Machine Learning Engineer Nanodegree

Capstone Project - Cuisine Categorization by the recipe of ingredients

Sudhahar Thiagarajan
February 19, 2018

I. Definition

Project Overview

I'm not surprised when I come across this article "*Its official bay area now the culinary capital of the US per Michelin"*[1]

With my Indian background, I always wonder the Indian dishes which are most colorful, awash in the rich hues and aromas of dozens of spices: turmeric, star anise, poppy seeds, and garam masala as far as the eye can see. Once I moved to SF Bay Area, I was astonished with the cuisines of various nations & cultures are represented in the Bay Area restaurants. For e.g. thousands of dishes & recipes from Caribbean, Chilean, Vietnamese, Japanese, Indian restaurants and much more. I believe it's because of the immigrants of the Bay Area over the years, bringing along with them their native cooking styles, authentic recipes and tradition of interesting ingredients. And, the strongest geographic and cultural associations are tied to a region's local foods. This playground competitions asks you to predict the category of a dish's cuisine given a list of its ingredients.

I'm motivated by all the above reasons to work on this Kaggle Competition named "*What's Cooking*"[2] to categorize various types of cuisines by analyzing the given recipes and their wonderful ingredients.

On this domain, there was an academic paper submitted at the course (CSE 255 – a graduate-level course in University of California, San Diego)

You can find the same at:
https://cseweb.ucsd.edu/~jmcauley/cse255/reports/fa15/028.pdf
The related Kaggle data source can be found at:
https://www.kaggle.com/c/whatscooking/data

[1] https://www.mercurynews.com/2017/10/30/its-official-bay-area-now-the-culinary-capital-of-the-u-s-michelin-says/
[2] https://www.kaggle.com/c/whats-cooking

Problem Statement

The goal is to predict the cuisine category of a given recipe by its ingredients. First of all, I plan to have Feature engineering process through which I can create some manual features that don't exist in the dataset as default. Then I plan to perform the grid search to find optimal tuning parameters from different parameter combinations, searching for the combination that has the best cross validated accuracy. And then train models on either the **document term matrix** or the **manually created features** and finally on both. Also "ensemble" the models by stacking them. Optionally I can compare/benchmark against models such as Random Forests, those have built-in ensembling. Finally make the predictions on the new/test data, measure the scores and submit the file on Kaggle to get the targeted (> .70) score. Till the targeted score repeat the steps.

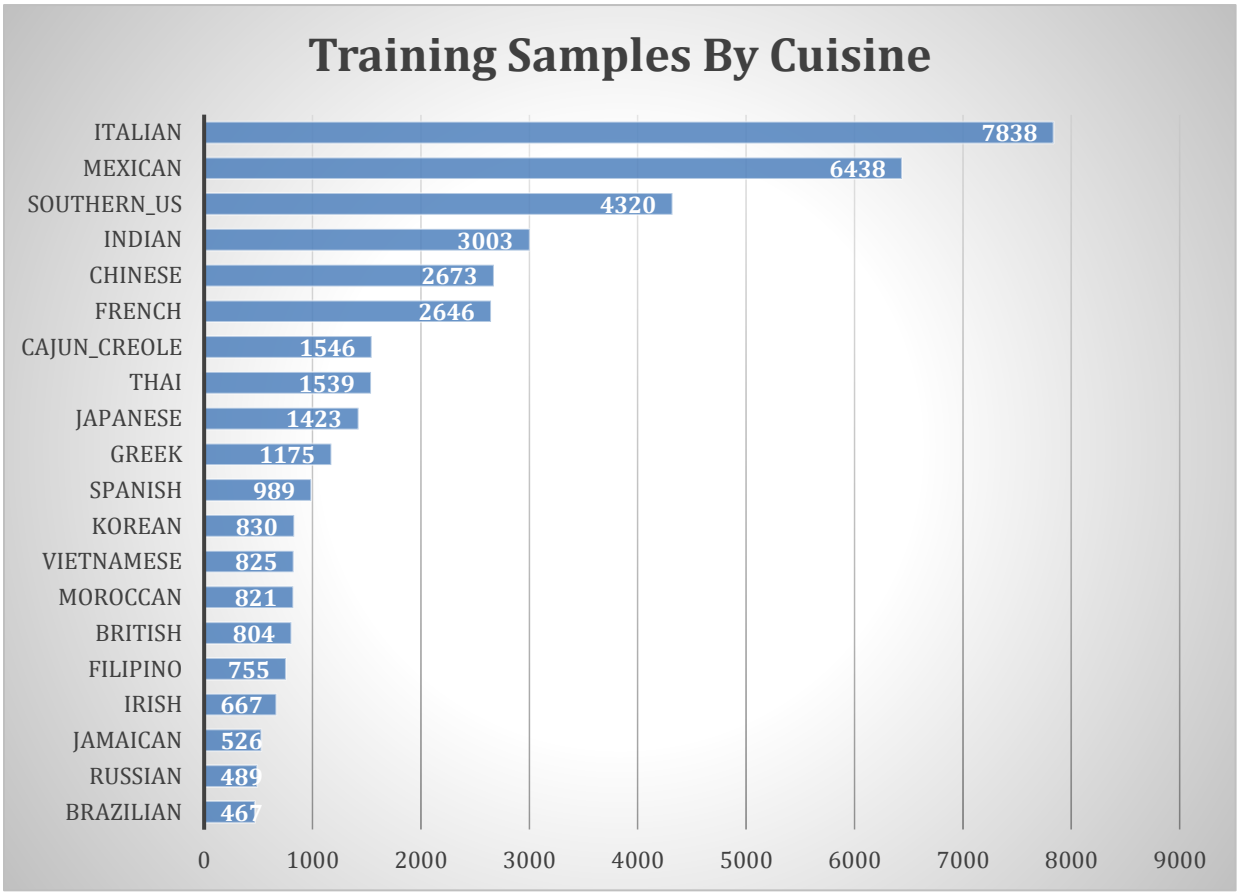
Kaggle provided the large dataset for various international cuisines and the respective recipes. Each recipe has the ingredients and the cuisine's category. Yummly3 is a number one recipe app with 2+ million recipes and 25+ million users. Yummly's mission is to be the smartest and most helpful food platform in existence. I'm here to develop the machine learning solution to is to predict a dish's cuisine category (Indian, Italian, Mexican etc.), by reasoning its ingredients.

Since there are various categories in the given dataset, I plan to use the ensembling4 of K-Nearest-Neighbor regression and multiclass Naive Bayes algorithms. Due to the nature of classification, I also plan to decide to try out various classifiers5 and compare the results produced by each of the following algorithms such as Decision Tree, Random Forest and XGBoost.

The dataset is to be downloaded from Kaggle competition "What's Cooking"6. The whole train dataset contains thirty thousand plus recipes along with their ingredients and cuisine categories. The test data contains close to ten thousand recipes. The training dataset has around twenty cuisine categories such as Indian, Greek, Jamaican, Filipino, Spanish, Mexican, Italian, Chinese, Thai, Southern USA, British, Vietnamese, Cajun, Brazilian, Japanese, French, Irish, Moroccan, Korean, and Russian.

I might need to add more features in addition to the default features such as number of ingredients, length of ingredients, string representation of the ingredients list. And, I plan to add more features using document term matrix7 along with manual added features, So I can train the models on both types of features. Mostly even I might need to combine both type of feature into one type of feature matrix SciPy8.

There are totally 39,774 records in the training dataset and 9,944 records in the testing dataset. The given dataset is highly biased as mentioned in the below. As we can see, the cuisine samples data distribution in training set is highly biased. Italian (total 7838 samples), Mexican (total 6438 samples) and Southern_US (total 4320 samples) cuisines comprises of almost 50% of the samples, while others have the remaining samples which is much less samples.



[3] https://www.yummly.com/
[4] http://blog.kaggle.com/2016/12/27/a-kagglers-guide-to-model-stacking-in-practice/
[5] https://github.com/rasbt/python-machine-learning-book-2nd-edition/blob/master/code/ch03/ch03.ipynb
[6] https://www.kaggle.com/c/whatscooking/data
[7] http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html
[8] https://www.scipy.org/scikits.html

Metrics

As per scikit documentation, below are the approaches to evaluate the accuracy of predictions of an any model:

- Estimator score method: Each estimator has a score method to provide a default evaluation criterion.
- Scoring parameter: Model evaluation methodology using cross validation such as cross_validation.cross_val_score, grid_search.GridSearchCV.
- Metric functions: The metrics module implements functions assessing prediction errors. for specific purposes.

Accuracy Score:

The accuracy_score function evaluates the accuracy, especially the number of correct predictions. By accuracy, it means that the ratio of the correct predictions to all the predicts. This metric helps us to evaluate the effectiveness of our model. It takes 4 parameters such as y_true, y_pred, normalize, sample_weight, where normalize & sample_weight are optional parameters. The parameter y_true takes an array of correct labels and y_pred takes an array of predicted labels that are returned by the model. It returns accuracy as a float value.

Formula:

$$\texttt{accuracy}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} 1(\hat{y}_i = y_i)$$

- $\hat{y}_i$ is the predicted value of the i-th sample.
- y_i is the corresponding true value.
- where $1(x)$ is the indicator function.

Why:

The Classification accuracy (percentage of correct predictions) is the number of correct predictions made as a ratio of all predictions made. But the challenge is that this is accurately suitable while the in the dataset are balanced. In order to use cross validation, I have to use pipeline. (Features will be created thru CountVectorizer, by sending the pipeline to cross_val_score and also, I can combine GridSearchCV with Pipeline. And:

- Classification accuracy is the easier to implement among other metrics
- But, it does reveal the underlying distribution of response variables
- And, it does not reveal what types of errors of my model.

Log Loss:

The log loss, also called logistic regression loss or cross-entropy loss, is a loss function defined on probability estimates. It's often used as an evaluation metric in kaggle competitions. Log Loss measures the accuracy of a given model by penalizing false classifications. It's generally to maximize the accuracy, we minimize the Log Loss. Here I plan to evaluate the probability outputs (predict_proba) of the classification model, rather than its distinct predictions.

Formula:

$$L_{\log}(Y, P) = -\log \Pr(Y|P) = -\frac{1}{N} \sum_{i=0}^{N-1} \sum_{k=0}^{K-1} y_{i,k} \log p_{i,k}$$

- $Y_{i,k}$=1 if observation i belongs to class j and 0 otherwise
- $P_{i,k}$=my predicted probability that observation i belongs to class k
- K=number of classes
- N=number of observations

Why:

- The use of log on the error results in highest punishments for being confident and wrong
- Log loss works by penalizing the false classifications, well suitable for multi-class classification. It's because Log Loss makes the classifier must assign probability to each class for all the samples.
- Log Loss's range is [0, ∞). Log Loss closer to 0 indicates higher accuracy; Log Loss is away from 0, indicates lower accuracy. So minimizing Log Loss gives greater accuracy for the model.

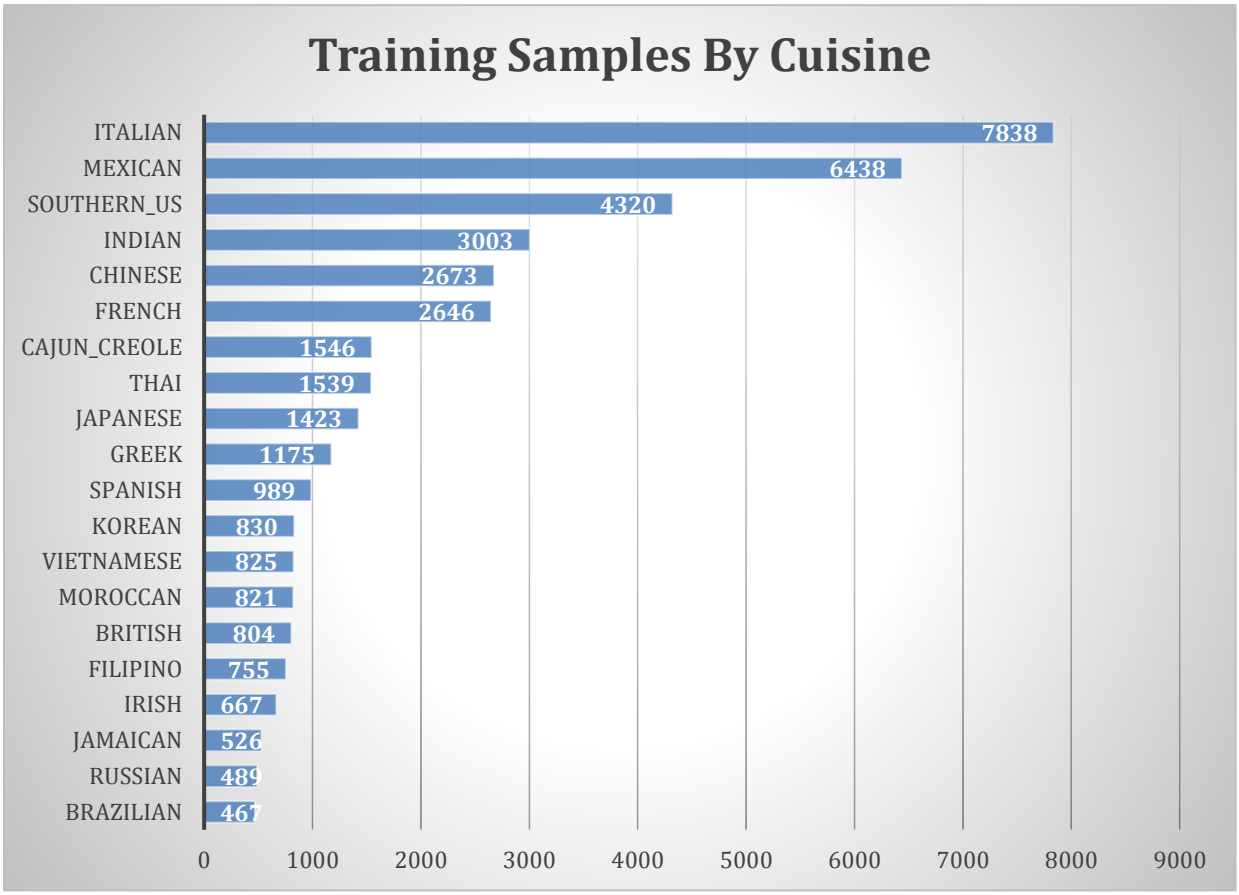
II. Analysis

Data Exploration

The dataset is to be downloaded from Kaggle competition "What's Cooking"6. The whole train dataset contains thirty thousand plus recipes along with their ingredients and cuisine categories. The test data contains close to ten thousand recipes. The training dataset has around twenty cuisine categories such as Indian, Greek, Jamaican, Filipino, Spanish, Mexican, Italian, Chinese, Thai, Southern USA, British, Vietnamese, Cajun, Brazilian, Japanese, French, Irish, Moroccan, Korean, and Russian.

Analysis of Cuisines:

There are totally 39,774 records in the training dataset and 9,944 records in the testing dataset. The given dataset is highly biased as mentioned in the below. As we can see, the cuisine samples data distribution in training set is highly biased. Italian (total 7838 samples), Mexican (total 6438 samples) and Southern_US (total 4320 samples) cuisines comprises of almost 50% of the samples, while others have the remaining samples which is much less samples.



Analysis of ingredients:

There are about 20+ set of least common ingredients and the following ingredients each present in only one recipe.

adobo all purpose seasoning, black grapes, broccoli romanesco, burger style crumbles, chinese buns, clam sauce, cooked vegetables, country crock honey spread, custard dessert mix, dried neem leaves, egg roll skins, flaked oats, ginseng tea, gluten flour, gluten-free broth, kraft mexican style shredded four cheese with a touch of philadelphia, mahlab, mild sausage, Minute white rice, orange soda, saffron road vegetable broth, schnapps, vegetarian protein crumbles, white creme de cacao.

Tuning the predictor to consider these least common ingredients might over-fit the training data. At the same time the most popular ingredients might have lower predictive power while categorizing the cuisines.

Exploratory Visualization

Next, let me explore on the data exploration. especially on

- how many ingredients there are? Out them, find the unique ingredients.
- how these ingredients made up of individual words?

To achieve this, a single list can be built by finding the ingredients in the dataset.

```
import re

total_ingredients = []
for lst_ingredients in train.ingredients:
    total_ingredients += [ingredient.lower() for ingredient in lst_ingredients]

no_of_ingredients = len(total_ingredients)
uniq_ingredients = len(set(total_ingredients))

print(no_of_ingredients)
print(uniq_ingredients)
```

```
428275
6703
```

After analyzing the dataset, there are 42,8275 ingredients and out of the ~43k ingredients, there are 6,703 total uniquely available ingredients.

But still I want to understand the correlation between the ingredients. For this I split them into individual words and also I took the uniquely available ingredient words.

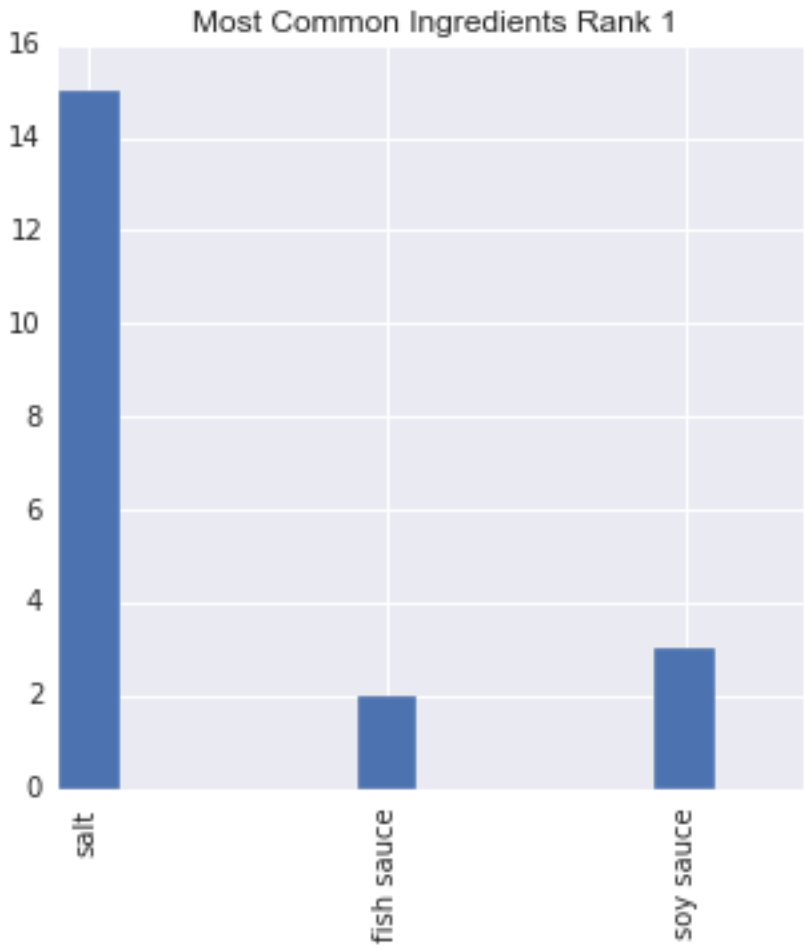
```
word_split = re.compile('[,. ]+')
total_ingredient_words = []

for ingredients in total_ingredients:
    total_ingredient_words += re.split(word_split, ingredients)

no_of_ingredients_words = len(total_ingredient_words)
uniq_ingredients_words = len(set(total_ingredient_words))

print(no_of_ingredients_words)
print(uniq_ingredients_words)
```

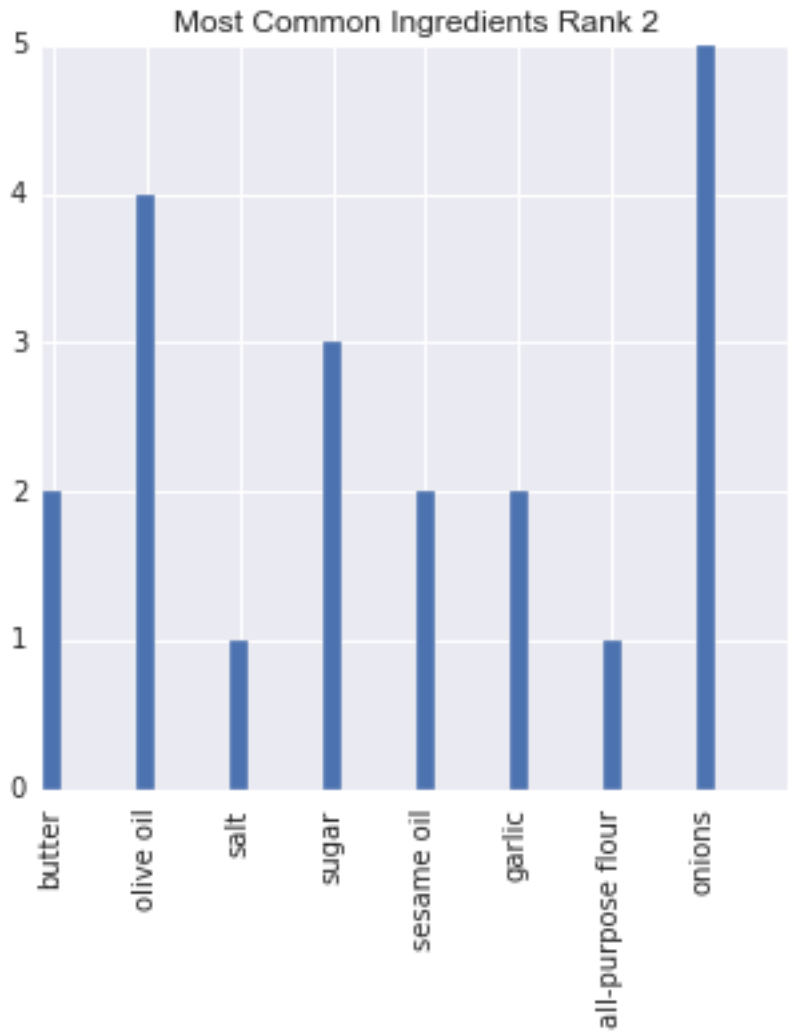
```
807802
3152
```

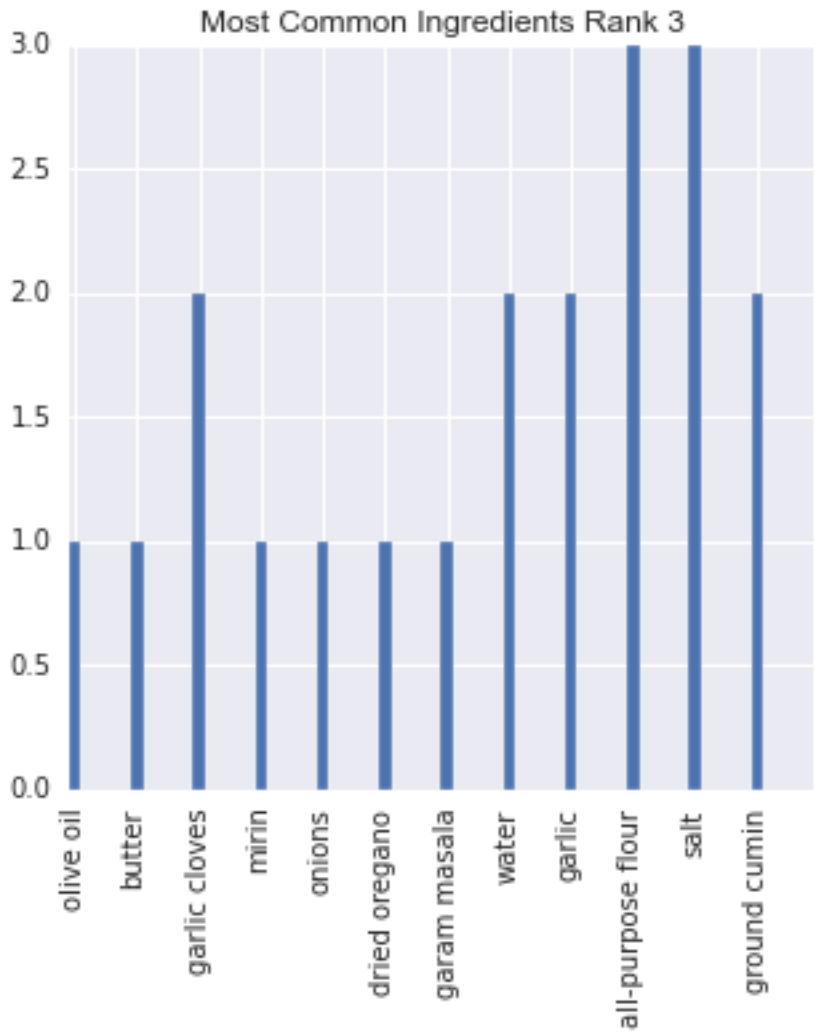
Stats Summary:

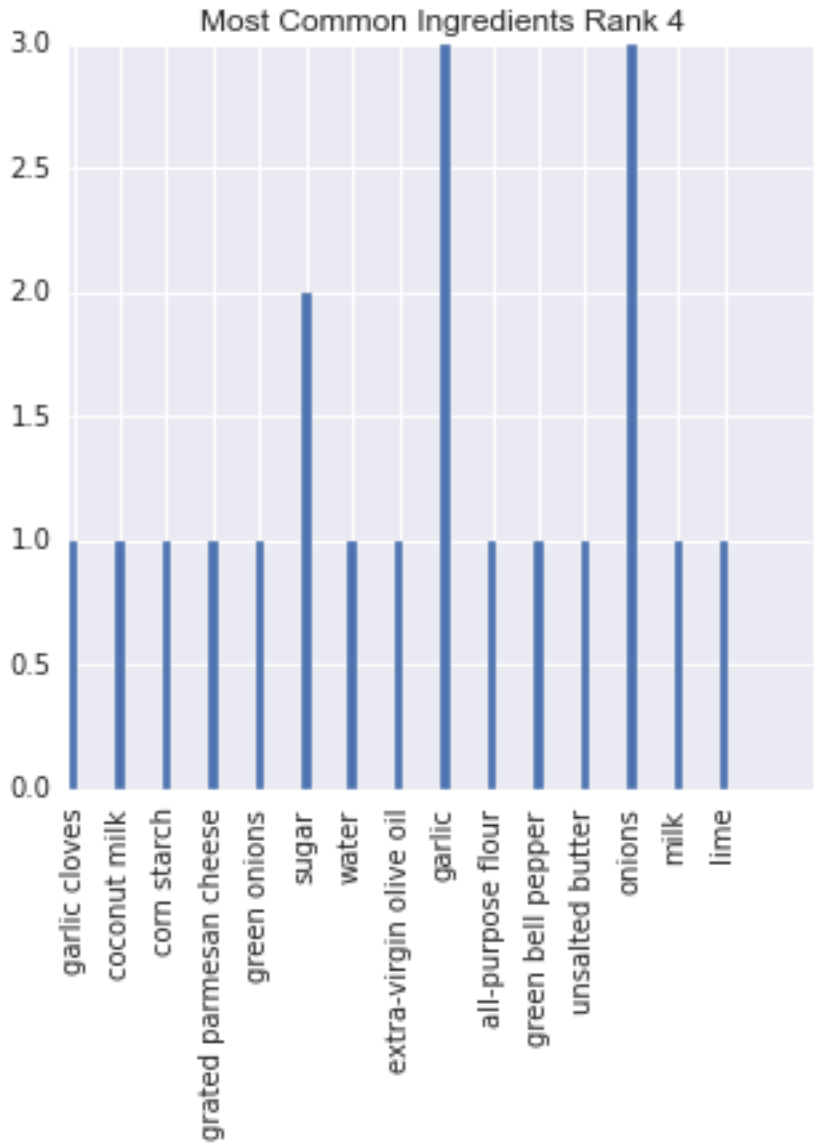
- Totally 42, 8275 ingredients
- Out of the above, totally 6,703 uniquely available ingredients
- Totally there are 807,802 individual ingredient words
- Out of the ~808k words, totally 3,152 uniquely available word

As I stated earlier, the cuisine samples data distribution in training set is highly biased. Italian (total 7838 samples), Mexican (total 6438 samples) and Southern_US (total 4320 samples) cuisines comprises of almost 50% of the samples, while others have the remaining samples which is much less samples. In precise, we can count the most common ingredients by cuisine. Later I did rank the most common ingredients by cuisine. The below charts mainly reveal that salty ingredient is the most common ingredient in almost all cuisines.


Most Common Ingredients Rank 1
16
14
12
10
8
6
4
2
0
salt
fish sauce
soy sauce


Most Common Ingredients Rank 2
5
4
3
2
1
0
butter
olive oil
salt
sugar
sesame oil
garlic
all-purpose flour
onions

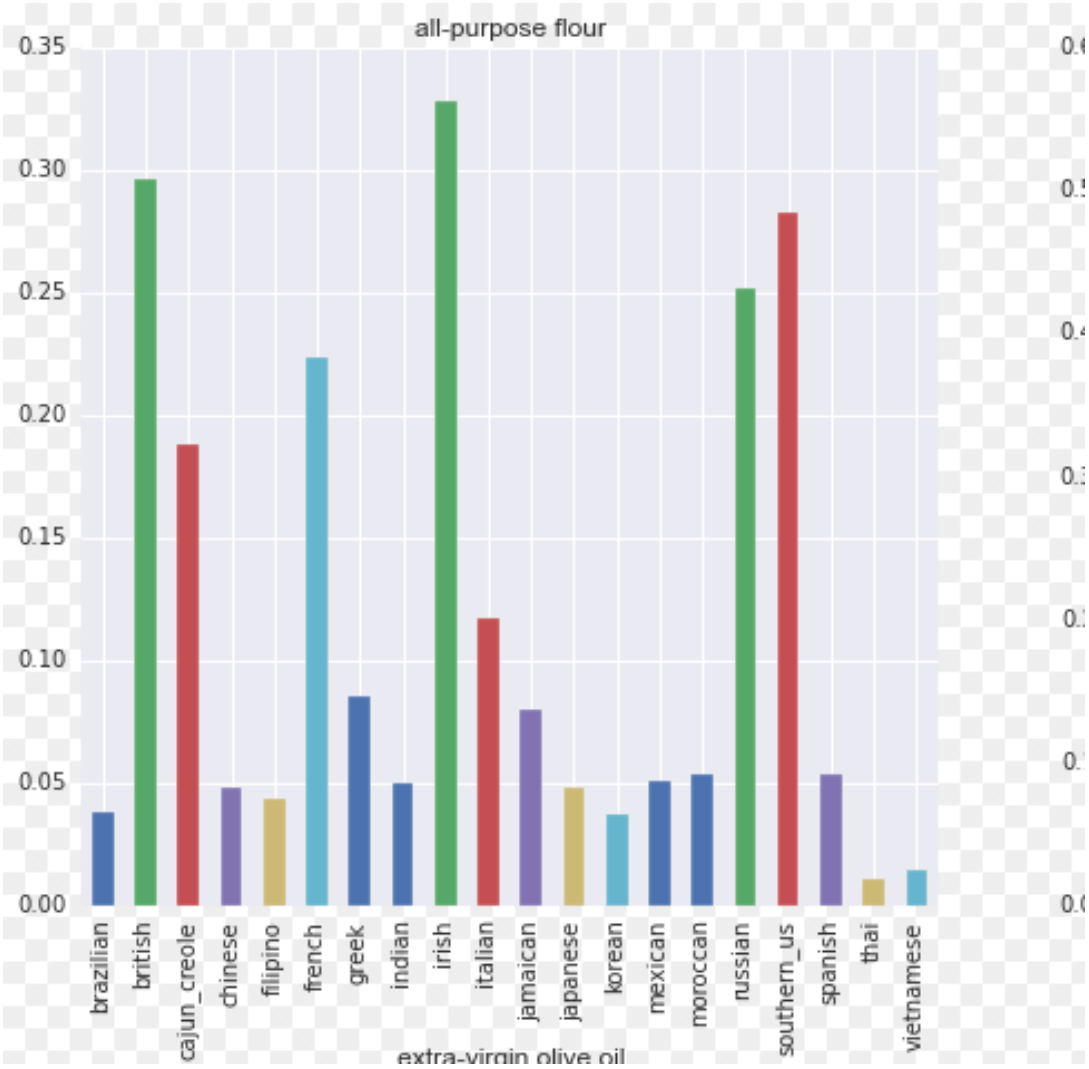

Most Common Ingredients Rank 3
3.0
2.5
2.0
1.5
1.0
0.5
0.0
olive oil
butter
garlic cloves
mirin
onions
dried oregano
garam masala
water
garlic
all-purpose flour
salt
ground cumin



Next, I'm going to plot this for 5 most common ingredients. First of all, determine the unique ingredients among the most common ingredients found in the previous section.

```
list_ingredients = np.unique(common_ingredients.values.ravel())

train['total_ingredients'] = train['ingredients'].map(":".join)

fig, ax = plt.subplots(5, 5, figsize=(40, 40))
for ingredient, ax_idx in zip(list_ingredients, range(25)):
    indexes = train['total_ingredients'].str.contains(ingredient)
    ingredient_occur = (train[indexes]['cuisine'].value_counts() / train['cuisine'].value_counts())
    ingredient_occur.plot(kind='bar', ax=ax.ravel()[ax_idx], fontsize=10, title=ingredient)

fig.savefig('ingredient_occur.plot.png')
```

Then I created the 5x5 sub plots as give below…




all-purpose flour
butter
coconut milk
corn starch
dried oregano
extra-virgin olive oil
feta cheese crumbles
fish sauce
garam masala
garlic
garlic cloves
grated parmesan cheese
green bell pepper
green onions
ground allspice
ground cinnamon
ground cumin
ground turmeric
large eggs
lime
milk
mirin
olive oil
onions
potatoes
brazilian
british
cajun_creole
chinese
filipino
french
greek
indian
irish
italian
jamaican
japanese
korean
mexican
moroccan
russian
southern_us
spanish
thai
vietnamese

Stats Summary:

- From the sub plots, I leaned that some ingredients have a high uniqueness. For e.g. in Asian cuisines, soy sauce & sesame oil have that; In Indian cuisines, garam masala has that and so on…
- Some ingredients like salt, sugar, pepper and oil are very common.

Observation:

- Special characters are there in the ingredient feature
- Brand names are there in the ingredient feature
- Spellcheck to be done
- Numeric are there

Because of these, I started with adding the below features.

- The total Number of ingredients
- The average length of ingredient names
- Convert the ingredients list into proper string

	cuisine	id	ingredients	no_ingredients	ingredient_len	ingredients_string
0	greek	10259	[romaine lettuce, black olives, grape tomatoes...	9	12.000000	[u'romaine lettuce', u'black olives', u'grape ...
1	southern_us	25693	[plain flour, ground pepper, salt, tomatoes, g...	11	10.090909	[u'plain flour', u'ground pepper', u'salt', u'...
2	filipino	20130	[eggs, pepper, salt, mayonaise, cooking oil, g...	12	10.333333	[u'eggs', u'pepper', u'salt', u'mayonaise', u'...
3	indian	22213	[water, vegetable oil, wheat, salt]	4	6.750000	[u'water', u'vegetable oil', u'wheat', u'salt']
4	indian	13162	[black pepper, shallots, cornflour, cayenne pe...	20	10.100000	[u'black pepper', u'shallots', u'cornflour', u...

Algorithms and Techniques

I'm using IPython notebook t solve this problem. I'm using Python 2.7 thru the Anaconda package with the necessary packages like Scikit-learn, Pandas, Numpy, SciPy and other additional required packages.

I'm using Decision Tree and Random Forest Classifiers to build the benchmark model.

Decision Tree:

- Strengths & Better Performance Areas:
  - Decision Tree models are mostly used for classification and regression.
  - Decision tree is capable of finding complex non-linear relationships in the data
  - In the case of interpretability, Decision Tree classifiers are very productive.
  - Feature scaling is not required.

    - Data Classification where we need to apply a a top down, divide & conquer and to be recursive approach to form a tree-based graph for data classification, decision tree models are well suitable.
- Weaknesses and Poor Performance Areas
    - Deeper the decision trees can easily result in overfitting.
    - Sensitive to training set rotation. (i.e. sensitive to small variations in the training data)
    - Too slow on bigger training set.

Random Forest

- Strengths & Better Performance Areas:
    - On a different random subset of the training set, say, a group of Decision Tree classifiers are trained on each set. And then we just take the the predictions of all individual trees, then predict the class that gets the most votes. Random Forest is such a versatile ensemble method of Decision Trees.
    - One of most powerful algorithms.
    - It works best when the predictors are as independent from one another as possible.
    - Can handle sparse data
- Weaknesses and Poor Performance Areas
    - Poorly perform for regression
    - Might over fit on noisy training data

For the actual implementation, I'm using KNN, Naive Bayes and ensemble them through model stacking.

Naive Bayes

- Strengths
    - It's simply easy and fast. It's well suitable for in multi class prediction areas.
    - While the independence assumption is true, this classifier performs better than other models like logistic regression.
    - Requires less training data set.
    - It performs well with non-numerical input variables too, such as categorical variables. This makes Naïve Bayes a better choice for text based classification.

I'm using the multinomial Naive Bayes classifier. It is very optimal for classification with disconnected features (e.g. in text classification). Though the multinomial distribution usually requires integer feature counts, that can be achieved by fractional counts using tf-idf sort of functions.

MultinomialNB implements the naive Bayes algorithm for multinomially distributed data. We can represent the data as word vectors. In other words, Naive Bayes allows us a chance that something can happen "given" that something else have already happened.

Let's say,

- there is an result Y & evidence X of the result.
- The probabilities are defined as such : The probability of having both the results "Y" and the evidence "X" is:

$$P(X \text{ and } Y) = (PX) \times P(B|A)$$

- P – Probability
- X – Event X
- Y – Event Y
- P(B|A) "Given"

It's based on Bayes' Theorem and as I explained above, the main factor is that the assumption of independence among predictors.
In other words, a Naive Bayes classifier

- assumes that the existence of specific feature in a class is not related to existence of any other feature
- For example, a fruit, say banana, if it is yellow, longer and about 6 inches in length.
    - Though these features depend on each other, these attributes independently add to the probability that this fruit is a banana
    - The intuition is popularly known as "Naïve".

There are three types of Naive Bayes s under scikit-learn library:

- Gaussian: Used in classification areas, mainly when there is normal distribution.
- Multinomial: It is used for discrete counts.
- Bernoulli: The binomial model is useful if the feature vectors are binary.

I chose Multinomial Naïve Bayes. Because there are many ways to improve such as:

- Implement transformation or different methods to convert it in normal distribution.
- Avoid over inflating by removing correlated features
- "alpha" - Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).
- fit_prior=[True|False] to learn class prior probabilities or not.
- Detailed pre-processing of data and the proper feature selection.

KNN

- Strengths
    - Implementation is easy
    - Has lot of flexibility on choosing features
    - Performs well for in multi class prediction areas.
    - By nature of the algorithm, KNN has the non-parametric, which is big advantage in the case of highly unusual data sets.

Overview:

Neighbors based classification is a non-generalizing learning.

- KNN stores the instances of the training data.
- The Classification is computed from a simple majority vote of the nearest neighbors of each point. For e.g KNeighborsClassifier's learning is based on the k nearest neighbors of each point.
    - where k is an integer value, can be specified manually.

I'm using the KNeighborsClassifier algorithm and can be other way explained as:

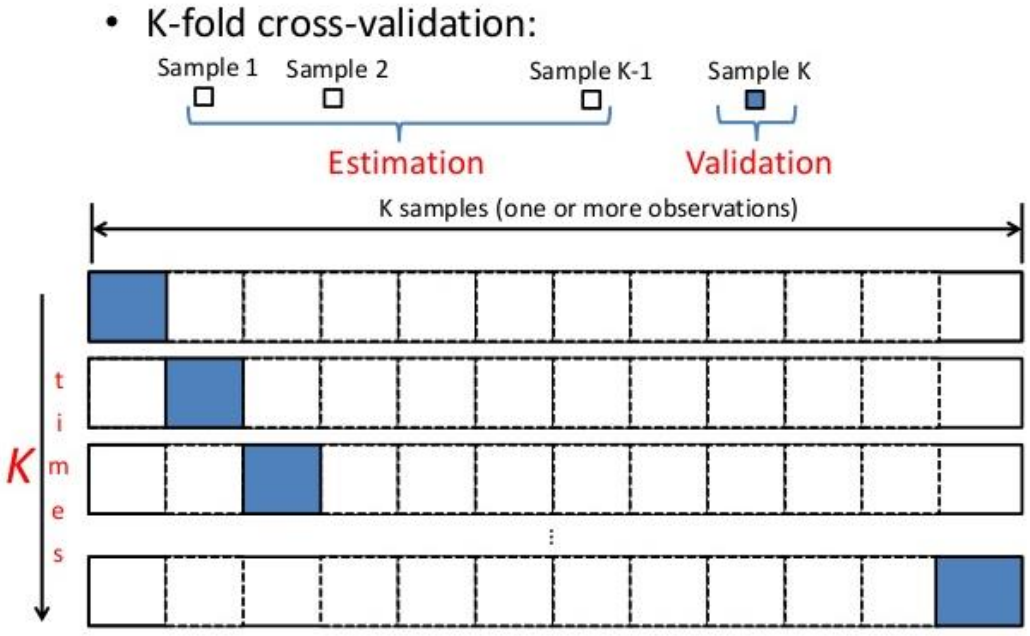
- We can specify the positive integer k along with a new test data
- The k entries which are closest to the new test data are to be selected,
- The most common classification of these entries is to be found.
- Now we can test this classification with the new test data

KNN Implementation:

- We can store the entire training dataset in KNN, whereas KNN uses as its representation.
- We can make KNN to make predictions just in time, where KNN calculates the similarity between an input test data and the training instances.

k-fold cross validation.

Here I can explain the method that can be used to tune the hyperparameter K.



- Generally, the best K is the lowest test error rate
  - When we carry out repeated measurements of the test error for different values of K, ends up in in capable of generalizing to newer observations nothing but overfitting.
  - Using the test set for hyperparameter tuning can lead to overfitting.
- In other way, we can estimate the test error rate by holding out a subset of the training set from the fitting process.
  - We can create an exclusive subset, known as validation set
  - It's to select the appropriate level of flexibility of KNN
  - One of the validation approaches is called k-fold cross validation.
- As seen in the above figure, k-fold cross validation randomly dividing the training set into k folds of equal size.
  - The first fold is a validation set
  - The method is fit on the remaining k−1 folds. The misclassification rate is then computed on the test data in the order of folds.
  - It's repeated k times; every time, different set of test data is treated as a validation set, results in k estimates of the test error and then averaged out.
- The scikit-learn has cross_val_score() method. By specifying as "cv=5" parameter, scoring metric as "accuracy", for the classification solutions.
- If needed, we can plot the K versus misclassification error.

For the better feature engineering and tuning purposes, I'm using the following tools and techniques for the improvisation of the accuracy of my solution.

- Using CountVectorizer, convert a collection of text documents to a matrix of token counts
- Adding/Creating features through Feature Engineering
- Feature processing & transformation
- Using a Pipeline, automate the workflow by chaining many transformers together.
- Optimize hyperparameters using RandomizedSearchCV and GridSearchCV
- While having lot of parameters to be tuned, it's tough or even not feasible to search all possible combinations of parameter values.
- Here I plan to use RandomizedSearchCV to search a sample of the parameter values so to control the computational limitation.
- Transformers - For data transformations, the transformer objects are widely used in my solution
- Using SciPy and FeatureUnion on adding features Ensembling models & Model stacking
- Using SciPy, I can combine the sparse and dense matrices.

Benchmark

As I mentioned above, I used Decision Tree for the benchmark purposes. First of all, I built the train & test datasets using the "train_test_split" function. Then I vectorized the ingredients and made a vocabulary dictionary of all tokens in the raw data of ingredients using CountVectorizer.

```
from sklearn.feature_extraction.text import CountVectorizer

# Preprocessing
vector = CountVectorizer(
    preprocessor = xform_string,
    analyzer = "word",
    token_pattern = r"(?u)\b[a-z]{2,40}\b",
    max_features = 4500
)

vector.fit(np.concatenate([X_train.ingredients, X_test.ingredients]))

print ("Total No. of features:", len(vector.get_feature_names()))
```

Then I predicted the results and ran the metrics too. I used "accuracy_score" and "fbeta_score" to measure the accuracy on the test and train datasets.
The accuracy_score function evaluates the accuracy, especially the number of correct predictions. By accuracy, it means that the ratio of the correct predictions to all the predicts. This metric helps us to evaluate the effectiveness of our model.

The F-beta score is the weighted average of precision and recall. It approaches its optimal value at 1 and its worst value at 0.

The metrics for Decision Tree:

```
Train Accuracy for Decision Tree ::  0.999622866840567
Test Accuracy for Decision Tree  ::  0.637460716530484
F-Score on Test for Decision Tree ::  [0.54945055 0.32913165 0.50671141 0.69538792 0.34801136 0.42744657
 0.56394316 0.75007638 0.39669421 0.73931889 0.48804781 0.59968229
 0.5104551  0.81358491 0.54224464 0.30373832 0.58120265 0.40251572
 0.60959792 0.45608108]
```

And also I tried the same with RandomForest too, just for the comparison purposes and also to set the right expectation.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

#RandomForest
benchmark_model_A = Pipeline([
    ("vector", vector),
    ("scl", StandardScaler(with_mean=False)),
    ("clf_A", clf_A)
])

benchmark_model_A.fit(X_train.ingredients,y_train.cuisine)

print ("#")
print ("# Best score:", benchmark_model_A.named_steps["clf_A"].oob_score_)
print ("#")

#DecisionTree
benchmark_model_B = Pipeline([
    ("vector", vector),
    ("scl", StandardScaler(with_mean=False)),
    ("clf_B", clf_B)
])

benchmark_model_B.fit(X_train.ingredients,y_train.cuisine)
```

Per scikit-learn documentation, the RandomForestClassifier is trained using bootstrap aggr egation, where each new tree is fit from a bootstrap sample of the z_i = (x_i, y_i). The out-of-bag (OOB) error is the average error for each training observations calculated using predict ions from the trees that do not contain the training observations in their respective bootstr ap sample. The best OOB Score is: 0.7257

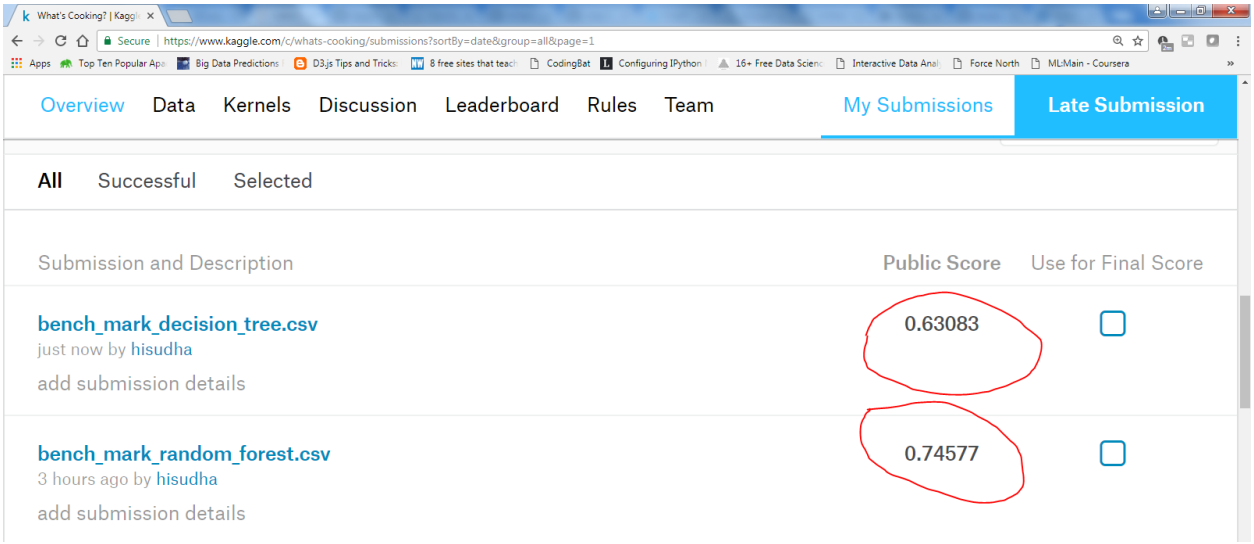
The model's implementations are such:

```
#Metrics

from sklearn.metrics import accuracy_score
from sklearn.metrics import fbeta_score
from sklearn.metrics import precision_score

pred_results_A = benchmark_model_A.predict(X_test.ingredients)
print(pred_results_A)

# Train and Test Accuracy for Random Forest
print ("Train Accuracy for Random Forest :: ", accuracy_score(y_train, benchmark_model_A.predict(X_train.ingredients)))
print ("Test Accuracy for Random Forest  :: ", accuracy_score(y_test, pred_results_A))

print ("F-Score on Test for Random Forest :: ",fbeta_score(y_test, pred_results_A,average=None, beta = 0.5))

pred_results_B = benchmark_model_B.predict(X_test.ingredients)
print(pred_results_B)

# Train and Test Accuracy for  Decision Tree
print ("Train Accuracy for Decision Tree :: ", accuracy_score(y_train, benchmark_model_B.predict(X_train.ingredients)))
print ("Test Accuracy for Decision Tree  :: ", accuracy_score(y_test, pred_results_B))

print ("F-Score on Test for Decision Tree :: ",fbeta_score(y_test, pred_results_B,average=None, beta = 0.5))
```

The metrics for Random Forest:

```
Train Accuracy for Random Forest ::  0.9995285835507087
Test Accuracy for Random Forest  ::  0.754242614707731
F-Score on Test for Random Forest ::  [0.77039275 0.55194805 0.71315372 0.75520048 0.62022901 0.56403051
 0.7688172  0.83874612 0.59055118 0.75391332 0.77844311 0.76701822
 0.82533589 0.85977911 0.8056872  0.5952381  0.69250871 0.69469835
 0.74528302 0.66666667]
```

Finally, I created the Kaggle submission files and submitted the same in Kaggle competition page. The results are:



III. Methodology

Data Preprocessing and Implementation

As I stated earlier, I started with adding the below features.

- The total Number of ingredients
- The average length of ingredient names
- Convert the ingredients list into proper string

	cuisine	id	ingredients	no_ingredients	ingredient_len	ingredients_string
0	greek	10259	[romaine lettuce, black olives, grape tomatoes...	9	12.000000	[u'romaine lettuce', u'black olives', u'grape ...
1	southern_us	25693	[plain flour, ground pepper, salt, tomatoes, g...	11	10.090909	[u'plain flour', u'ground pepper', u'salt', u'...
2	filipino	20130	[eggs, pepper, salt, mayonaise, cooking oil, g...	12	10.333333	[u'eggs', u'pepper', u'salt', u'mayonaise', u'...
3	indian	22213	[water, vegetable oil, wheat, salt]	4	6.750000	[u'water', u'vegetable oil', u'wheat', u'salt']
4	indian	13162	[black pepper, shallots, cornflour, cayenne pe...	20	10.100000	[u'black pepper', u'shallots', u'cornflour', u...

First, I defined the regex pattern for tokenization. Then I imported and declared the Multinomial Naive Bayes along with the default parameters. I assigned the Naive Bayes wit h a pipeline of vectorization.

Then I started optimizing cross validation using Pipeline.

The steps are:

- Using pipeline with cross_val_score, features are created from X (through CountVectorizer) in each fold of cross validation.

- By having this implementation, we can simulate the real world of solutions, where our out of sample data can contain features that were not seen while training the model.

And then I integrated GridSearchCV along with Pipeline for the following reasons.

- GridSearchCV can be used to locate better tuning parameters through rigorous grid search of different parameter probables, searching for the right combo that has the optimum cross validated accuracy.
- By integrating a Pipeline to GridSearchCV, we can search tuning parameters for both the vector and the algorithm.

I cross validated the entire pipeline. I created a grid of parameters to search and specify the pipeline step along with the default parameters. I examined the score for each combination of parameters and then printed the single best score & parameters that produced that score.

```
# print the single best score and parameters that produced that score
print(gridCV.best_score_)
print(gridCV.best_params_)
```

```
0.748328053502
{'countvectorizer__token_pattern': "'([a-z ]+)'", 'multinomialnb__alpha': 0.5}
```

The next step of the preprocessing is that I want to go for search Optimization for the better parameter tuning using RandomizedSearchCV. Because

- Generally, we need great computation power to search all the possible combinations of parameters, if we need to tune many parameters.
- RandomizedSearchCV can be used to just search the sample, through which we can have the controlled computation power

The additional parameters are achieved thru number of searches (n_tier) and random_state

```
In [47]: # additional parameters are achieved thru number of searches (n_tier) and random_state
         rdm = RandomizedSearchCV(pipeline, parameters_grid, cv=5, scoring='accuracy', n_iter=5, random_state=1)
```

This time, I get the better score as given below.

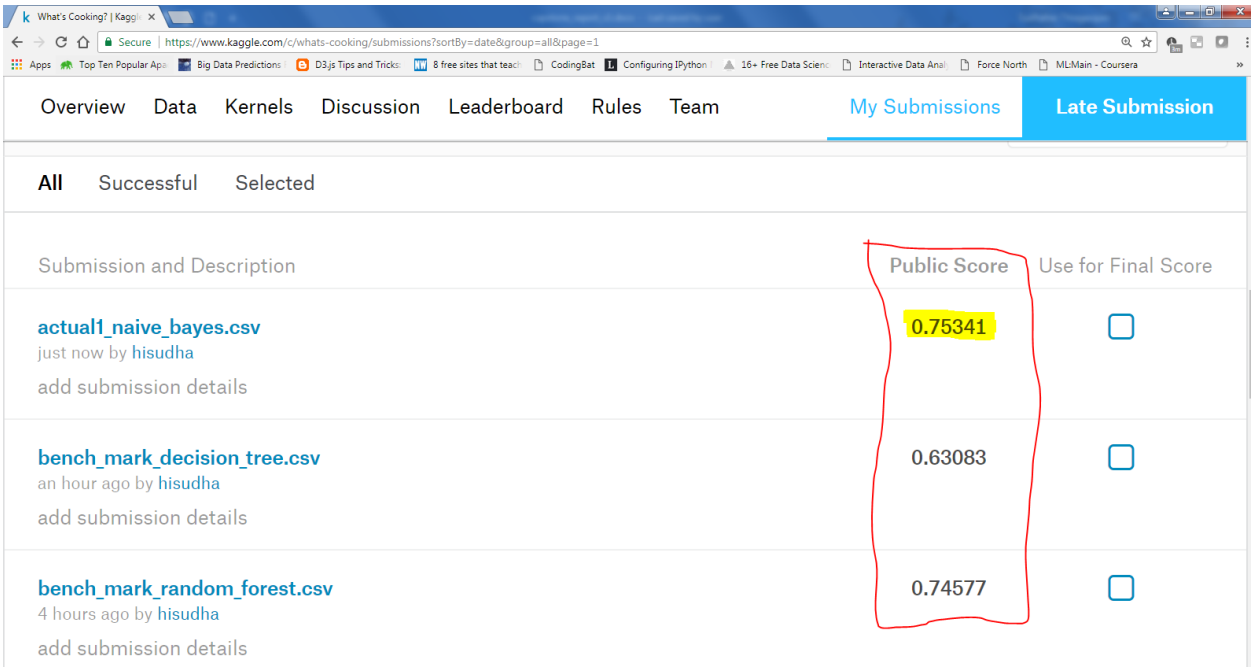
```
In [50]: print(rdm.best_score_)
         print(rdm.best_params_)
```

```
0.751370241867
{'countvectorizer__token_pattern': "'([a-z ]+)'", 'multinomialnb__alpha': 0.30233257263183977, 'countvectorizer__min_df': 2}
```

Now I'm ready for making predictions on the actual Kaggle's test data. Before that let me ch eck the model accuracy and the metrics as described below.

```
Test Accuracy for Naive Bayes  ::  0.8060339409176619
F-Score on Test for Naive Bayes ::  [0.73170732 0.56092843 0.66091954 0.85251919 0.77639752 0.61972864
 0.81105169 0.89589126 0.71976967 0.85911343 0.85125448 0.8606921
 0.87057011 0.94104837 0.77816492 0.66810345 0.69948187 0.69459173
 0.80964153 0.72115385]
[[5.89240974e-18 7.48208712e-30 7.30999588e-15 ... 6.35602347e-15
  1.22933539e-24 4.76621442e-28]
 [1.77925363e-09 1.97908548e-14 1.73674256e-12 ... 7.95308199e-10
  1.84379531e-10 8.95577483e-10]
 [7.42135793e-04 8.50274374e-03 3.72194131e-05 ... 8.29727436e-05
  1.62195684e-05 8.60671909e-06]
 ...
 [7.39623432e-16 8.03029057e-19 1.32200987e-18 ... 6.10036370e-15
  1.85268841e-06 2.71867404e-07]
 [9.07314846e-09 6.10852474e-11 2.45338675e-06 ... 1.98897380e-10
  4.15301325e-04 4.39099569e-07]
 [4.98210665e-07 5.24377121e-08 2.90516130e-06 ... 1.19341226e-05
  7.07634818e-09 5.04177753e-10]]
Log Loss for Naive Bayes ::  0.8503381263441233
```

The test accuracy is lot better compared to the benchmark. Let me create the submission fil e and let me check the Kaggle score.



The Kaggle score is better now than the benchmark.

Challenges:

1. Dataset is not balanced
2. Noise in features, feature selection and feature engineering
3. Computation power
4. Model Evaluation and Refinement

To overcome this, I used various tools and techniques such as CountVectorizer, Pipeline and Transformers.

Used SciPy and FeatureUnion on adding features

I used GridSearchCV and RandomizedSearchCV to overcome performance issues.

I used various metrics and evaluation methods to measure my model on each & every iteration.

Refinement

Next, I want to create the features to a document term matrix using SciPy and the steps are:

- First, I trained the models using the document term matrix and then using the custom created features.
- Now I'm going to train the model on both the feature types
- For this, I have to combine them into a single matrix of features.
- To combine the sparse & dense matrices, I'm using SciPy, which is simple and powerful.

Let me now add the features to a document-term matrix using FeatureUnion.
I created an alternative process that does allow for proper cross-validation, and does integrate well with the scikit-learn workflow. I implemented this process using transformers, FunctionTransformer, and FeatureUnion.

Transformers - Overview:

Transformer objects provide a transform method to perform data transformations. Here are a few examples:

- CountVectorizer
    - fit learns the vocabulary
    - transform creates a document-term matrix using the vocabulary
- Imputer
    - fit learns the value to impute
    - transform fills in missing entries using the imputation value
- StandardScaler
    - fit learns the mean and scale of each feature

    - transform standardizes the features using the mean and scale
- HashingVectorizer
    - fit is not used, and thus it is known as a "stateless" transformer
    - transform creates the document-term matrix using a hash of the token

I created a function that takes a DataFrame & returns the custom created features. I created a stateless transformer from the get_custom function.

```
In [77]: get_custom(train).head()
```

Out[77]:

	no_ingredients	ingredient_len
0	9	12.000000
1	11	10.090909
2	12	10.333333
3	4	6.750000
4	20	10.100000

I executed the function using the transform method. I created a function that takes DF and returns the ingredients string. Finally, I created and tested another transformer.

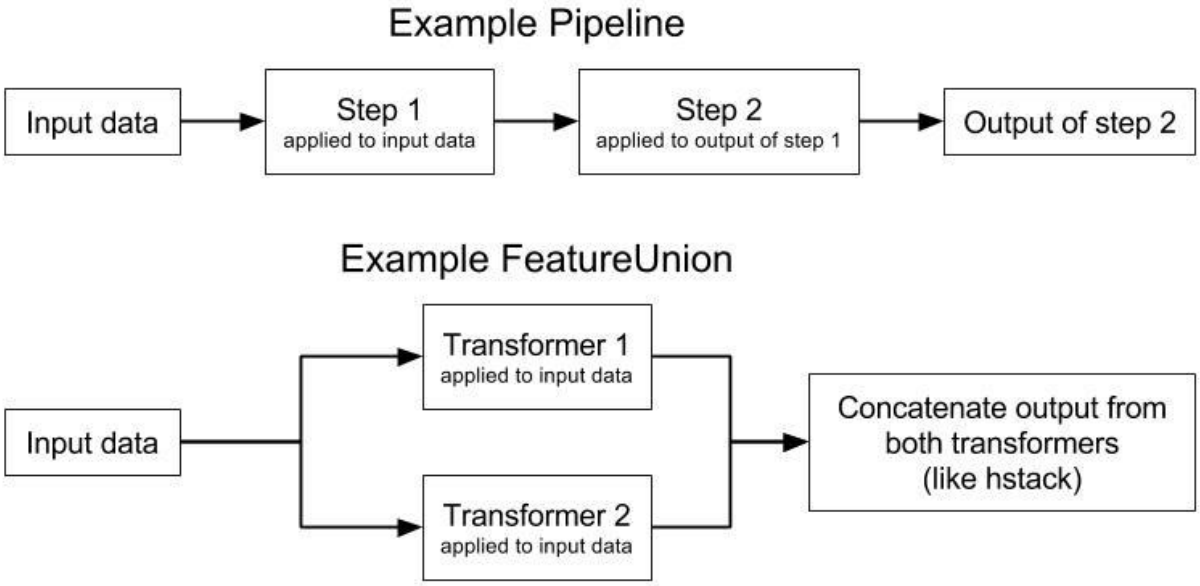
```
In [80]: # execute the function using the transform method
         get_custom_ft.transform(train).head()
```

Out[80]:

	no_ingredients	ingredient_len
0	9	12.000000
1	11	10.090909
2	12	10.333333
3	4	6.750000
4	20	10.100000

The process of combining feature extraction steps

- FeatureUnion applies a list of transformers in parallel to the input data (not sequentially), then **concatenates the results**.
- This is useful for combining several feature extraction mechanisms into a single transformer.



Now I created a document term matrix using the entire training data. I just replicated it as a FeatureUnion by using transformer. I properly combined the transformers into a FeatureUnion.

Now I'm ready for the Ensembling models.

Ensembling models

Rather than combining features into a single feature matrix and training a single model, we can instead create separate models and "ensemble" them.

Ensembling - Overview

Ensemble learning (or "ensembling") is the process of combining several predictive models to produce a combined model that is **better than any individual model**.

- **Regression:** average the predictions made by the individual models
- **Classification:** let the models "vote" and use the most common prediction, or average the predicted probabilities

For ensembling to work well, the models must have the following characteristics:

- **Accurate:** they outperform the null model
- **Independent:** their predictions are generated using different "processes", such as:
  - different types of models

    - different features
    - different tuning parameters
    -
- **The big idea:** If you have a collection of individually imperfect (and independent) models, the "one-off" mistakes made by each model are probably not going to be made by the rest of the models, and thus the mistakes will be discarded when averaging the models.

First, I built the KNN model using default features. Secondly, I built the Naive Bayes model using default features

I ensembled the both the models and I calculated the mean of the predicted probabilities for all rows.

```
In [111]: # calculate the mean of the predicted probabilities for all rows
          new_pred_proba = pd.DataFrame((new_pred_proba_knn + new_pred_proba_rdm) / 2, columns=knn.classes_)
          new_pred_proba.head()
```

Out[111]:

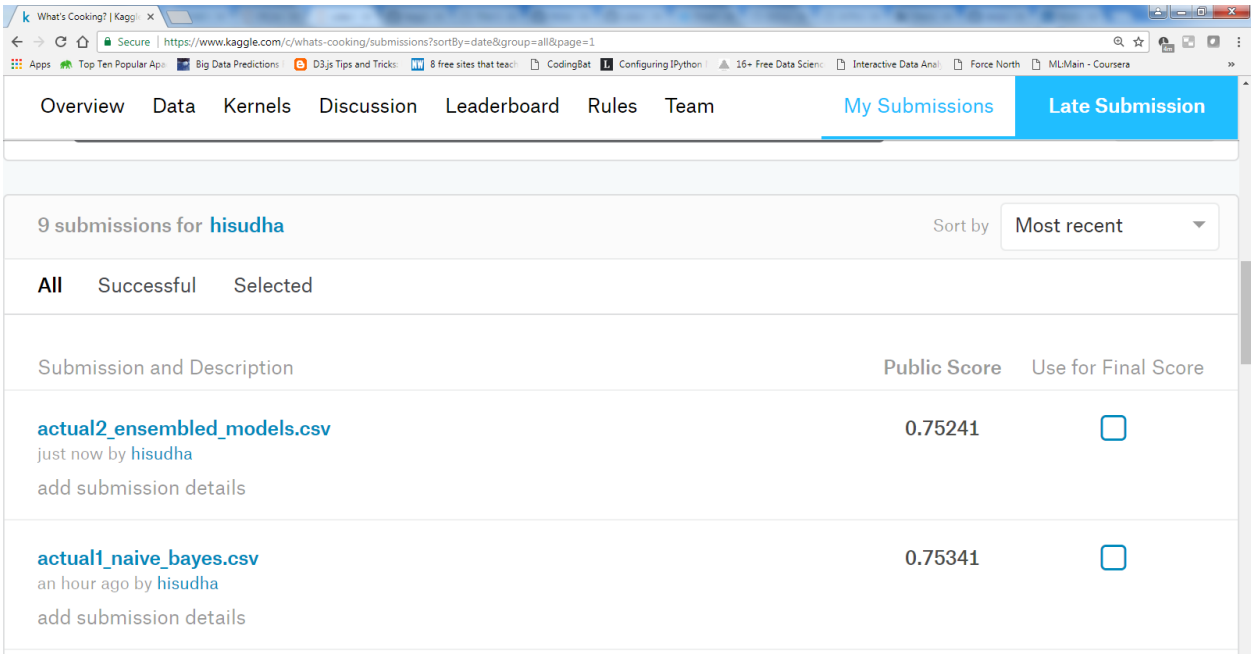
	brazilian	british	cajun_creole	chinese	filipino	french	greek	indian	irish	italian	jamaican	japanese	korean	mexican
0	0.013443	0.269089	0.006900	0.021912	0.018698	0.040685	0.008963	0.037658	0.106674	0.069392	0.005134	0.037780	0.009377	0.08294
1	0.008752	0.011324	0.016875	0.045000	0.018132	0.023884	0.015625	0.046250	0.010629	0.070625	0.005626	0.027501	0.021875	0.06687
2	0.013158	0.009389	0.006951	0.020000	0.015010	0.041365	0.010101	0.029376	0.013372	0.408696	0.005628	0.038752	0.007500	0.08063
3	0.003125	0.004375	0.533750	0.038750	0.001875	0.023125	0.006250	0.075625	0.001250	0.051875	0.011875	0.008125	0.003125	0.10750
4	0.001878	0.009856	0.020097	0.021250	0.003125	0.044922	0.017501	0.013750	0.012547	0.640841	0.003752	0.007500	0.003750	0.08312

Finally, I found the field with the highest predicted probability.

I created the Kaggle submission file.

```
In [113]: # create a submission file
          pd.DataFrame({'id':new.id, 'cuisine':new_pred_proba_class}).set_index('id').to_csv('../data/actual2_ensembled_models.csv')
```

The Kaggle score:



IV. Results

Model Evaluation and Validation

I did various approaches to evaluate the accuracy of predictions of the models, such as

- Estimator score method: Each estimator has a score method to provide a default evaluation criterion.
- Scoring parameter: Model evaluation methodology using cross validation such as cross_validation.cross_val_score, grid_search.GridSearchCV.
- Metric functions: The metrics module implements functions assessing prediction errors. for specific purposes.

Model Evaluation Comparison Table:

Model	Test Accuracy Score	Kaggle Score

Bench Mark - Decision Tree	0.6374	0.6308
Bench Mark - Random Forest	0.7542	0.74577
Naive Bayes	0.8060	0.75341
Ensembled Models		0.75241

When I validated my Kaggle score against the Leaderboard, it comes around top 60%.

Model Justification:

There was not much cleaning as the data was from Kaggle. As explained above, there were lot of noise and outliers, which needs to be handled. I built a general framework to analyze input and output of data through Pandas library. The initial approach I built was to make all texts into lower case, usage of NLTK library, stemming the given words. But it didn't help much. As I stated earlier, the major problem was as some ingredients have more than one words in their designation like "garam masala", "romaine lettuce", "all-purpose flour" and etc.

The benchmark model Decision Tree did the test accuracy as 0.63. Here I just vectorized the ingredients and there was lot of room for tuning this feature.

In my next model (Naïve Bayes), I learned that the model follows that ingredients in an instance were independent of each other. But I understand that this will not help much and a wrong assumption. Because all my data analysis explored that ingredients for a given cuisine are highly correlated. I tried MNB with both GridSearchCV and Pipeline to locate better tuning parameters. By having this implementation, **we can simulate the real world of solutions, where our out of sample data can contain features that were not seen while training the model**.

The next step of the preprocessing is that I want to go for search Optimization for the better parameter tuning using RandomizedSearchCV. This helped me to overcome the challenge of great need of computation power to search all the possible combinations of parameters, if we need to tune many parameters. The additional parameters are achieved thru number of searches (n_tier) and random_state.

Finally, I believe that this model can be trusted for the following reasons:

a. This model performed better that Decision Tree and also ensembled algorithm Random Forest too

b. And also this model performed as good as my own assembling models of KNN & Naïve Bayes.

c. The log loss score was also reasonably good.

d. For the entire test dataset, it took less than minute even in my lap top. In real world scenarios, it can perform well even for the bigger datasets with reasonably bigger co mputation power.

e.

V. Conclusion

Reflection

My approach to is to predict the cuisine category of a given recipe by its ingredients has been broken into various parts such as

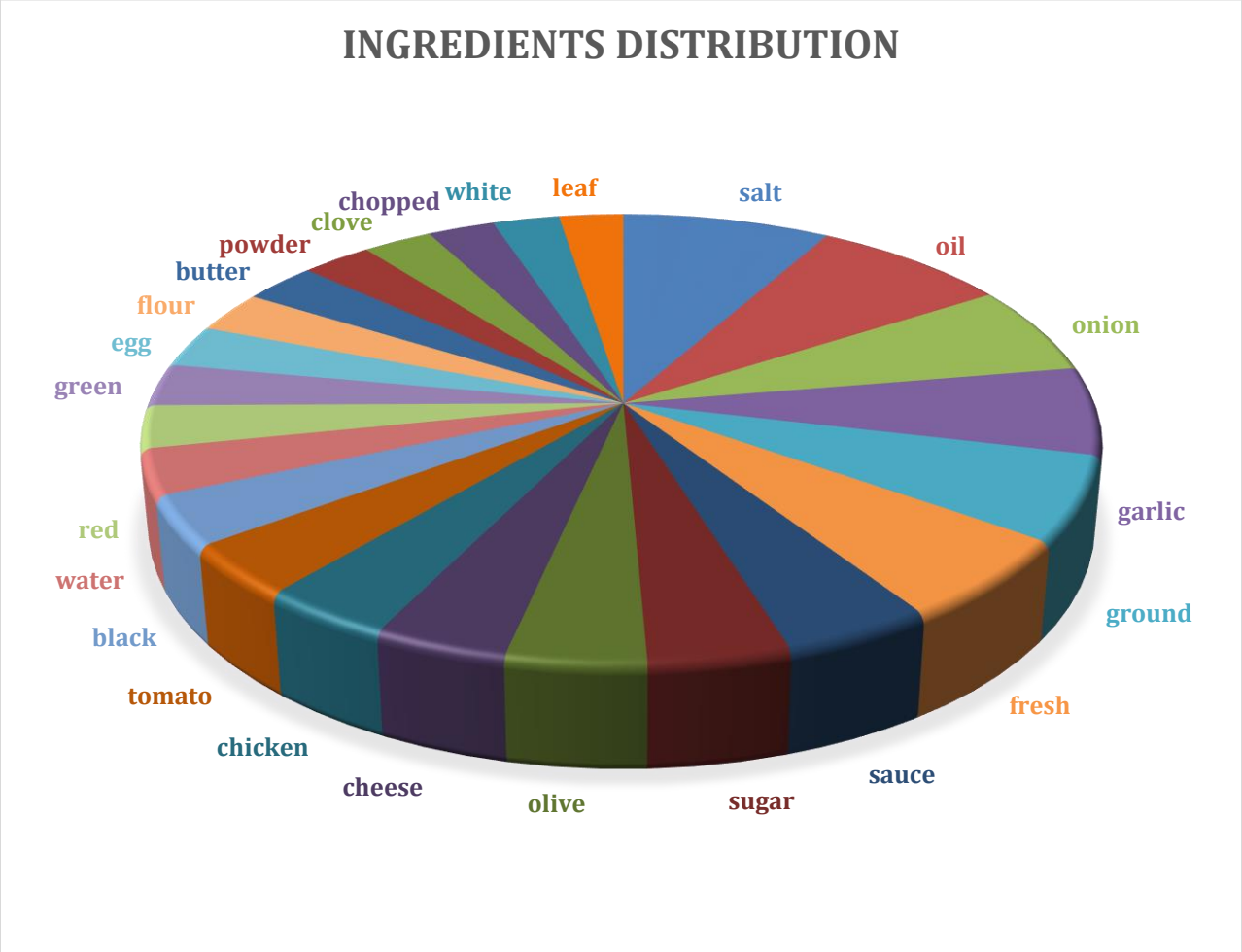
1. Extract the data set from Kaggle
2. Exploratory Visualization
3. Benchmark Models
4. Adding custom features
5. Optimization using various tools & techniques
6. Built model based Naïve Bayes
7. Created Transformers
8. Built the Nested Pipeline's Grid search
9. Built KNN model using only custom features & Naive Bayes model using default features
10. Ensembling models

Feature Importance:

An important task when performing supervised learning on a dataset like the text data I worked here is determining which features provide the most predictive power. By focusing on the relationship between only a few crucial features and the target label we simplify our understanding of the phenomenon, which is most always a useful thing to do.

As per my data analysis explained earlier, the noise in ingredient names will lead to inaccurate predictions. For example, some ingredient names contain adjectives and unit information and some others contain special characters.

We can easily understand the complex distribution of ingredients, which results as an impo rtant task when performing text classification on this dataset. I did detailed study here is determining which features provide the most predictive power. By focusing on the relation ship between only a few crucial features and the target label I simplified the understanding of the phenomenon, which is most always a useful thing to do. In the case of this problem, how can a small number of feature sets of ingredients that can result in strong predictor for cuisine categorization.



To resolve this, I defined the regex pattern for tokenization and tried to use the methods like feature_importance_. This function ranks the importance of each feature when making predictions based on the chosen algorithm.

Feature Selection:

With less features required to train, we might expect that training and prediction time is much lower but at the cost of performance metrics. From the figure above, we see that the top 10 most ingredient features contribute more than half of the of all ingredient features. This compels that we should reduce the feature space and make more robust insight to be f ed into the model to learn. So I used the various metrics to choose the features carefully aft er various trials & tests.

Feature Engineering:

And, I used various tools & techniques to improve the model performance, such as

- Using CountVectorizer, convert a collection of text documents to a matrix of token counts
- Using a Pipeline, automate the workflow by chaining many transformers together.
- Optimize hyperparameters using RandomizedSearchCV and GridSearchCV
- While having lot of parameters to be tuned, it's tough or even not feasible to search all possible combinations of parameter values.
- Transformers - For data transformations, the transformer objects are widely used in my solution
- Using SciPy and FeatureUnion on adding features Ensembling models.

Interesting Observations:

a. The Russian recipes are highly misclassified
b. Asian cuisines often share common ingredients.
c. Ingredient adjectives ("small", "units") impacted the classification accuracy in big way and Etc.

What I Learned:

- ✓ Throughout the project, I made different custom feature sets along with default feature set. And., I t analyzed and computed them with different models such as Decision Tree, Random Forest, Naive Bayes and K nearest neighbor.
- ✓ The result shows that Naïve Bayes yielded the best performance.
    - Even though it has the highest accuracy, the performance is almost same as KNN or Random Forest.
- ✓ I mainly learned about the impact of correlation of the classes. Because the correlation reduces the accuracy of the algorithm.
- ✓ Especially the training set is not uniformly distributed, and I learned how to balance them, because just duplicating the classes with lesser size of instance size won't help to achieve the better results
- ✓ The biggest challenge is to pre-process the ingredients because of the presence of special characters, Brand names, spelling mistake, numeric and etc. (Highly Noisy)
- ✓ I learned various tools and methodologies to pre-process the data, tuning hyperparameters such as
    - Pipeline – to automate the workflow.
    - RandomizedSearchCV and GridSearchCV (Optimize hyperparameters)
    - Transformers - For data transformations
    - SciPy and FeatureUnion - adding features
    - Ensembling models & Model stacking

Improvement

I believe that there are many ways and methodologies to take this further.

1. Better Feature Engineering
    a. Better text processing methods, e.g. Tfidf vectorizer for text data
    b. word stemming for ingredients
2. ML Algorithms
    a. XGBoost
    b. SVC with OVR mode
    c. Sequential from Keras (Neural Net Classifiers)

Further **improvements** can be done on the following areas.

- Investigating the reasons for the wrong classifications
    - Is this because of simplistic ingredients?
    - Common ingredients between the cuisines?
- Advanced preprocessing of ingredient strings
- Locating groups of similar cuisines
- Model stacking
    - The outputs from one level are used as inputs to another level.
    - In this case, we can create a model that will predict the cuisine group for a recipe. Within each of the pre-defined groups, we can create another model that predicts the actual cuisine.
    - The different models can be tuned differently for maximum accuracy. End of the day, the final process will be more accurate than a single-level model.

References:

http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.make_pipeline.html
http://scikit-learn.org/stable/modules/generated/sklearn.grid_search.GridSearchCV.html
http://scikit-learn.org/stable/modules/generated/sklearn.grid_search.RandomizedSearchCV.html
http://docs.scipy.org/doc/scipy/reference/stats.html
http://docs.scipy.org/doc/scipy/reference/sparse.html
http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.FunctionTransformer.html
http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.make_union.html
http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html
http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.FeatureUnion.html
http://scikit-learn.org/stable/modules/ensemble.html#votingclassifier
https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor/