What's Cooking? Predicting Cuisines from Recipe Ingredients

Kevin K. Do
Department of Computer Science
Duke University
Durham, NC 27708
kevin.kydat.do@gmail.com

Abstract

Kaggle is an online platform for hosting data science competitions. On September 9, Kaggle opened a contest called "What's Cooking?". The goal of this contest is to predict the type of a cuisine (e.g. "Chinese" or "Mexican") from the list of ingredients in a recipe. In this paper, we describe our methodology for training an algorithm to make such predictions, which ultimately achieved a score of .78107 on Kaggle, placing 366th out of 841 teams.

1 Data

The training JSON file is an array of objects, each with the list of ingredients and the cuisine. The testing file contains objects in the same format with the cuisine removed.

```
"id": 24717,
"cuisine": "indian",
"ingredients": ["tumeric", "vegetable stock", "tomatoes", "garam
masala", "naan", "red lentils", "red chili peppers", "onions",
"spinach", "sweet potatoes"]
```

2 Methodology

2.1 Data Exploration

Before training any machine learning algorithms, we first examined the data to get a feel for its general structure. The training file contains 39,774 entries described in 666,921 lines. The testing file contains 9,944 entries across 157,117 lines. These are fairly modest numbers that do not require sophisticated parallelism or out-of-core algorithms. A cursory glance at 5 random entries in the training and test file showed that the structure and contents of the files are similar, and that the ingredients list does not contain stopwords (i.e. words with relatively little semantic meaning, like "the"). However, some of the ingredients have accents. We decided to normalize these words by turning them into their unaccented counterparts to prevent issues where one version of the word is accented and another is not.

On the basis of this initial data exploration, we decided to use a fairly standard approach in Python using `scikit-learn`.

2.2 Transforming Input Data

The approach we used for representing the textual data is from `scikit-learn's` tutorial, "Working With Text Data".

Loss function	Empirical accuracy	Generalization accuracy
hinge	0.73	0.71
log	0.62	0.62
modified_huber	0.78	0.76
squared_hinge	0.77	0.76
perceptron	0.80	0.73

Table 1: Empirical and generalization accuracies for a variety of loss functions for the SGD classifier

We wrote a basic custom parser to remove accents from the lists of ingredients and concatenate each list into one string, or document. This method removes a small amount of structure from the text, since we can no longer represent the list boundaries between ingredients. Nevertheless, this is a relatively small price to pay, as we are not giving up clause, sentence, or paragraph structure (which is typical in other text classification tasks).

The most basic and popular method of representing text for machine learning is the bag-of-words approach. Using this approach, we represent lists of ingredients by vectors of counts of each of the different words in the vocabulary. This is quite feasible given the relatively small size of our data set: the resulting vocabulary contained $\sim$3000 words.

We further improved on these vectors with two optimizations built into `scikit-learn`. First, we divided each of the word counts by the total number of words in that document to get term frequencies. This helps us normalize over various document lengths. The second optimization we performed was to weight words based on the inverse of number of documents they occur in. This helps us remove focus from words that are common to many documents and are therefore not very informative for distinguishing cuisines.

To evaluate each algorithm's ability to generalize, we divided the training data into two vectors: one containing 35000 points on which to train the algorithms, and another of the remaining 4774 points for evaluation. In the remainder of this report, "empirical accuracy" refers to the percentage of the 35000 training points that are correctly labeled, while "generalization accuracy" refers to the percentage of the other 4774 points that are correctly labeled.

2.3 Naive Bayes Classifier

Our first attempt at a classifier was the Naive Bayes Classifier. Though Naive Bayes is not typically regarded as a strong text classifier, it served us well as a foundation for further studies. The Naive Bayes classifier achieved fairly good performance, with an empirical accuracy of 0.69 and a generalization accuracy of 0.68. The similarity in these values suggests that there is not much overfitting occurring.

2.4 SGD Classifiers

We next used the standard tool for text classification, a linear classifier optimized using stochastic gradient descent. `scikit-learn's SGDClassifier` allows for many different loss functions. We tested the ones appropriate for classification: `hinge`, `log`, `modified_huber`, `squared_hinge`, and `perceptron`. The accuracies for the different loss functions are shown below. Most of the loss functions do not show a significant amount of overfitting, except the `perceptron` loss function. In the end, we settled on the `modified_huber` loss function as the most promising.

2.5 Alternative Regularization Schemes

Next, we tuned the parameters for the SGD classifier with the `modified_huber` loss function. The results above used the default L2 regularization. We tested L1 regularization but the results were poor, with empirical and generalization accuracies of 0.69. Furthermore, we had no interest in producing a sparse model, so we did not pursue L1 regularization (and therefore the ElasticNet) further.

2.6 Parameter Optimization

The regularization parameter determines how heavily the optimization weights the L2 regularization term. We performed a search of the parameter space (Figure 1) and determined that the optimal regularization parameter was 1e-4.

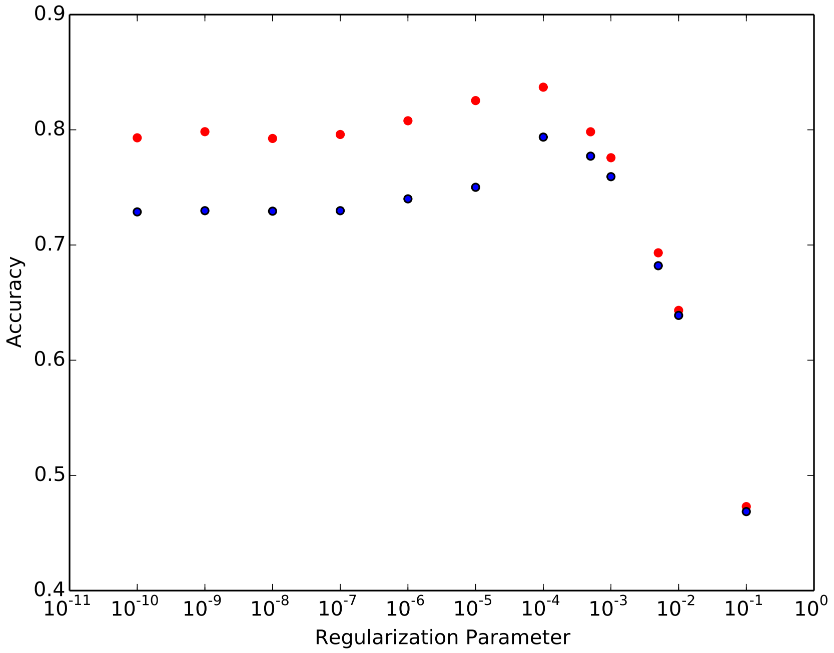

Figure 1: Plot of empirical (red) and generalization (blue) accuracies versus regularization parameter. The accuracies decrease sharply with regularization parameters greater than 1e-3.

2.7 Voting Classifier

With the SGD classifier, further improvement was not obvious. We therefore turned to ensemble methods in an effort to improve the classifier's performance. `scikit-learn` has a `VotingClassifier` that combines different classifiers into one based on either hard or soft voting. Basic tests indicated that soft voting yielded superior results. We then set out to determine what other classifiers could easily be built for inclusion in the VotingClassifier.

As noted above, Naive Bayes was not very good. Decision trees were good, with generalization accuracies around 0.65. However, we decided to upgrade the decision trees into randomized forests, which, after tuning, achieve generalization accuracies around 0.79. We found that optimal parameters for the forest were a maximum tree depth of 60 and a maximum number of trees of 30. Any more slowed down training without an increase in generalization accuracy (though empirical accuracy continue to increase, yet another example of overfitting).

The `VotingClassifier` itself has parameters to tune, namely the weights it places on the probabilities outputted by the subclassifiers (i.e. the randomized forest and the linear SGD classifier). The SGD classifier seemed to generate superior generalization accuracies in preliminary testing, but we performed a grid search over several different weightings. The final weighting of 1 for the randomized forest and 3 for the SGD classifier yielded the best generalization accuracy.

2.8 Performance concerns with K nearest neighbors

In testing on a small subset of the entire training set, the K nearest neighbors (KNN) classifier seemed promising. In fact, its generalization accuracy was greater than that of the linear SGD classifier. However, we were unable to complete a single training of a KNN classifier on the entire large dataset. On the first attempt, the process's memory usage increased to 30 GB before it was stopped. In an effort to keep the memory usage under 16 GB (the amount of RAM on the training computer) and prevent excessive paging to disk, we switched to more memory efficient tree structures for the KNN algorithm. Though both the `kd_tree` and `ball_tree` data structures kept the memory usage around 1.3 GB, we were nevertheless still unable to train a KNN algorithm on the entire dataset.

3 Conclusion

As our first attempt at training a machine learning algorithm, we are pleased with our performance. Our results show the power of ensemble learning, since it is doubtful we would have been able to improve the linear SGD classifier any more.

One potential avenue for exploration is to include the KNN algorithm in some sort of ensemble classifier. Results that we achieved by training on a small subset of the training data were promising, but we did not have the computing power necessary to train the KNN algorithm on the full dataset.

Acknowledgments

The author is grateful to Danielle Riggin for fruitful discussions about randomized forests.

References

[0] https://www.kaggle.com/c/whats-cooking

[1] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

A Code

```
import json
import io
import unicodedata
import sys
import numpy as np
from nltk import word_tokenize
from nltk.stem import WordNetLemmatizer
from sklearn import metrics
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import SGDClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.ensemble import RandomForestClassifier

DIAGNOSTICS = True

def strip_accents(s):
    return ''.join(c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn')

class LemmaTokenizer(object):
    def __init__(self):
        self.wnl = WordNetLemmatizer()
    def __call__(self, doc):
        return [self.wnl.lemmatize(t) for t in word_tokenize(doc)]

# Load data (train_small.json , train.json)
train_file = io.open('train.json', 'r')
train_json = json.loads(strip_accents(train_file.read()))
test_file = io.open('test.json', 'r')
test_json = json.loads(strip_accents(test_file.read()))

i = 0

X1 = []
Y1 = []
X1s = []
Y1s = []
for o in train_json:
    X1.append("␣".join(o['ingredients']))
    Y1.append(o['cuisine'])
    i += 1
    if i <= 3000:
        X1s.append("␣".join(o['ingredients']))
        Y1s.append(o['cuisine'])

id_test = []
X2 = []
for o in test_json:
    id_test.append(o['id'])
    X2.append("␣".join(o['ingredients']))

# Print diagnostics
if DIAGNOSTICS:
    print '%d␣=␣%d' % (len(X1), len(Y1))
    print '%s␣:␣%s' % (X1[0], Y1[0])
    print '%d␣=␣%d' % (len(X2), len(id_test))

# TfidfVectorizer
print "vectorize"
sys.stdout.flush()
vect = TfidfVectorizer(tokenizer=LemmaTokenizer()).fit(X1)
tf1 = vect.transform(X1).todense()
tf1s = vect.transform(X1s).todense()
```

```
tf2 = vect.transform(X2).todense()

# Train
print ”train”
sys.stdout.flush()

clf1 = RandomForestClassifier(max_depth=40, n_estimators=20).fit(tf1, Y1)
clf3 = SGDClassifier(loss='modified_huber', penalty='l2', alpha=1e-4,
    n_iter=5, random_state=65).fit(tf1, Y1)
clf = VotingClassifier(estimators=[('dt', clf1), ('sgd', clf3)], voting='
    soft', weights=[1, 3]).fit(tf1, Y1)

# Predict
print ”predict”
sys.stdout.flush()
Y1_hat = clf.predict(tf1)
Y2_hat = clf.predict(tf2)

# Print diagnostics
if DIAGNOSTICS:
    #print vect.vocabulary_
    print ”Empirical_accuracy:_%f” % np.mean(Y1_hat == Y1)
    #print metrics.classification_report(Y1, Y1_hat)

# Output predictions
out = io.open('submission.csv', 'w')
out.write(u'id,cuisine\n')
for i in range(len(X2)):
    out.write('%s,%s\n' % (id_test[i], Y2_hat[i]))
```