**Data Optimization**

Suffix-Optimization

We realized that we can apply an optimization to the train.json and test.json data that we received. The optimization consists of replacing each ingredient in the JSON files with the last word each ingredient. For example let us say that train.json contains the following:

[

{

“id”: 1,

“cuisine”: “greek”,

“ingredients”:

[

“lettuce”,

“olives”,

“black pepper”,

“purple onion”

]

},

{

“id”: 2,

“cuisine”: “greek”,

“ingredients”:

[

“Romanian lettuce”,

“black olives”,

“pepper”,

“onion”

]

}

]

If we apply our optimization to the data above, we receive the following data:

[

{

“id”: 1,

“cuisine”: “greek”,

“ingredients”:

[

“lettuce”,

“olives”,

“pepper”,

“onion”

]

},

{

“id”: 2,

“cuisine”: “greek”,

“ingredients”:

[

“lettuce”,

“olives”,

“pepper”,

“onion”

]

}

]

This optimization throws out the preceding adjectives in a given ingredient. Most of the time, these adjectives are not useful to us.

This optimization is useful for several reasons. First of all, the optimization causes our algorithms to use less space and time when processing the data because we have compressed the data to contain only the useful parts. However, more importantly, this optimization allows ingredients with varying prefixes to be classified as the same ingredient. For example, we would like to consider “olives” and “black olives” as the same ingredient because there is a small difference between the two. We can view our optimization as a method of “cleaning” or “filtering” the noisy data into simpler, more accurate data.

We realize that this optimization is not perfect and has downsides. For example, applying the optimization will result in our algorithms thinking “garlic powder” and “curry powder” as the same ingredient. This optimization yields better results for relatively simpler algorithms and yields worse results for more complex algorithms.

**Naïve-Bayesian Classifier**

The Naïve-Bayesian Classifier should be one of the first algorithms we should try when attempting to solve a classification problem. The reason is because the Naïve-Bayesian Classifier is very simple to implement (~70 lines in Python) and runs relatively fast (under 5 seconds).

We ran the Naïve-Bayesian Classifier on train.json in order to first build the classifier. We then ran the Naïve-Bayesian Classifier on the same train.json used to create the classifier, and we got an accuracy of 83%. We then ran Naïve-Bayesian Classifier on test.json to get the classifications and submitted our output to Kaggle. We received an accuracy of 64%. An accuracy of 64% is decent considering the simplicity and speed of this algorithm.

When the Naïve-Bayesian Classifier was applied to the suffix-optimized data, we received an accuracy boost of 0.3 percent.

Overall, the Naïve-Bayesian Classifier is useful to us because it is very quick and can give us a good baseline for the rest of our estimates. However, the downside is that it is not very accurate. We will explore more accurate algorithms later in this paper.

**Decision Tree Classifier**

We used Scikit-learn DecisionTreeClassifier module in order to build a decision tree from the training data. Using this decision tree, we were able to classify the test data.

Using decision trees took very long. Building the decision tree and classifying the data took at around 5 minutes. The decision tree approach was also relatively inaccurate. Its prediction percentage on the test data was only 60%. The decision tree approach is both considerably slower and less accurate than the Naïve-Bayesian classifier.

In order to use the DecisionTreeClassifier module, we needed to convert our “ingredients” data into a binary feature vector for each recipe. The size of our feature vector is equivalent to how many different ingredients we have in our entire training data. Each element in the feature vector is either 0 or 1: 0 if the recipe doesn’t contain the current ingredient and 1 otherwise. Since there are 6714 different ingredients in our training data, this means each feature vector for each recipe has 6714 elements. Since we have 39,774 different recipes in our training data, that means that we have 39,774 different feature vectors. Just to simply store our decision tree, we are using 1.1 GB of space (assuming integers are 4 bytes). This large memory requirement can lead to lots of page-swapping in the operating system, which could be part of the reason why this method is so slow. Note that we also need to convert our test data into feature vectors in order to get a prediction from the DecisionTreeClassifier. After the decision tree is built, we can pass our tests-data feature vectors to the classifier and receive predictions for our test data.

Overall, using decision trees for this data set has no benefits. It is both slow an inaccurate.

**Stochastic Gradient Descent**

Stochastic Gradient Descent (SGD) is a simple but efficient way to create a linear classifier. It makes use of Support Vector Machines (SVM) in its computation. The SGD classifier works by minimizing a multivariable objective function by using the “gradient descent” strategy. The “gradient descent” strategy is a method to finding the local minimum of a function by following the negative gradient of a function at a given point until a local minimum is achieved.

We made use of SGD in our Python programs by importing the SGDClassifier module. We convert our training data into feature vectors and then pass the SGDClassifier a set of feature vectors to build the classifier. We then convert our training data into feature vectors and pass it to the classifier in order to receive predictions.

SGD is a good algorithm for this problem. It runs very fast considering that it needs to convert all the training data into feature vectors first. SGD is runs in about 10 seconds and yields a 77% prediction rate on the testing data. This is a high prediction rate considering the speed of the algorithm.

SGD requires a lot of training data (more than 1 million samples) in order to achieve good prediction results. Note that although we only have around 40,000 samples, we are still able to get decent results with SGD. If we had more training samples, then SGD might be the most optimal algorithm given its prediction rate increase and its speed.

SGD is widely used in the industry for text-classification and Natural Language Processing (NLP). Since our classification problem is somewhat related to both of the applications, it can help explain why SGD produces good results in this case.