest computer. We would have provided additional data points but due to the restraints of the project it was not feasible.



We agreed upon using 250 as the amount of hidden layers and with the remaining time in the project attempted to look at other attributes to further optimize our algorithm. We used the 250 baseline as a guiding point of 74% accuracy. We attempted to optimize the amount of iterations that the programmed performed to see if it had an effect on the accuracy. 250 iterations was the default count that Sklearn is set to. Decreasing the iterations to 50 decreased our accuracy to 67%. Increasing the iteration to 500 also decreased the accuracy to 71%. At this point we decided that the default amount of 250 was appropriate and that it determined to have defaulted there appropriately. This can be seen in the following graph.



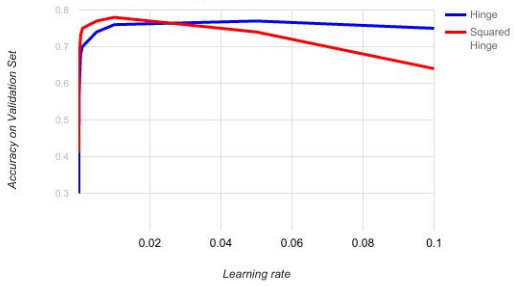
If we had more time, we could have run our training data multiple times and continued our evolution of possible results. But with the time restraint and wanting to do additional evaluations of other formulas we had to finish this section. Additional tests would be on the continued feature set as seen in the Decision Tree and other learning models to name a few.

Linear SGD

Initially, we tested our data using an L2 regularizer and hinge loss function, leaving other parameters to their default and using all of the features in the set. This gave us an accuracy of 76% on the validation set. We changed the loss function to squared loss and found an accuracy of 72%. Since the accuracy between using hinge and squared hinge only has 4% difference, we wanted to see how adjusting other parameters would affect the accuracy using both loss functions. The other two parameters we started tuning were the learning rate and epochs. The classifier for the SGD model will be initialized as *SGDClassifier(loss="hinge", penalty="l2")* and *SGDClassifier(loss="squared_hinge", penalty="l2")* and we will be adjusting additional parameters, *learning_rate, eta0* and *n_iter*.

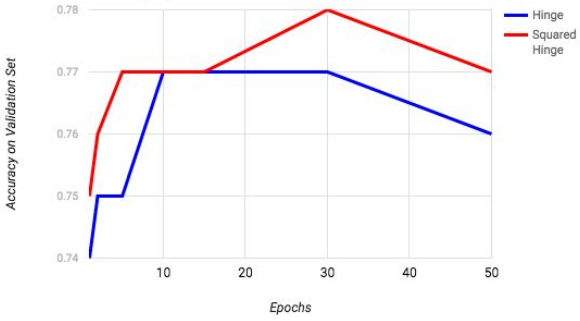
First, we wanted to find the best learning rate. To tune the learning rate, we define the learning rate as *learning_rate ="constant"* and set the *eta0* to a rate we choose. Figure 4 below shows our results from tuning the learning rate. We started with a very small rate at *eta0 = 0.00001* and this gave us a low 30% accuracy for hinge loss and 41% for squared hinge loss. From there we kept increasing the rate until we reached the optimal rate for both loss functions. The optimal learning rate for hinge loss was *eta0=0.05* and for squared hinge loss *eta0 = 0.01* with 77% and 78% accuracy respectively.

Figure 4: Linear SGD Accuracy Based on Tuning Learning Rate



Now using the optimal learning rate, we will tune the epoch. To tune the learning rate, we define the learning rate as *n_iter* to an integer we choose. For the learning rate, we used the optimal rate 0.05 for hinge loss and 0.01 for squared hinge loss. We started with *n_iter = 1* and continued until we found the optimal epoch. Figure 5 below shows our results. The accuracy didn't change much, in that it stayed between 74% ~ 78% and didn't make it any better than not defining the epochs at all. The optimal *n_iter* was 30 for squared hinge loss and between 10~30 for hinge loss.

Figure 5: Linear SGD Accuracy Based on Tuning Epochs



Overall, using the squared hinge loss function performed better than using the hinge loss function. The learning rate at 0.01 and epochs 30 gave the best accuracy at 78%. The squared loss performed better overall because

it penalizes violated margins more strongly than hinge loss does. The hinge loss function penalizes linearly, but squared hinge loss penalizes quadratically. The learning rate was crucial for solving the problem using linear SGD. The learning rate needs to be set to an appropriate value for SGD to work correctly. If the rate is very small, like 0.0001, then the value is too small and will require many iterations to converge. However if the rate is very large like 0.1, then it moves too fast that it might skip the optimal solution. It's very important to choose a learning rate that appropriately controls how much coefficients can change on each update. Determining the learning rate is a trial-and-error; for this particular problem, a learning rate of 0.01 was the best rate that gives the highest accuracy. One issue we faced with tuning the epochs was that Sklearn did not check for convergence or allow early stopping. We will never know how many iterations are enough for convergence. A way to control this is to do early stopping or check for convergence. If you keep training a model even after it has converged, it's possible that the model will overfit. By trial-and-error, we found that 30 epochs gave the highest accuracy for squared hinge loss and if we increased the epochs the accuracy decreased.

5. CONCLUSION

Our group worked together to design, execute, and experiment with our chosen algorithms and data. After selecting our project of cuisine classification, we met to determine which algorithms we wanted to pursue. We initially decided upon a decision tree as we felt that the ingredients data might fit well with that algorithm. After testing that algorithm with our data and getting fairly poor accuracy, we felt that a neural network might work better. We attempted to adapt a neural network algorithm from online, but ran into issues as the code was written for a three-class classifier, not a twenty-class classifier. Cathy got SciKit to work, and then we were able to run their neural network as well as their stochastic gradient descent algorithm on our data, with much better results. Ultimately, we found that, based upon our limited research, predicting a type of cuisine based on its ingredients is a machine learning problem that is best addressed by stochastic gradient descent.

6. REFERENCES

[1] "1.17. Neural network models (supervised)" *SkLearn*, 20 March 2017, http://scikit-learn.org/stable/modules/neural_networks_supervised.html .