

Project 1 Report

What's Cooking?

Team Flying Dutchman

Ruiming Cao (104296587)

- Feature Selection and CARs

Quanjie Geng (204160668)

- Data Preprocessing, Feature Selection, and Logistic Regression

Zhao Yang (904 169 203)

- Data Preprocessing, Performance, SVM, and Ensemble

Weijia Yu (204202814)

- Decision Tree and Naive Bayesian Classifier

Introduction

The goal of this project was to predict the cuisine category of some dish given its list of ingredients. Using RBF kernel based support vector machines, we achieved 79.435% accuracy on test data and ranked 136th out of 1027 contest participants on Kaggle as of Dec. 4th, 2015.

This report analyzes the various modeling methods we employed and presents our thought process in identifying and tackling some of the major challenges. This report also includes sections for data preprocessing and future suggestions for improvements.

Performance

Before delving into the details of the algorithms, we'd like to present a graph that summarizes our results.

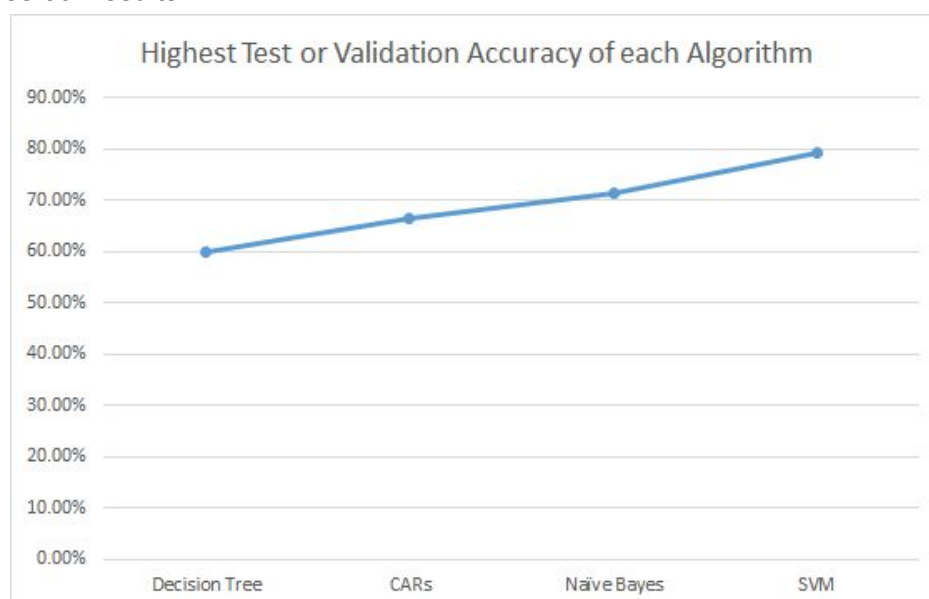


Fig.1. The percentage is the highest of 5-fold validation accuracy or test accuracy of each algorithm. Due to memory constraint, although logistic regression was applied, we did not obtain an accuracy.

Throughout our models, we applied 5-fold cross validation to evaluate the performance of our algorithms before the submission to Kaggle. This step is crucial because we are allowed only a maximum of 5 submissions daily on Kaggle.

We mainly applied cross validation to ensure our model does not overfit the data. If we do not use multiple validation sets to test our algorithm, our model may match too exactly the training data and cannot generalize well to the test data. Our good validation results ensure our algorithms do not overfit or underfit.

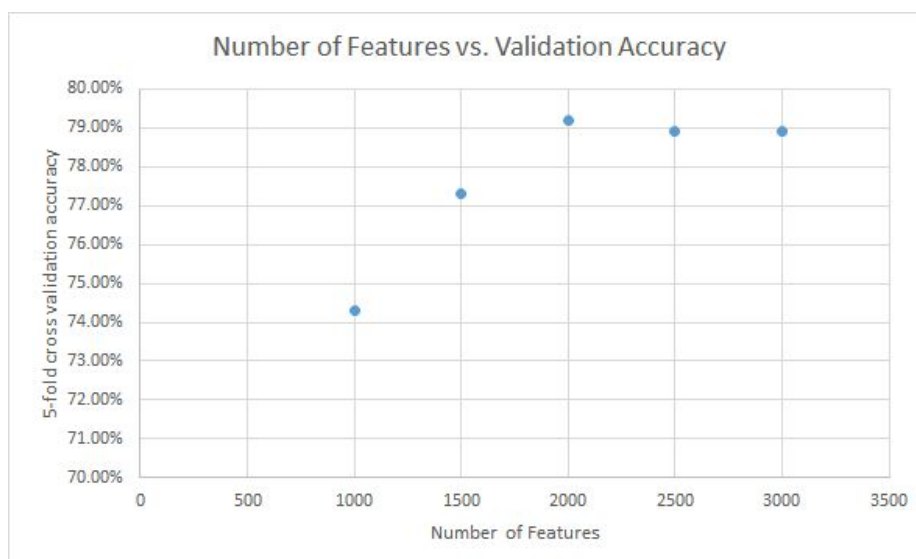


Fig. 2. For SVM, the cross validation accuracy first increases and then decreases as the number of features increases.

We selectively increased the number of features when tuning the SVM algorithm. In Fig. 2., we observe that the validation accuracy peaks at 2000 features, and then starts to decline as the number of features grow. Because our features are frequency based, this may suggest that the first 2000 most frequent ingredients capture the most helpful information for the classification of cuisines. The less frequent ingredients can add noise to our prediction models.

Data Preprocessing

One of the major challenges we saw in this project was the way data was presented. All of us were unfamiliar with data stored in JSON format and were used to data stored in CSV format, which allowed easy access to statistics of rows and columns. Thus, the problem naturally became how to convert the given JSON files into CSV files.

JSON had its advantages: it stored data in a very compact way, keeping the data file small; it was very structured in that it stored data in a way that very much resembled arrays in programming languages. In “What’s Cooking?”’s data set, each recipe was encoded as an object having three fields:

- 1) id, which represented an unique identifier,
- 2) cuisine, which represented the cuisine that the recipe belonged to,
- and 3) ingredients, which was an array of ingredient name strings.

Each recipe object was an element of the JSON array, so we essentially had an array of objects. Due to this structure of the JSON data file, we needed a programming language that supported easy and efficient manipulations of arrays for data preprocessing. Therefore, we chose python, whose list was a winner for this problem.

In order to convert our data stored in JSON format into CSV format, we needed to consider ways of transforming our data such that the statistics we could calculate from rows and columns would actually be useful. One direct approach would be having each recipe as

a row entry, with the ingredients spanning horizontally. But this approach would not give us meaningful columns except for the id field and cuisine field. Upon understanding the bag-of-words model suggested in the project spec, we decided that we would transform the data entries into n-vectors. The idea was that we would first form a set of all possible ingredients, call it features (n features). These features, plus the id and cuisine fields would constitute the header of our CSV file. For each recipe, we would check if the recipe contains the ingredient of the column. If yes, the column would be marked as “1”, and “0” otherwise. (Because there were no duplicates in the ingredients of a recipe, our n-vectors would only contain 1’s and 0’s.)

However, this approach had one severe drawback. The size of the CSV file was exponentially larger than the JSON file. The reason was clear to see. In JSON format, each recipe would have about ten ingredients, but in CSV format (n-vectors), each recipe would have fields for all possible ingredients (roughly 6000+), with only about ten 1’s and the rest 0’s. Since processing big files were very challenging in terms of both time and space, and we actually did not need that many features to create good models (also to avoid overfitting our models), we chose to do feature selection to reduce the number of features to a reasonable size and still be able to make sound predictions.

One way of selecting features was calculating the tf-idf value of each feature. This “tf-idf” value represents the relative importance of a feature. The term “tf” stands for “term frequency”, which in our case is the number of occurrence of an ingredient in a recipe. The term “idf” stands for “inverse document frequency”, which is a measure of how much information the feature provides (an ingredient that appears in most recipes would provide low information; an ingredient that appears in only one recipe would provide high information). In other words, “tf” rewards a frequent feature, and “idf” penalizes a feature for being too frequent. In our case, the maximum number of occurrence of ingredient in a recipe is one, so our “tf” is actually a “boolean frequency”. Therefore, the calculation of “tf-idf” can be roughly reduced to finding the frequency of a feature in the whole dataset and sorting features in frequency descending order. We then extracted the top k features for building our models. (The number k depends on the specific modeling method and we have tested various k’s for better accuracies.)

From observations, we noticed that the ingredients were actually not classified according to their contents. For instance, “tomato sauce” and “15 oz. tomato sauce” were put as too different ingredients in the raw data. It was obvious that these two features were referring to the same thing and thus should be treated indistinguishably. In order to map features like “15 oz. tomato sauce” into “tomato sauce” we tested for substrings. Say if for each recipe we extracted the top 300 frequent features, we then proceeded to check if any of the frequent features marked “0” was a substring of the rest of the 6000+ features and marked the field “1” if that was the case. In this way, we still made use of some of the leftover features when building our models.

The data preprocessing was implemented as a python script, parser.py. To generate test and train files, we run generate_train_and_test_1.py, which essentially runs parser.py.

Methods

Rule-based Classification

Rule-based classifier comes very natural when we try to build a multi-class classification model. The principle of rule-based classifier is very straightforward. We can suppose every type of cuisine may have some ingredients or sets of ingredients that can be used as rules to identify it. For example, “wasabi” is a unique ingredient that only appears in Japanese dishes, and the combination of “curry”, “shrimp” and “lime” are mostly likely to be in Thai food. Before we start to train the classifier, there are several things that need to be mentioned in the original train file.

First, some ingredients, actually point to the same thing, despite the fact that they have different names. “all-purpose flour” and “flour” are exactly the same ingredient. However, if we use bag of word model to extract feature, they will be counted as two features.

Second, there exist two different ingredients that have very close meanings. In most cases, one is the superset of the other. In some way, we can consider them as the same object. For instance, “diced tomatoes” are the subset of “tomatoes”. “nonfat milk”, “low-fat milk” and “whole milk” are all in “milk”. When we train rule-based classification model, it would be hard to associate them during the training process. So, it would be beneficial if we can build a relationship before we train.

Third, there might also be a situation that two ingredients are completely different things in cooking, even though they are very close in their name. “Shaoxing wine” and “white wine” might be a good example. “Shaoxing wine” is a commonly used ingredient in Chinese kitchen, and can rarely be found in other places, while white wine is widely used in western countries.

In short, we will map those similar or identical ingredients into one superset ingredient. In this way, we will need to figure out which part of an ingredient is important, and can represent its true identity, and which part cannot. There comes a big challenge in feature selection. On one hand, if the relationship map we build is too strong, we may lose some information (if we map “Shaoxing wine” to “wine”). On the other hand, if the relationship map is too loose, many features, which are supposed to be combined together, are sparse in space, and we may not be able to find some important rules as we probably can do with strong, combined features.

To build a feature mapping, we use a method to eliminate those frequent adjectives in ingredient names, since adjective words have a large possibility to carry a trivial information (as “fresh tomatoes” makes no difference with “tomatoes”). First, we analyze every ingredient in our training set, and find all words that are used as adjectives using Natural Language Toolkit in Python. Also, many words end with “-ed” are used as adjectives as well, and we should add them on our adjective list. Then, as there are some nouns that are used as adjective form in our adjective list, we should not eliminate them since they have greater chance to have a real meaning in ingredient names. We use a pre-built noun list to search and remove any nouns in our adjective list. Finally, we go back to our training data to count the appearance of each adjective word in our list. In order to reserve more information, we remove those words that only appear few times in training data. So we can be sure that the adjective list only contains those commonly shared adjectives in all types of cuisines and does not contain those words that uniquely appear in a specific type of food (e.g. the word “Jamaican” only appears in ingredients for Jamaican cuisine).

We build a rule-based classifier using PART method, which will generate a decision list using separate-and-conquer. PART method builds a partial C4.5 decision tree in each iteration and makes the leaf with the largest coverage into a rule every time.

To prevent our model from overfitting to train data, we set the minimum number of instances per leaf when we construct tree. Moreover, we also use Reduced Error Pruning method to prevent overfitting. This method will split dataset into a training set and a testing set. When a new rule is built, it will remove one of the conjuncts in the rule, and compare error rate on the testing set before and after pruning. If error rate increases after pruning the conjunct, remove the conjunct.

The actual testing result of rule-based classifier is 0.63, which is much lower than our expectation. We find there are certain drawbacks in rule-based classifier. First, since our method is heavily based on decision tree, our training dataset is expected to be well-balanced. However, in our actual training data, as shown in the Figure 3 below, Italian and Mexican cuisines far outnumber than any other type of food, while Brazilian cuisines only take less than 2 percent of training data. Second, rule-based classifier does not have good time efficiency and memory efficiency. We can only train it using limited number of frequent ingredients. It does not have scalability as good as SVM. Third, as we mentioned above, rule-based classifier requires a proper mapping of ingredients in order to provide both strong common features between dishes in the same class, and a clear gap between different classes.

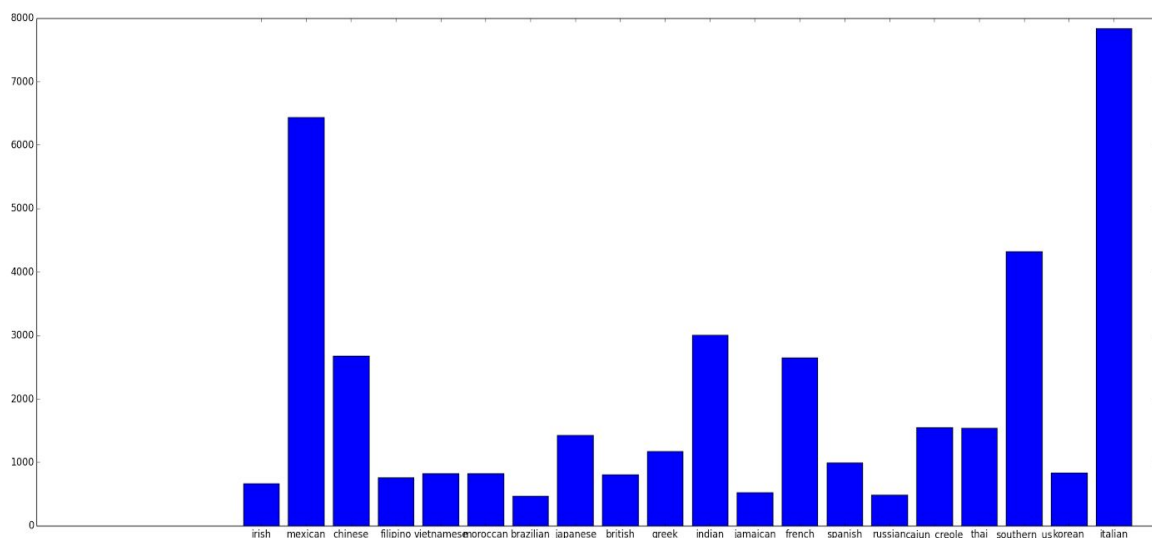


Fig. 3. Number of entries for each type of cuisine in training data.

Support Vector Machine

In our project, support vector machine (SVM) has achieved the highest test accuracy on Kaggle, 79.435%. We'd like to provide a brief introduction to the underlying rationale of this algorithm and explain why it has been relatively successful at predicting the cuisines.

Intuitively, SVM aims to find the hyperplane that best separates the data points in high-dimensional spaces. It achieves so by maximizing the functional margin (distance from

the closest data point to the decision boundary) of the decision boundary, while allowing misclassified data points within this margin. It also uses the method of linear classification to efficiently generate nonlinear decision boundaries by employing kernel functions, which is essential to the success of classifying nonlinear data points such as in our project.

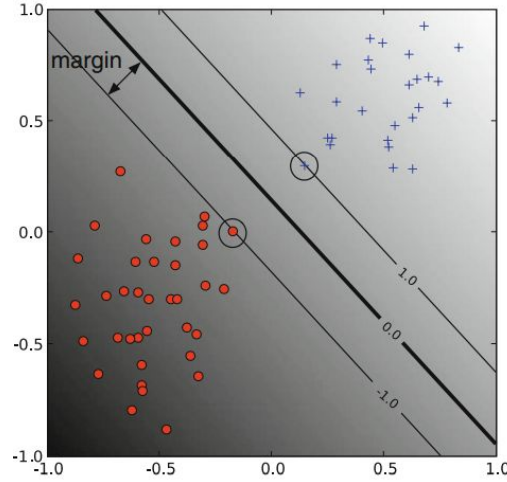


Fig. 4. An illustration of a linear SVM in two-dimensional space. The circled two data points are support vectors. They define the best decision boundary for separating the two classes because the boundary is as far as possible from each point and the classification error is within the acceptable range, in this case, 0. [1]

The decision boundary in Fig. x. is defined by: $w^T x + b = 0$.

The mathematical optimization problem to find the boundary in Fig. x. in its primal form is the following:

$$\min_{w,b} \quad \frac{1}{2} \|w\|_2^2$$

$$\text{s.t. } y_n [w^T \phi(x_n) + b] \geq 1, \quad \forall n$$

, where we assume $y = 1$ or -1 , and ϕ is the mapping function of x to another space. However, this formulation does not allow any misclassification as when the data is not linearly separable, $y_n [w^T \phi(x_n) + b] < 0$, there is no feasible w to satisfy the constraint.

Therefore, the soft-margin formulation is introduced, as shown below.

$$\min_{w,b,\xi} \quad \frac{1}{2} \|w\|_2^2 + C \sum_n \xi_n$$

$$\text{s.t. } y_n [w^T \phi(x_n) + b] \geq 1 - \xi_n, \quad \forall n$$

$$\xi_n \geq 0, \quad \forall n$$

, where ξ_n is called the “slack variable”,

which enables us to increase the margin at the cost of classification errors in support vectors, and C is a really important parameter that regulates the tradeoff between maximizing the functional margin and minimizing classification error among support vectors.

During the actual implementation of SVM, soft-margin formulation is used. Moreover, due to the simplicity to optimize its dual form, we convert the primal form into its dual form by using Lagrangian Multiplier.

$$\max_{\alpha} \quad g(\{\alpha_n\}, \{\lambda_n\}) = \sum_n \alpha_n - \frac{1}{2} \sum_{m,n} y_m y_n \alpha_m \alpha_n k(x_m, x_n)$$

$$\text{s.t.} \quad \alpha_n \geq 0, \quad \forall n$$

$$\sum_n \alpha_n y_n = 0$$

$$C - \lambda_n - \alpha_n = 0, \quad \forall n$$

$$\lambda_n \geq 0, \quad \forall n$$

, where α_n is the

coefficient of the n th example in the linear combination of \mathbf{w} (we assume \mathbf{w} is a linear combination of the input features), and $k(x_m, x_n)$ is a kernel function. Using a general-purpose quadratic optimizer, we can solve the above problem.

Furthermore, it turns out that the optimization problem only depends on the inner product of features, therefore, we can replace the inner products with kernel functions to generate nonlinear decision boundary. In our project, we applied polynomial kernel and Gaussian radial basis function (RBF) kernel and tuned the parameters of each model using cross validation and grid search.

Because the feature optimization process is in parallel with model testing, the following accuracies were collected before the last round of feature optimization and therefore is one-notch lower than our final test result on Kaggle. Nonetheless, it clearly demonstrates our methodology on tuning the hyperparameters for SVM.

The 5-fold cross validation accuracies when tuning the RBF kernel using 2000 features:


				
	4^{-3}	4^{-2}	4^{-1}	
6	72.5	74.7	77.7	
8	74.4	78.9	78.9	
10	78.8	79.0	78.9	
12	78.9	79.0	78.2	

Fig. 5. Tuning parameters of the RBF kernel and the corresponding accuracies.

The 5-fold cross validation accuracies when tuning the polynomial kernel using 2000 features:

				
	1	2	3	
4^{-4}	61.1	61.7	61.7	
4^{-3}	61.7	70.1	74.2	
4^{-2}	62.5	70.9	74.2	
4^{-1}	62.5	71.2	71.0	

Fig. 6. Tuning parameters of polynomial kernel and the corresponding accuracies.

In all of our experiments, we used 5-fold cross-validation to prevent overfitting. From the tables above we can see that RBF kernel achieved the best results on our data, which is consistent with the general belief that RBF kernels tends to achieve better accuracy than polynomial kernels.

Furthermore, we can observe a pattern from the RBF table, which is the color-coded diagonals in the table. We find that the values of Gamma and C complement each other in the sense that as one increase and the other decreases or vice versa, the accuracy tends to remain on the same level, as observed on the diagonals of similar accuracies. Such results are confirmed by other studies we found online. It is commonly found that there is a certain range of C and Gamma which form a band that yields equally good accuracies.

Gamma is the inverse-width parameter of the RBF kernel. The larger it is, the smaller the influence area of one data point is. Therefore, when Gamma is large, the features for one data point are refined to describe only one data point, which leads to increased variance and overfitting. On the contrary, the smaller the Gamma is, the smoother the decision boundary is, because each data point contributes its influence to a larger range of feature values.

In addition, C is an offset between training error and complexity. A small C allows more training error, therefore results in smoother boundary and fewer support vectors. A large C heavily penalizes training error and aims to fit more precisely to training data, therefore uses more support vectors and increases variance.

The more variance we allow in our prediction model, the greater the risk of overfitting. On the contrary, the more bias we have in our predicting model, the greater the risk of underfitting. Therefore, in almost all predicting models, we strive for a balance between bias and variance. This issue is so important and prominent that it is an independent topic of study in machine learning, which is the bias/variance tradeoff. To summarize previous discussions, in RBF kernel SVM, as C parameter increases, variance increases and bias decreases; as Gamma increases, variance increases and bias decreases.

Why then, do Gamma and C appear to have contrary effects on accuracy? Further research explained this phenomenon. At some large value of Gamma, due to overfitting, the curvature of the decision boundary starts to disappear. Normally the curvature would increase due to overfitting, but in this case the curvature increases so much that it starts to wrap around the individual points and destroys the overall curvature. At this point, a large C will “restore” the curvature by penalizing margin error of the support vectors. In summary, while large C and large Gamma both increase variance, there is a range during which, due to the different working mechanism of each parameter, their individual effects offset each other.

In our project, SVM obviously found a set of hyperplanes that separates reasonably well the data points in high dimensions.

Logistic Regression

One straight-forward way of approaching this multi-class classification problem is breaking it down to a set of binary classification problem. Instead of attempting to put the recipe in a specific cuisine directly, we can check its probability of being in each cuisine and

then select the cuisine with the highest probability. This was the idea behind the logistic regression approach.

Logistic regression is a type of regression whose response variable is categorical. Unlike linear regression where the response variable is a numerical value, logistic regression tells us the possibility of a data point being in a class. Other aspects (e.g. MSE) of logistic regression is basically the same as linear regression.

R studio has a lot of built in regression functions, including glm which can be used to run logistic regression. We wanted to create models to calculate the probability of a recipe belonging to a specific cuisine. In order to do so, we made dummy field for each cuisine class, and marked recipe belonging to that cuisine class "1" for the field and recipe not belonging to that cuisine class "0" for the field. Usually, for regression models, we choose some criteria for model pruning. For example, if we use features' p-value as selection criterion, we would take away features having high p-value (fail to reject null hypothesis).

Because the number of features we needed to include was so big (100+) , model pruning was a very challenging task. Obviously, this was not doable by hand, so we used stepAIC provided in the MASS package, which used AIC as pruning criterion. However, the computation was really expensive and not very effective. In order to compute the best model, we need to consider the power set of all features (100+), which is a huge number. We ran stepAIC for about 2 hours and no result was output. Since this was only for one class model, we decided that we would stick to the other methods instead of logistic regression.

Decision Tree

The first method we tried is decision tree. Decision tree is a supervised method used for classification and regression. This means the training data come with labels suggesting the class of observations. We use scikit API to construct the decision tree. It uses C4.5 algorithm which is a greedy algorithm finding for each node the feature that produces the most information gain. It then prunes the tree by removing rules' preconditions if their removal can improve the accuracy. It will stop when all samples in a node belong to the same class, or no features give more information gain, or no samples are left. After building the tree with training data, we can then iterate the tree with test file.

One problem of decision tree is overfitting. This may happens in an induced tree, because it has too many branches and some may be noise. We used cross validation to test the accuracy. With default decision tree setting of scikit, the accuracy calculated from Kaggle's data is 61.2%. The accuracy calculated from 10-fold cross validation is 59.8%. The small difference indicates our decision tree does not have overfitting problem.

By adjusting parameter of decision tree classifier, we can improve our result. The default setting criteria is Gini impurity. It randomly labels dataset and measures how often they are wrong. We changed the setting to entropy for the information gain. Entropy calculated by the following formula:

$$E(S) = -p_{\oplus} \cdot \log_2 p_{\oplus} - p_{\ominus} \cdot \log_2 p_{\ominus}$$

p_{\oplus} is the positive proportion of examples in a class. p_{\ominus} is the negative proportion of examples in a class. Then we decide which class to choose by comparing their information gain by the following formula:

$$Gain(S, A) = E(S) - I(S, A) = E(S) - \sum_i \frac{|S_i|}{|S|} \cdot E(S_i)$$

Then we choose the attribute maximizing the information gain.

The accuracy calculated from Gini is 59.8% with 10-fold cross validation while from information gain is 54.8%. Therefore, Gini is a better way to construct our decision tree.

Overall, the accuracy calculated by decision tree is relatively low. It mainly due to some drawbacks of decision trees. The first is imbalance. If we use unbalanced dataset to train the attribute, prediction will be biased to the class that is more frequent. Second is noise. A small change in the dataset may result in a completely different decision tree. For example, in the train file, there are 9945 datasets, and each dataset has 1000 attributes. I changed 3 attributes value in one datasets. The accuracy changed from 59.8% to 60.5%. It is consideration variation considering how big is the data and how small is the dataset change.

Naive Bayesian Classifier

Similar to decision tree, Naive Bayesian Classifier method is also a supervised learning algorithm. It assumes the independence among features. It uses the following formula:

$$P(y | x_1, \dots, x_n) = \frac{P(y)P(x_1, \dots, x_n | y)}{P(x_1, \dots, x_n)}$$

y is a class variable and xn is dependent features. Then we used naive assumption:

$$P(x_i | y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i | y)$$

Then we can simplify the first formula to:

$$P(y | x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^n P(x_i | y)}{P(x_1, \dots, x_n)}$$

Because the denominator is constant, we can determine which attribute has the most probability by comparing:

$$P(y) \prod_{i=1}^n P(x_i | y)$$

Because in our dataset we judge if an ingredient belong to the classifier by whether it equals 0 or 1, we decided to use Bernoulli Naive Bayes Model to calculate $P(x_i | y)$. This model only takes account of whether the term occurs but not how often the term occurs.

We used 10-fold cross validation to test if there is overfitting problem. With default decision tree setting of scikit, the accuracy calculated from Kaggle's data is 71.2%. The accuracy calculated from 10-fold cross validation is 71.4%. The difference is very small, so there is no overfitting problem.

One advantage of Naive Bayesian Classifier is noise handling. I changed the same dataset as decision tree above. The result remains unchanged.

One disadvantage of Naive Bayesian Classifier is the assumption that features are independent. This will definitely make our prediction inaccurate. There are certainly some connections among ingredients. For example, Chinese restaurant will use rice and soy sauce. Naive Bayesian Classifier ignores their dependency.

Ensemble

The ensemble method applied in our project is bootstrap aggregating (bagging). Bagging samples the complete training data set to form many smaller training data sets, and then trains a weak learner such as decision tree on each individual sample training data. Finally, it incorporates the votes from each decision tree to make a decision when predicting on a new data point.

In our project, because of the large number of examples and features, bagging took very long to finish. In fact all ensemble methods take very long to finish on large data sets because they require the training of dozens, hundreds, or even thousands of weak learners.

We used the *fitensemble* function in MATLAB and used bagging as our method and decision tree as the weak learner. We obtained 74.3% cross validation accuracy using 2000 features on 100 weak learners. Due to the long training time, we were not able to use more features or more weak learners. Note that the underlying weak trainer is decision tree, which in previous part has been shown to be of a mediocre performer. Therefore, the enhancement from bagging should be a result of combining many weak learners, and thus heavily influenced by the number of weak learners.

The low accuracy is probably a result of the small number of weak learners. We did not have more time to wait for the results of training more weak learners. A plot which shows the decreasing error with respect to the number of weak learners would have helped to support our arguments, but again we did not have time to do more analysis. Usually bagging achieves superior performance in object classification. If we have time to train more weak learner, we are sure we can increase its accuracy.

Future Improvements

There are a couple of things that are worth to work on. First, as decision tree classifier and PART rule-based classifier cannot give good result, we may need to balance our training dataset in feature selection part. We can either randomly split classes with more instances into smaller pieces, or duplicate classes with fewer instances. With balanced dataset, we can expect a better result using decision tree based classifier. Second, to further improve our result, we still need to work on feature selection. As natural language processing is always a challenge in computer science, instead of mapping ingredient name ourselves, we can find and utilize an online cooking ingredient database, and map training and testing ingredients into uniform ingredients in online database. Furthermore, we can search for result from online cooking database, and use it as a reference and testing method when we train our model.

Conclusion

Overall, our project was successful. We reached 79.435% accuracy on the test set on Kaggle via careful feature selection and SVM. All of us explored and learned new aspects of classification problems and enjoyed this learning experience.

References

Asa Ben-Hur and Jason Weston. A User's Guide to Support Vector Machines. *Methods in Molecular Biology*. **2010**, 609, 223-39. DOI: 10.1007/978-1-60327-241-4_13. [1]