Kaggle: What's Cooking? Project 1: CS 145 Fall 2015, UCLA

TEAM NAME: CCTRR

TEAM MEMBERS and their respective contributions:

Raymond Nguyen (UID 203-565-684): Participated in planning initial framework, setup and managed the version control of the project through GitHub.Wrote the vast majority and framework of the ID3 algorithm from scratch. Debugged and tested the ID3 algorithm. Implemented the idea of pruning biased and insignificant ingredients from datasets to substantially improve computation speed at the cost of biased accuracy. Implemented the entire Naive Bayesian Classifier from scratch alone. Wrote the entire algorithm for the Classification Based on Association Algorithm alone.

Chih Chin Chang (UID 304-541-390): Wrote the entire decision tree implementation using Gini Index from scratch. Wrote the entire decision tree implementation using the Scikit-learn libraries. Debugged and tested both of the algorithms

Theodore Nguyen (UID 704-156-701): Participated in planning initial framework, setup the communication platform (Slack), wrote summary documents. Wrote the Entropy() and Info() functions in the algorithm implementation of the ID3 classification algorithm from scratch. Partially assisted in testing and debugging of full ID3. Added the (short) random sample training set implementation of ID3. Wrote the SVM algorithm using the Scikit-learn libraries.

Richard Chung (UID 804-050-445): Assisted with debugging and testing each algorithm to compare results. Wrote the partial first draft, which eventually implements the framework of the entire report.

Introduction and Overview

The “What’s Cooking?” Kaggle project posed a multi-class classification problem. We are given two datasets - train.json and test.json, which are the 'training' and the 'testing' data respectively. Both files are a list of recipes and their respective ingredients; train.json also contains the type of cuisine for each recipe. Given this, the task at hand was to classify each of the recipes in the testing set into different cuisine types. To do this, we build a model using the training data that, when inputting a list of ingredients of a recipe into the model, will output the cuisine type of the recipe This report will outline and log our steps to solve this problem, including - but not limited to - choosing algorithms, problems in implementation, rationales of accuracies, and most importantly, distinct comparisons of algorithms.

Implementation thoughts and analysis, slight overview of report and of different algorithms

The first thought was that this problem was obviously a classification problem. The classes in the problem are the cuisine types; the entries in the database are each individual recipe; the attributes that we work over are each individual unique ingredient in the set of all ingredients that occur in the all the recipes - these attributes take on the values 0 and 1, depending on whether the attribute (ingredient) is present in the recipe or not.

Quickly following, we deduced what was a good algorithm to pick to solve this problem. The first thought was to see if any of the algorithm we learned in class were a good fit: Classification via Decision trees, Bayesian Classification, Neural Networks, Support Vector Machines, Classification Based on Association.

We concluded that each individual attribute would not necessarily be independent (take a recipe of peanut butter and jelly for example - the two almost frequently occur together), so Bayesian was lower on the list. Classification Based on Association created CARs in order to classify an entry; what if the recipe consisted of ingredients that were not present in any of the CARs? The algorithm would classify it in the default case. We thought (primitively) that this was not an accurate representation since it does not fully use data not present in CARs, so this was marked off as well. The class notes on Neural Networks were sparse, and SVMs seemed a little too complex to implement for our liking. Therefore, the most likely conclusion was to write a decision tree algorithm, especially since the latter has multiple methods of attribute selection, in case one method didn't pan out. The first algorithms we implemented, thus, were decision trees ID3, and then with Gini Index measure.

Ironically, these initial decision tree implementations ended up being the least accurate despite our rationalizations. We chose to program in Python and eventually used the Scikit-learn libraries found at http://scikit-learn.org/stable/ along with standard Python libraries.

Algorithms:

Decision Trees Overview

In approaching this classification problem, we selected the model of classification via decision trees for several reasons. Empirically, the classification model of decision trees was thoroughly explored during lecture, making it easier to implement given that the related algorithms and characteristics (i.e. information gain, entropy, etc.) were already familiar. As mentioned before, decision trees are also characterized by several attribute selection measures, from the attribute of information gain to Gini index. This variety in selection metrics hence allows us more flexibility in selecting which attribute would give us more accurate results. Overall, the simplicity, comprehensibility, and plausibility of implementing classification via decision trees are reasons that motivated our selection.

However, in selecting decision trees as our classifier, we must also address all the potential shortcomings that are classically associated with this method. Given the size of the data set provided by Kaggle, the induced tree might become too large and create too many branches, leading to the issue of model overfitting. That is, once the tree becomes too large, its test error rate begins to increase even though its training error rate decreases.

Despite this main fault, there are several ways to approach the issue of overfitting. First, we may use the approach of pre-pruning, which halts the tree-growing algorithm before generating a fully grown tree that perfectly fits the entire training data. This is governed by the condition that we do not split a node if the goodness measure or observed gain in impurity measure falls below a certain specified threshold. However, there is difficulty is selecting the correct threshold as too high of a threshold would result in an under-fitted model while too small of a threshold may not address the original issue of overfitting at all. Second, we may use the approach of post-pruning, which handles the decision tree after it is grown to its maximum size. In this step, the decision tree is pruned in a bottom-up fashion in which selected subtrees are replaced with either a new leaf node whose label is determined from the majority class associated with the subtree or the most frequently used branch of the subtree.

Decision Trees - ID3 algorithm implementation

Maps/dictionaries as data structures

Initially we thought that we should focus on the data structures backing the data in the algorithm. For that we decided to represent the ingredients in recipes as binary. Thus for every recipe you would get a long list of 0's and 1's. However this approach showed that there would be a vast amount of 0's as you would need to represent each of the 6000+ ingredients in the data set for just one recipe. Thus we decided to convert the sparsely populated arrays and use the dictionary data structure also know as maps. This would greatly reduce the amount of space each

recipe would occupy while still giving us O(1) random access time. The only problem is that we would need to store a separate list of all ingredients to supplement it.

Recursion Depth Limit

Probably the greatest obstacle we faced was the recursion depth limit. Though our algorithm worked perfectly for small data sets as we tested against large datasets with recipes towards in the thousands our Python interpreter would crash due to the maximum recursion depth that is self enforced. The depth limit is in place to protect against stack overflow. Though you can bend this limitation by manually increasing it through the system it is dangerous if you do not know your own machine's architecture. To be sure, we ran our decision tree classifier on our own personal computers as well as on SEASnet and UCLA servers. However, these running scripts were often killed on these servers, our assumptions being that they were inundating the servers and using too much computational power.

Training data sampling

As a result of the recursion depth limit in Python, we could only execute the ID3 algorithm for up to about 20,000 lines of a snippet of a train.json. An idea we had was to randomly sample out of the 39,774 recipes in train.json a maximum number of recipes (I found this number to be in the neighborhood of 1300 by trial and error) to build the model without exceeding the recursion depth limit. Ideally, this way the snippet of the training set could more accurately represent the population of the original training set. However, this method did not produce any noticeable difference from just training off the first 20,000.

Pruning, unique recipe identifiers

After some time we realized that our decision trees were heavily skewed in one direction. After some delving the problem was due to ingredients that uniquely identified recipes, or in other words was only present in a single recipe. Our algorithm would pick up on this unique identifiers and choose them since they gave an entropy measure of zero. In order to stem this we decided to prune the data before passing it to our algorithm. "How much should we prune though?" was a question we had to ask. After analyzing the we found that these unique ingredients were far more prevalent and composed 26.2% of all ingredients. In fact a deeper look at the dataset revealed that ingredients that occurred 4 times or less comprised of 50.3% of the total ingredients. This was the cause of our extremely skewed tree. See Table 2 at the end for further details.

The maximum accuracy we obtained through our ID3 algorithm made from scratch was 48%, operated on the first 80,000 or so lines of train.json after raising the recursion limit 4-fold. This was not that much different from the accuracy given from 20,000 or so lines, so we can see that the ID3 algorithm does not go too far past the neighborhood of 40%-50% accuracy for our implementation.

Decision trees - Gini index

The effectiveness from the Gini indexing approach is derived by a complicated mathematical idea of comparing the equality curve and the Lorenz curve. The equality curve, in the perspective of a classification decision tree, is one projected by an attribute that simply does not offer much information gain. It is essentially the default attribute that doesn't much in favor of deciding the outcome in any classes. The Lorenz curve can be intuitively thought of as the curve that illustrates a factor of deciding a classification. In other words, the Lorenz curve shows that a particular attribute can offer some information gain towards the final decision. And by finding the difference between the equality curve and the Lorenz curve, the idea of information gain is materialized and can be maximized.

The reason behind the Gini index's success over other decision tree splitting algorithms is that instead of just using the entropy, or how much a feature can "affect" the current partition, Gini index combines both this idea and the idea of accuracy in terms of probability a confirmation of a feature or disconfirmation of a feature would affect each classification. This probability is also seen as the accuracy.

Implementation:

The single biggest hurdle to implementing the Gini Index from scratch is its recursive nature. And in an run-time protected environment like the Python interpreter, having a highly recursive algorithm running over a large dataset would become problematic. This is the reason why the implementation created from scratch cannot run on large datasets. Each recursive partition would add another set of partition on the top of the stack of many partitions as the tree is built along a path. One of the improvements scikit learn library has over writing from scratch is its use of multi-threaded programming. However, a common problem with such parallelism is race conditions. And this has affected the outcome somewhat, because the csv writer would often skip a line or a list may sometimes miss a few lines.

But without considering such limitations, the procedure to building the Gini index resembles building an ID3 decision tree quite closely. From a high level, the procedure begins when a partition is concerned and an attribute is given as an input for partition. Check all the classifications in the partition, and if the entire classification is the same, then a leaf node is discovered and add the class leaf to the tree. This is the base case. Otherwise, calculate the Gini index, which is given by:

$$Gini(ingredient = yes||ingredient = no) = \frac{Y}{T_p}\left(1 - \sum_{i=1}^{n_Y}\left(\frac{y_i}{Y}\right)^2\right) + \frac{N}{T_p}\left(1 - \sum_{i=1}^{n_N}\left(\frac{n_i}{N}\right)^2\right)$$

For Y = yes counts, N = no counts, Tp = partition total, Yi = yes counts in cuisine I, Ni = no counts in cuisine i

To accomplish this calculation, several helper functions are defined. There are two partition functions that takes a partition and an ingredient, and returns a partition of recipes that do include the ingredient and another that do not respectively. This function is how the recursion travels down the path of containing and not containing an ingredient after a splitting ingredient is determined. There are two functions to return a list of unique cuisines and ingredients of a partition. These lists are important to loop the number of unique cuisines and count their totals. A tree builder function is responsible for calling the helper functions and the gini index to determine the split and the next traversal. This is highly recursive and because of the many function calls and local partitions, this function is extremely costly.

The tree class is extremely simple. It is simply a single node. A node is defined as an item, which can be either an ingredient for the inner nodes or a cuisine for a leaf, and two pointers, yes and no. Yes pointer leads to traversing along the path including the splitting ingredient, and no the other. In this manner, a traversal can be easily maintained by the recursion itself instead of depending on an interpreter, which would've improved the performance, but also added to the complexity.

After the tree building comes the traversal of the test data. This procedure simply follows along the path of the tree until a leaf is reached, and return the leaf. In this way, a recursive traversal is followed to complete the test.

Evaluation:

However, because of the complex layering of these recursive function calls that are expensive both in terms of heap and stack memory (heap for local partitions and the stack for pointers and function calls in Python), a practical run through of this algorithm in a single threaded manner using hundreds of thousands of lines of data per stack is not possible. Fortunately, we have devised a way to sample the large dataset. And from these samples, the accuracy produced by this method turned out to be around 48% from a tiny ten thousand line training sample.

To more accurately predict the test data, we have used SciKit Learn library to train and predict from its own decision tree implementation, which is defaulted to the Gini index method. And from this method, the accuracy returned from training on the full training data was at 62%.

Bayesian Classification

Despite the simplicity of construction, low computational costs, and high efficiency of the naïve Bayes classifier, we initially ruled it out because we must first apply the simplified assumption that all attributes and data are class-conditionally independent. Given that the problem deals with the association of ingredients and recipes, there exist correlations between ingredients that cannot be ignored by assuming class conditional independence. There exist

dependencies that must be modeled and reflected in order for the resulting data to be accurate and sound.

Even though we initial waved it off, our troubles parsing through the large datasets with our limited hardware we had little choice but to explore all options, including the simple NBC. We still decided to stick with the dictionary/map representation of our data in order to save on space. Its runtime was extremely efficient and very easy to implement. However we were still worried because of the assumption of class conditional independence.

The algorithm was written without any help from libraries, and ran much faster than all of the decision tree algorithms. In the end, it gave us the second highest accuracy at 67%.

Bayesian belief networks

Bayesian belief networks deal with the issue of dependency found in Bayesian Classification by allowing class conditional independencies to be defined between subsets of variables. Although this is a sound choice if we wanted to utilize Bayes' Theorem and conditional probability, we decided against implementing a Bayesian network because of our limited experience with it in class.

Classification Based on Association (CBA)

As an iterative approach to frequent itemset mining, classification based on association makes multiple passes over the given data and the attributes to generate frequent itemsets and rules which are then used to build a classifier. These resulting CARs are then ordered by support, confidence, and order of generation to construct the classifier.

The Classification Based on Association seemed like a good compromise between the NBC and Decision Tree Learning in terms of complexity and accuracy. Though we expected this algorithm to work the best, we ran into issues in determining what confidence and support minimums we should use. Though the algorithm favors confidence over support we realized to exclude the unique recipe identifiers we would have to increase the support as that acts as our pruning. Determining the confidence was another balancing act as if you expect that higher confidence leads to better rules, however this causes other problems as the higher the confidence the worse at covering all the recipes became due to the pruning of all the potential rules. In the end, due to time constraints and lack of a methodology for the case, we were not able to come up to a good balance between minimum support and minimum confidence. The execution we performed had 56% accuracy for operating on the entire dataset with min_conf = 0.60 and min_supp = (5/39774) = 0.000125.

Support Vector Machines (SVM's)

The use of Support Vector Machines is an effective classifier because it avoid many of the issues presented by decision trees and Bayesian classification. In using support vectors, SVM's are much less prone to overfitting than decision trees and most models by modifying its C parameter to maximize the margin and fitting the training data. And unlike Bayesian classification, which can be prone to inaccurate data due to its naïve assumption of class-conditional independence, SVM's are generally extremely accurate due to its ability to model complex nonlinear decision boundaries.

Our implementation of SVMs was the last algorithm we implemented - thankfully, it was also the most accurate. This implementation was the algorithm that relied most extensively on the Scikit-learn libraries; the libraries already have a built in SVM function that simply requires an attribute array and a class array as input. Other than the nuances I ran into parsing data with python, I simply had to have one input array be the size of the number of recipes, where each element of the array is the cuisine type of that recipe, and another array be the array of ingredients, where each index of the array is an array telling us if that specific recipe has certain ingredients.The latter follows the "Bag of Words" model; a unique ingredient list was initialized to count every single unique ingredient. Each index of the unique ingredient array corresponds to the ones inside each of those arrays following the "Bag of Words" model, where the value is 1 if the ingredient is present in the recipe, and 0 otherwise. Inputting these two arrays into the .fit() function would automatically create a model for the training data. We simply call predict with the test set afterwards to get our result, so the libraries do most of the work.

The algorithm yielded the highest accuracy of our algorithms at 77% accuracy.

Conclusion

The ID3 algorithm experienced serious overfitting problems due to individual ingredients uniquely defining a recipe. This same problem is relevant in all variants of ID3, even when some of these ingredients are pruned – a cause for the lack of accuracy. The main problem with the Gini Index implementation was that it required too many recursive calls to even achieve an accuracy. The Naïve Bayesian Classifier managed to get a higher accuracy despite possible dependencies perhaps due to the size of the dataset dwarfing the already minimal dependencies. Classification Based Association had a remarkably high accuracy relative to how far we completed it – we have no idea what optimum parameters it should have, and it did substantially better than Decision Trees and almost as good as NBC. SVMs are one of the most accurate algorithms given an appropriate kernel to always allow linear separation. The only downside was its training efficiency, where here the dataset is not large enough to warrant a problem; so if it runs, it will generally give great accuracy, as in our case.

Table 1: Details on attached files, summary statistics on algorithms

File name	Description	Algorithm Accuracy
id3.py	Decision Trees with ID3 algorithm	48%
randomID3.py	Same as id3.py, but includes random sampling	47%
gini.py	Decision trees with Gini Index Algorithm	N/A
scikit.py	Decision trees algorithm using Scikit-learn library	61%
nbc.py	Naïve Bayesian Classifier Algorithm	67%
cba.py	Classification Based Association Algorithm	56%
sklearn-svm.py	SVM algorithm using scikit-learn library	**77%**
svm.csv	classification output of SVM algorithm, highest accuracy	Not an Algorithm

Table 2: Shows statistics of ingredients, evidence for rationale of pruning and skewed decision trees

x	Number of ingredients appearing in only x recipes	Percent of total ingredients appearing in only x recipes
1	**1759**	**26.20%**
2	**796**	**11.86%**
3	**484**	**7.21%**
4	**338**	**5.03%**
5	**275**	**4.10%**
6	221	3.29%
7	172	2.56%
8	133	1.98%
9	136	2.03%
10	128	1.91%
11	93	1.39%
12	82	1.22%
13	65	0.97%
14	70	1.04%
15	62	0.92%
16	55	0.82%
17	54	0.80%
18	59	0.88%
19	43	0.64%
20	43	0.64%
21	62	0.92%
22	43	0.64%
23	30	0.45%
24	24	0.36%
25	27	0.40%