Final Project Report - What's Cooking?

(Kaggle Competition)

SUBMITTED BY:

Anubhav Bhatti

Student ID: 20169150

STAT 857 Statistical Learning- Winter 2020

Final Project

Table of Contents

1 What's Cooking? (Kaggle Competition) 3
1.1 Background and Objective 3
1.2 Data Description 3
1.3 Exploratory Data Analysis and Data Pre-Processing 3
1.3.1 EDA 3
1.3.2 Data Pre-Processing (Data Cleaning) 5
1.3.3 Count Vectorizer and TF-ID Vectorizer 6
1.4 Model Training and Hyperparameter Tuning 7
1.4.1 K-Nearest Neighbors, Decision Trees, and Random Forest with 10-fold Cross-Validation .. 8
1.4.2 Model Training with TFID Vectorizer 9
1.5 Best Model 13
1.5.1 Score Table 14
1.7 References: 15

1 What's Cooking? (Kaggle Competition)

1.1 Background and Objective

What's Cooking is a competition hosted by Kaggle. In this competition, we need to predict the category of a dish's cuisine based on the dish's ingredients. The dataset is provided by Yummly. The provided dataset includes recipe id, the type of cuisine, and the ingredients required for cooking that recipe. The data is stored in JSON format. The objective of this competition is to train a machine learning algorithm that can predict the type of cuisine by using the ingredients of that cuisine. Looking at the problem statement, we can understand that the given task is a Multi-class classification i.e., there are more than 2 categories to predict. This multi-class classification would require text processing and analysis.

Based on my preliminary analysis, I think that classification algorithms like SGD classifier, Logistic Regression, and Neural Networks would be a good choice of algorithms for predicting multiple categories. Also, we can use SVMs to predict the classes by converting the given problem into one vs all classes problem.

1.2 Data Description

As mentioned earlier, the dataset is provided by Yummly has different recipes in different rows. The training dataset has 39774 different recipes, while the test dataset has only 9944 different recipes. Also, the dataset has three attributes:

1. Id: This is a unique identifier for each recipe.

2. Cuisine: This column contains the cuisine type of each recipe. This is our target variable, as we need to predict this in the test dataset.

3. Ingredients: This column contains a list of ingredients required for each dish. We would use this column to train our machine learning algorithm and predict target variable.

1.3 Exploratory Data Analysis and Data Pre-Processing

1.3.1 EDA

Now, let's jump onto the exploratory data analysis and try to identify important patterns and extract valuable insights from the dataset. These insights would help us design our strategies to pre-process the text given in the column 'ingredients'. Since, the dataset did not have any missing or null values so, we don't have to deal with any missing values.

There are 20 unique cuisines in the training dataset. These cuisines along with their distribution, in terms of count, is shown in the figure below:

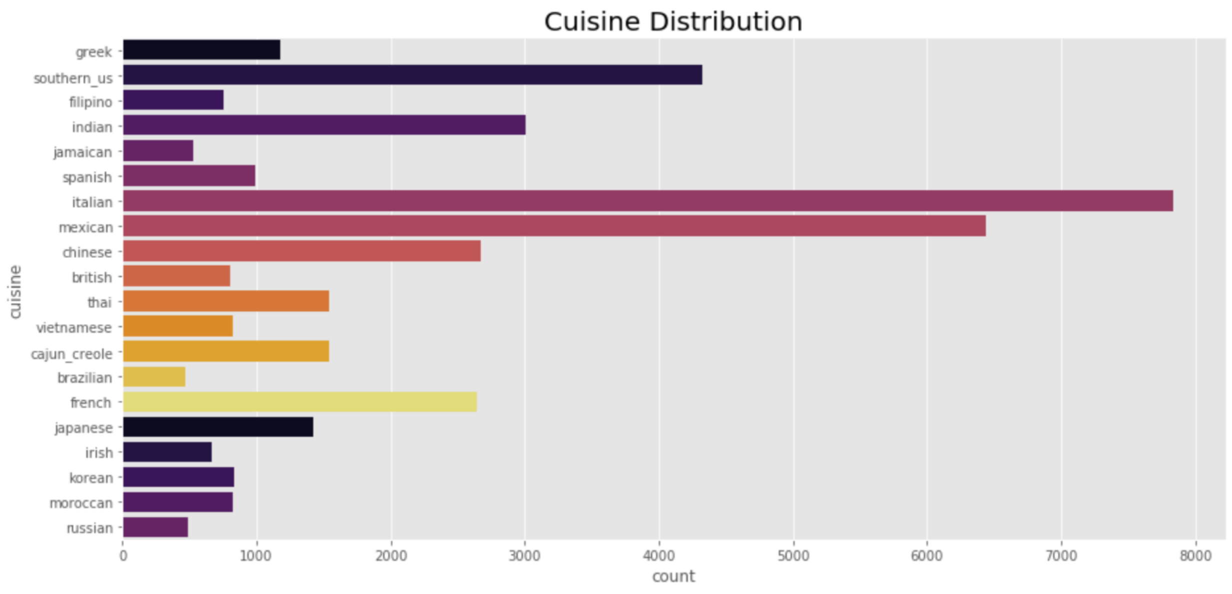


Fig. 1: Bar graph showing distribution of Cuisines vs Count

We can see from the above bar graph that majority of the recipes given in the training dataset belongs to either ‘Italian’ or ‘Mexican’ cuisines. Hence, we can conclude that our dataset is imbalanced. Now, let us look at the distribution of ingredient lengths.

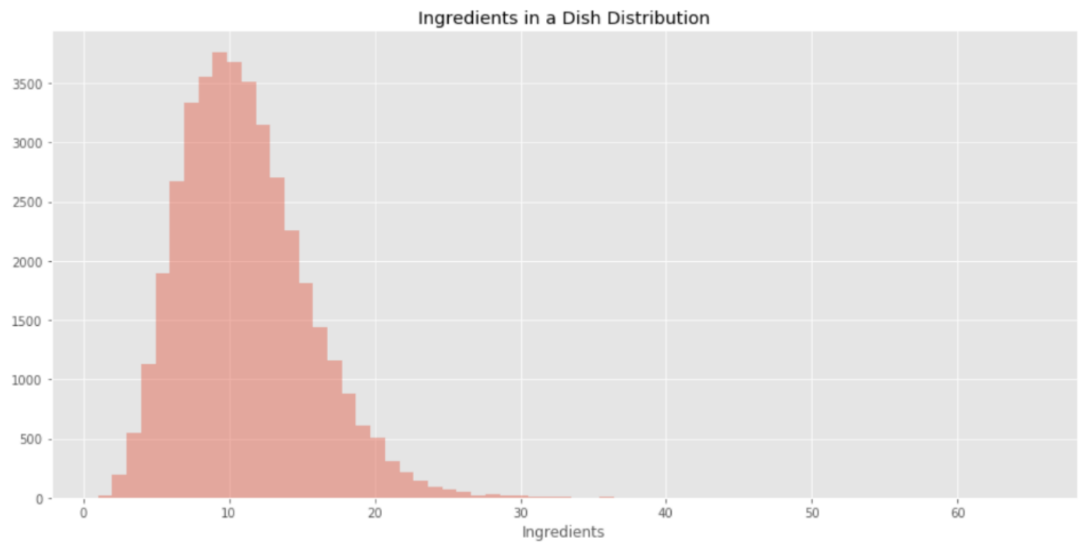


Fig. 2: Bar graph showing distribution of Ingredients vs length

Before doing some more analysis, let us first perform the data cleaning on the train and test dataset to remove unwanted strings and special characters (discussed in next section).

After the data cleaning, let us check what are the top ingredients in all cuisines.

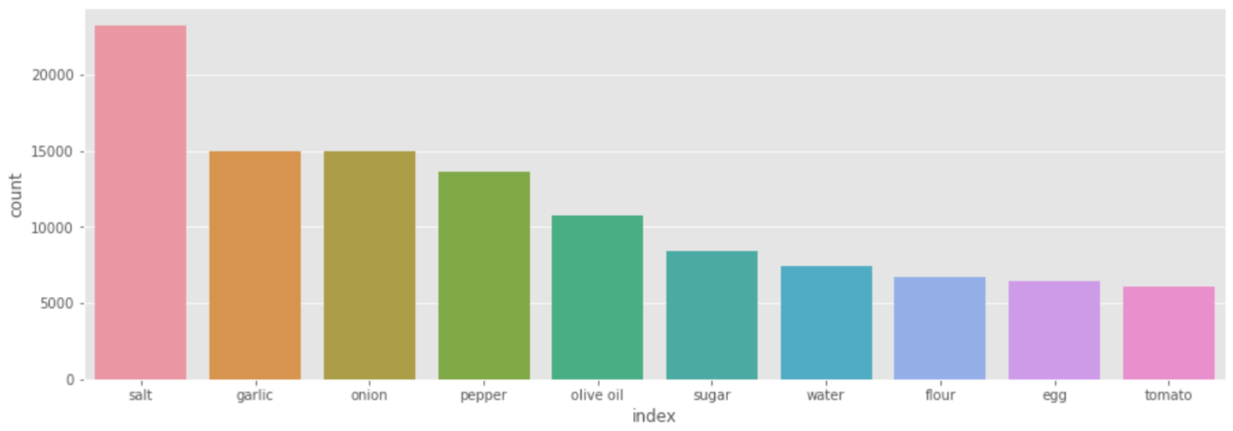


Fig. 3: Bar graph showing distribution of top 10 ingredients from all Cuisines

1.3.2 Data Pre-Processing (Data Cleaning)

We can divide our data cleaning into 8 categories. They are listed below:

1. Special Character Exceptions: After going through the train and test datasets, I found that there are certain special words/characters that we would need to remove first before applying any other data cleaning because then we would not be able to spot those special words/characters in the datasets. Some of these special words are '\xada', 'Ã£', etc.

2. Certain Phrases: In the train and test datasets, I identified that there are some words that have similar meaning but are written in different forms e.g., "sun dried" and "sundried" are same. Hence, we can replace one of to another so that there is no discrepancy in the datasets.

3. Handling Plural Words: In the train and test datasets, there are so many ingredients with their plural forms. Hence, we identified some of those and replaced those since they mean the same ingredients. For example, "drumsticks" and "drumstick".

4. Brand Names: This step was very important because the other text processing techniques can not remove brand names associated with the ingredients in the datasets. I could not identify any other automated method to identify and remove them so, I did it manually. Fortunately, I identified one kernel [1] that had mentioned almost every brand name in the datasets; however, some of them were still missing, so I added them in the code. Some of the brand names associated with the ingredients are kraft, and Tabasco.

5. Unnecessary Keywords: There were a lot of unnecessary keywords associated with the ingredients in the train and test datasets. These keywords could be removed without the loss of any important information. Some of the examples of these keywords are "drained and chopped", "thawed", "firmly packed", etc.

6. Measurements: I also removed the measurements associated with different ingredients.

7. Phrases with Similar Meaning: There are a lot of ingredients mentioned with different names but can be replaced with the one main ingredient e.g., “green onion”, "red onion", and "purple onion" can be replaced with “onion” since they are the same ingredients.

8. Usual Special Characters: Finally, removing the usual special characters in the datasets with space or “no space”.

1.3.3 Count Vectorizer and TF-ID Vectorizer

Both Count Vectorizer and TFID Vectorizers are the methods for converting the textual data into numerical feature vectors such that these vectors having numerical values can then be used to model machine learning algorithms. Count Vectorizer is a simple vectorizer that converts the textual data into vectors by counting the number of times a word has appeared in the document. Hence, Count Vectorizers are not suitable for a dataset with imbalanced word counts as it gives bias to the most frequent words, ultimately ignoring the rare words.

On the other hand, TFID Vectorizer also converts the textual data into vectors but instead of taking the count of words into account, it considers overall weightage of the word in the document and ultimately helps us to handle rare words present in the documents as well. This could be the main reason why Count Vectorizer did not perform well on this dataset while we could get a way better Kaggle score with TFID Vectorizer.

1.3.3.1 *Count Vectorizer*

For training our model with Count Vectorizer, I first separated each ingredient by a pipeline i.e., '|' and then for the ‘token_pattern’ parameter of the Count Vectorizer used the pipeline as the separator. By doing so, the count vectorizer considered each ingredient as an individual word (i.e., it considered ‘black olives’ as one ingredient instead of considering two different ingredients ‘black’ and ‘olives’). For the parameter ‘vocabulary’ I used the complete set of ingredients present the training dataset. I made the ‘binary’ parameter as True since, I needed that all non-zero ingredient count should be labeled as 1. I did not use any other parameter such as “stop_words” or “ngram_range” as they were not required for the current problem.

With these parameters, I created a vectorized matrix with columns as vectorized features and rows as the number of dishes. Then instead of using this matrix for modelling, I created a sparse matrix from the vectorized matrix by using library csr_matrix from Scipy Sparse. The csr_matrix converts the vectorized matrix into sparse matrix this makes the processing and training very fast.

After training a random forest with best grid searched parameters, I could only score 0.69569 on Kaggle. This score was not satisfactory. So, I decided to move to TFID Vectorizer.

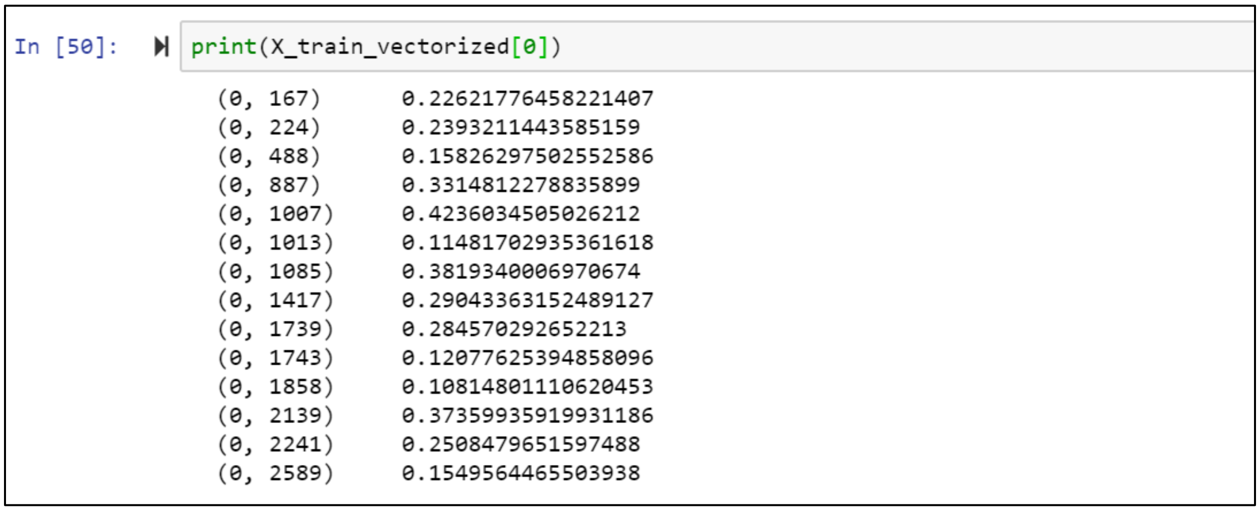
1.3.3.2 *TFID Vectorizer*

For training with TFID Vectorizer, I used the default parameters of the vectorizer with default value of ‘token_pattern’ and the default value range of parameter ‘ngram_range’ i.e, (1, 1). Default values of the ‘token_pattern’ and ‘tokenizer’ means that my ingredients would be divided on the basis of each word.

Also, ‘ngram_range’ of (1, 1) means that only unigrams would be extracted from the ingredients list. This does make sense since we do not need n-grams for this problem statements. For TFID vectorizer, I did not use any stop words as well. Also, very important to mention, I have taken the ‘binary’ parameter as True, this does not mean that the outputs will only have 0/1 values, instead only the tf-term (i.e., term-frequency) in the tf-idf would be binary. The parameters of TFID Vectorizer are mentioned below:

```
TfidfVectorizer(analyzer='word', binary=True, decode_error='strict',
                dtype=<class 'numpy.float64'>, encoding='utf-8',
                input='content', lowercase=True, max_df=1.0, max_features=
None,
                min_df=1, ngram_range=(1, 1), norm='l2', preprocessor=None
,
                smooth_idf=True, stop_words=None, strip_accents=None,
                sublinear_tf=False, token_pattern='(?u)\\b\\w\\w+\\b',
                tokenizer=None, use_idf=True, vocabulary=None)
```

After applying the vectorizer, my training dataset’s first entry looked like this:



```
In [50]:    print(X_train_vectorized[0])

              (0, 167)      0.22621776458221407
              (0, 224)      0.2393211443585159
              (0, 488)      0.15826297502552586
              (0, 887)      0.3314812278835899
              (0, 1007)     0.4236034505026212
              (0, 1013)     0.11481702935361618
              (0, 1085)     0.3819340006970674
              (0, 1417)     0.29043363152489127
              (0, 1739)     0.284570292652213
              (0, 1743)     0.12077625394858096
              (0, 1858)     0.10814801110620453
              (0, 2139)     0.37359935919931186
              (0, 2241)     0.2508479651597488
              (0, 2589)     0.1549564465503938
```

It can be observed that only the row and feature column where the value is ‘True’ is stored in the matrix along with its weighted value.

After this, I label encoded my target variables for my machine learning algorithm using the Scikit-learn’s Label Encoder library package and then I used the Scikit-learn train_test_split library to split my train dataset into train and validation set. By performing these operations on my train and test dataset, now my dataset was ready to be used for training machine learning algorithms.

1.4 Model Training and Hyperparameter Tuning

Now for predicting the target variable, I trained different machine learning models for classification such as Logistic Regression, SVC, SGD Classifier, Neural Networks, Random Forest, Decision Trees, and K-Nearest Neighbors.

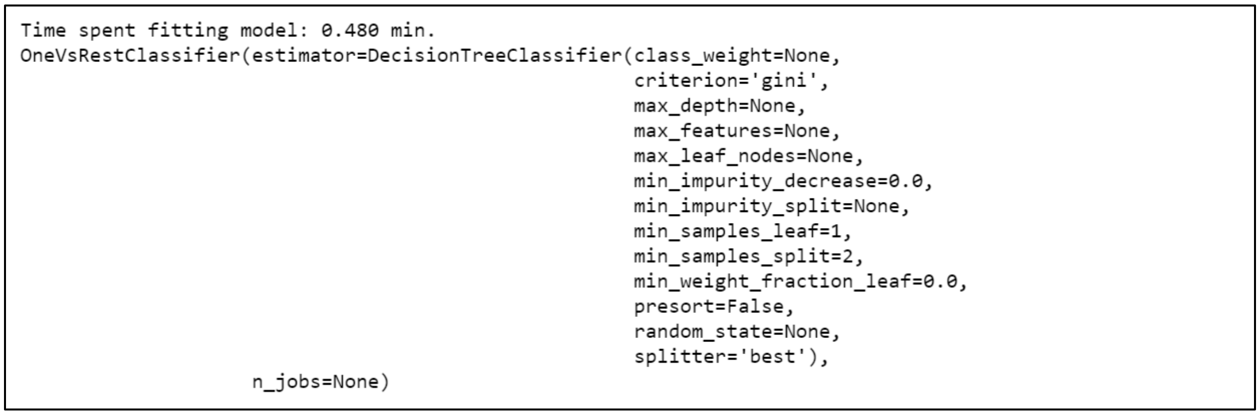
1.4.1 K-Nearest Neighbors, Decision Trees, and Random Forest with 10-fold Cross-Validation

At first, I trained my KNNs, Decision Tree, and Random forest with 10-fold cross-validation on the training dataset prepared by Count Vectorizer.

KNN is a simple and very easy to train supervised machine learning algorithm. In fact, there is no training at all for KNN since, they directly memorize each of the datapoint. When a new datapoint is added to the dataset for prediction, they simply predict the target label of this datapoint based on its neighbors determined by Euclidean Distance (defined by parameter “n_neighbor”). Usually, the value of “n_neighbors” are chosen to be an odd value so that there are no ties while predicting.

For training the KNN model, I used an arbitrary value of parameter “n_neighbors” as 5 and performed a 10-fold cross-validation on the training dataset. After performing the cross-validation, I could get validation accuracy of 69%. I expected KNN model to perform a little better on the validation set as I thought that the related ingredients would belong to the same cuisines.

Decision Trees are also a simple to implement supervised machine learning algorithm. This algorithm uses a tree-like graph for making simple decision rules for predicting the target variable. Since we had a multi-class classification problem, for training the decision trees, I used the decision tree classifier in combination with the “OnevsRestClassifier” from scikit-learn package. The “OnevsRestClassifier” fits a single classifier per target label. The parameters used for training the model are shown below:



```
Time spent fitting model: 0.480 min.
OneVsRestClassifier(estimator=DecisionTreeClassifier(class_weight=None,
                                                     criterion='gini',
                                                     max_depth=None,
                                                     max_features=None,
                                                     max_leaf_nodes=None,
                                                     min_impurity_decrease=0.0,
                                                     min_impurity_split=None,
                                                     min_samples_leaf=1,
                                                     min_samples_split=2,
                                                     min_weight_fraction_leaf=0.0,
                                                     presort=False,
                                                     random_state=None,
                                                     splitter='best'),
                    n_jobs=None)
```

As expected, the decision tree classifier with ‘one vs rest’ classifier fit the training model very well. After training the model, I got the validation accuracy of 91.7%. However, when I predicted the target labels on the test dataset, the decision trees could give me Kaggle score of 0.60166 (with TFID Vectorizer, refer the main code file named, “EDA, Data Pre-Processing, and Modelling (Best)”).

Random Forest classifier also performed well on the training dataset. After the training it on the training dataset with 10-fold cross-validation, the validation accuracy was around 93.18%. However, on the test dataset, the algorithm did not perform as expected (even with the tuned hyperparameters) and I could only get a Kaggle Score of 0.58628.

To further inspect the reason behind such a low accuracy, I checked the dataset and predictions. I observed that some to the cuisines have very similar ingredients e.g., British and Irish cuisines have mostly similar ingredients, while Southern US and Mexican cuisines also shared some ingredients. Since, there were a lot of cuisines with very similar ingredients, it would have confused the KNN, DT and even the Random Forest algorithm; hence, leading to a very low validation accuracy.

To improve the score, as already mentioned, I used the TFID Vectorizer instead of Count Vectorizer with other machine learning algorithms.

1.4.2 Model Training with TFID Vectorizer

With TFID Vectorizer, I trained the following models:

1.4.2.1 *Logistic Regression*

Logistic Regression is a classification algorithm that transforms its outputs using logistic sigmoid function to return a probability value and hence, the target label is predicted based on the highest probability. For training the logistic regression, firstly, I transformed the data into a sparse matrix using TFID Vectorizer (already discussed). Then, I split the training dataset into train and validation set by using "train_test_split" library from scikit-learn package. After that, I used the Logistic Regression classifier to fit on the training dataset with arbitrary parameters i.e., C = 10. The 'C' parameter in Logistic Regression in called Inverse regularization parameter i.e., lower value of 'C' would represent high value of Lambda regulator. So, for a start I choose the value to be 10 (default value is 1).

By training the classifier on my train data set, I got a validation accuracy of ~79.34%. Then I used the same parameters to make the prediction on my test dataset. I got the Kaggle score of ~78.75. This score was way better than DT, KNN and Random Forest in the above section.

For improving this model, I performed a grid search with the following parameters:

```
LR_clf = LogisticRegression()

grid = {'penalty' : ['l1', 'l2'], 'C': np.logspace(-4, 4, 20), 'solver' : ['liblinear']}

Gridsearch_LR = GridSearchCV(LR_clf, param_grid = grid, cv = 3, verbose=3, n_jobs=-1)

LR_grid = Gridsearch_LR.fit(X_train_vectorized, y_transformed)
```

Since, I wanted to apply a stronger regularization; hence, I choose 20 values of 'C' on the log space by using 'np.logspace'. After completion of grid search, the best parameters came out to be:

```
LR_best_parameters {'C': 4.281332398719396, 'penalty': 'l2', 'solver': 'liblinear'}
```

By predicting the test target labels using this model parameters, I could get a Kaggle Score of 0.78841. This was almost equal to logistic regression classifier with 'C' equals to 10.

1.4.2.2 *SVC – Support Vector Classification*

For my second model, I used the Support Vector Classification (SVC) from the Scikit-learn package. The implementation is based on 'libsvm'. For training, I used the following parameters:

```
# Training with random parameters

clf2 = SVC(C=100, gamma=1, kernel='rbf')
clf2.fit(X_train , y_train)
clf2.score(X_test, y_test)
```

Same as Logistic Regression, the 'C' parameter represents the regularization parameter. Along with it, I also changed the parameter 'gamma' to 1. Gamma represents the kernel coefficient for the kernel 'rbf'. On the train and validation dataset, I could a validation accuracy of ~81% while, as expected, SVC classifier performed better than the previous algorithms and I got a Kaggle score of 0.80812, with these arbitrary parameters. To improve the model, I performed a grid search with the following parameter grid:

```
SVC_clf = SVC()

grid = {'C':[1,10,80,100],'gamma':[1,0.1], 'kernel':['linear','rbf']}

Gridsearch_SVC = GridSearchCV(SVC_clf, param_grid = grid, cv = 3, verbose=3, n_jobs=-1)

SVC_grid = Gridsearch_SVC.fit(X_train , y_train)
```

The best parameters after the grid search came out to be:

```
SVC_best_parameters {'C': 10, 'gamma': 1, 'kernel': 'rbf'}
```

With these optimized hyperparameters, I could achieve a Kaggle score of 0.80842, which is slightly better than the arbitrary parameters.

1.4.2.3 *SGD Classifier*

According scikit-learn, SGD Classifier belongs to a class of Linear Classifiers (SVM) with Stochastic Gradient Descent training. This classifier implements regularized linear models with stochastic gradient descent (SGD) learning. For training my SGD classifier, I used the default parameters and got a Kaggle score of 0.77373 on test dataset.

For improving my model, I performed the hyperparameter tuning of 'alpha' with a grid search. I used the following grid parameter:

```
SGD_clf = SGDClassifier()

grid = {'alpha': [4e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2, 1e3], # learning rate
        'max_iter': [1000], # number of epochs
        'loss': ['hinge'],
        'penalty': ['l2'],
        'n_jobs': [-1]}

Gridsearch_SGD = GridSearchCV(SGD_clf, param_grid = grid, cv = 3, verbose=3)

SGD_grid = Gridsearch_SGD.fit(X_train , y_train)
```

With the grid search, the best parameters came out to be:

```
SGD_best_parameters {'alpha': 4e-05, 'loss': 'hinge', 'max_iter': 1000,
                     'n_jobs': -1, 'penalty': 'l2'}
```

With these optimized hyperparameters, I could achieve a Kaggle score of 0.78077, which is slightly better than the arbitrary parameters.

1.4.2.4 *Neural Networks*

Neural Networks are powerful machine learning classifiers. They are set of algorithms, that recognizes patterns. For this problem, I used 'Keras', an open source neural network python library that is capable of running on top of TensorFlow. To predict the target labels, I trained Six different neural networks with different architecture i.e., with different nodes, hidden layers, Dropout values, Regularization values, and Early Stopping. However, the performance I could get was from the simplest model i.e., neural network 1. Since, GPU was required to train the nets, I used Google's Collaboratory for training and prediction.

Let's discuss the various architectures that I used in more detail below.

Neural Network 1: In this NN, I used 5 hidden layers with 256 node units at first layer, 512 node units for next four layers, and fully connected layer with 20 node units (equal to number of cuisines). For each first four layers, I used 'ReLU' i.e., Rectified Linear Unit, as the activation function. 'ReLU' greatly improves the computation speed since, the out this function is either zero or one based on the sign of the variable. For the last dense layer, I used 'Softmax' as the activation function. Along with these layers, I also included, Dropouts of 0.5 after every layer except the last fully connected layer i.e., drop every node on each layer with the probability of 0.5. Including dropouts greatly reduces the problem of 'Overfitting' in neural networks. For compiling the model, I used 'categorical cross entropy' as the loss function and 'adam' with default learning rate as the optimizer. While fitting, I used the validation split of 80% training and 20% validation and set the epochs to 100 with a batch size of 95.

Neural Network 2: For this Neural Network, I increased the hidden layers to 6 with 256 units at the first layer, 512 units for next two layers, 1024 units for the fourth and five hidden layers, and finally a fully connected dense layer with 20 units. I kept all the other parameters same as the previous neural network. The motive was to make the neural network more complex with more nodes/weights and hidden layers.

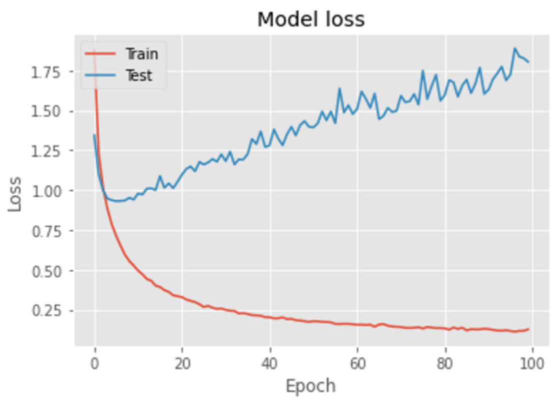
However, the Kaggle score on the test dataset did not improve after increasing the complexity. The major reason for this would be that the neural networks are overfitting.

Neural network 3: Since, the earlier neural networks are overfitting, I tried to increase the dropouts after every layer to drop more nodes at each layer. This would try to reduce the over fitting of the neural networks. However, the performance of these neural networks did not improve.

Neural Network 4 and 5: For model 4 and 5, I introduced a 'L2 Regularizer' with the 'l = 0.001' to further reduce the overfitting of the model. By including the L2 Regularizer, the training vs test loss graph was smoothened slightly (fig. 5); however, the performance of the neural networks still could not surpass the first neural network.

Neural Network 6: By observing the training loss vs test loss graph of above Neural networks, I could understand that the difference between the training and test loss increased significantly after 20 epochs. So, I introduced the 'Early Stopping' from Keras callbacks package. This allowed me to monitor 'validation loss' (referred to as test loss above) and when it did not decrease after 4 epochs it will end the training to avoid further overfitting.

Neural Network 2



Neural Network 3

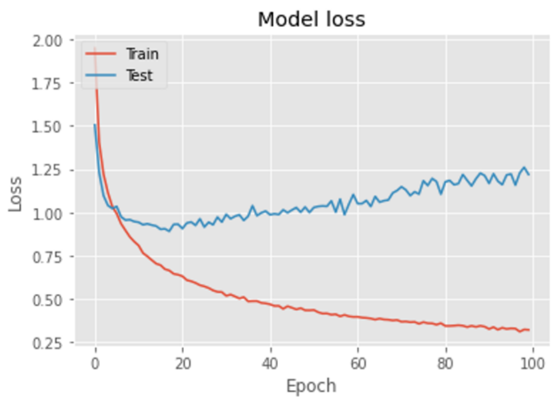
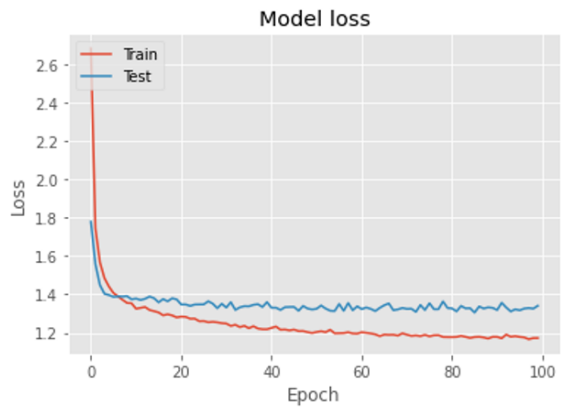


Fig. 4: Graph of Neural Networks 2 and 3's Training Loss vs Test Loss

Neural Network 4



Neural Network 5

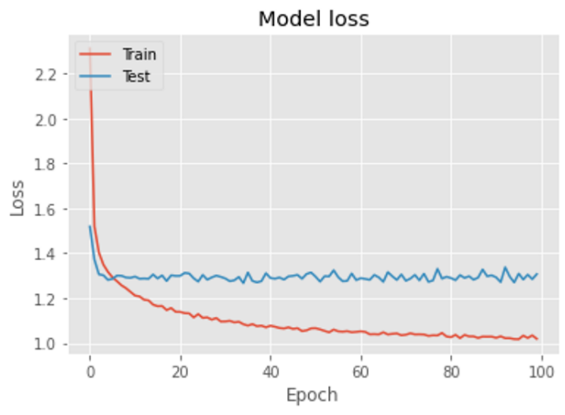


Fig. 5: Graph of Neural Networks 4 and 5's Training Loss vs Test Loss

1.5 Best Model

I could get the best results by feeding my three models i.e., Logistic Regression, SVC, and SGD Classifier to the Voting Classifier from the Scikit-learn package.

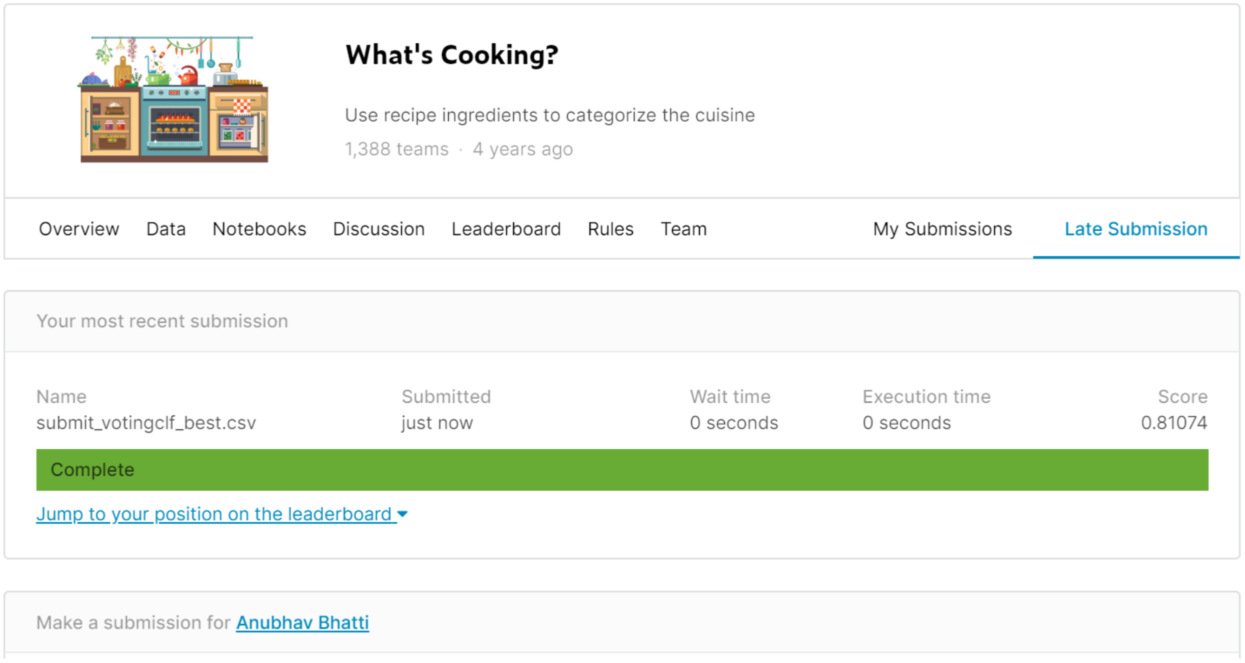
Voting Classifier, fits the clones of the estimators and will output the best prediction on the test dataset based on the parameter 'voting'. It the parameter 'voting' is set as 'Soft' then it will predict the target label based on the argmax of the sums of the predicted probabilities. If it set for 'hard' it will use the predicted class labels for majority rule voting. I used the following parameters for the Voting Classifier:

```
Best_vclf = VotingClassifier(estimators=[('clf1', LogisticRegression(C=10,dual=False, multi_class= 'ovr')),
                                        ('clf2', OneVsRestClassifier(SVC(C= 10, gamma= 1, kernel= 'rbf',
                                                                     probability=True))),
                                        ('clf3', SGDClassifier(alpha = 4e-05, loss = 'log', max_iter = 1000,
                                                              n_jobs = -1, penalty = 'l2', verbose = 1))],
                            voting='soft',weights=[1,2,3])

Voting_clf_best = Best_vclf.fit(X_train_vectorized, y_transformed)

y_predicted = Best_vclf.predict(Result_transformed)
y_predicted_final = encoder.inverse_transform(y_predicted)
predictions = pd.DataFrame({'cuisine' : y_predicted_final , 'id' : test_df.id })
predictions = predictions[[ 'id' , 'cuisine']]
predictions.to_csv('submit_votingclf_best.csv', index = False)
```

With the Voting Classifier, I could get the Kaggle Score of 0.81074. The screenshot of the Kaggle score is given below:


What's Cooking?
Use recipe ingredients to categorize the cuisine
1,388 teams · 4 years ago
Overview
Data
Notebooks
Discussion
Leaderboard
Rules
Team
My Submissions
Late Submission
Your most recent submission
Complete
Jump to your position on the leaderboard
Make a submission for Anubhav Bhatti

Name	Submitted	Wait time	Execution time	Score
submit_votingclf_best.csv	just now	0 seconds	0 seconds	0.81074

1.5.1 Score Table

The Kaggle score table for each model individually are shown below:

Serial No.	Machine Learning Model	Kaggle Score	Submission File Name
1.	Voting Classifier with Logistic Regression, SVC, SGD Classifier [Best Parameters]	0.81074	submit_votingclf_best.csv
2.	SVC Classifier [Best Parameters]	0.80842	submit_SVC_best_grid_params.csv
3.	SVC Classifier [Default Parameters]	0.80812	submit_SVC_random.csv
4.	SGD Classifier [Best Parameters]	0.78077	submit_SGD_best_grid_params.csv
5.	SGD Classifier [Default Parameters]	0.77373	submit_SGD_default.csv
6.	Neural Network 1 [5 Layers]	0.77051	submit_mod_ANN.csv
7.	Neural Network 2 [6 Layers]	0.76528	submit_mod_ANN_1.csv
8.	Neural Network 3 [Increase Dropouts]	0.76448	submit_mod_ANN_2.csv
9.	Neural Network 5 [L2 Regularizer]	0.76981	submit_mod_ANN_4
10.	Neural Network 5 [Early Stopping]	0.76518	submit_mod_ANN_5.csv
11.	Neural Network 4 [L2 Regularizer]	0.76076	submit_mod_ANN_3.csv
12.	Random Forest Classifier [TFID + Best Parameters]	0.75955	submit_RF_best_grid_params.csv
13.	Decision Tree Classifier [One vs Rest Classifier]	0.60166	submit_DT_default.csv

1.7 References:

1. What’s Cooking? Kaggle Competition. Link: https://www.kaggle.com/c/whats-cooking/overview
2. What’s cooking: Link: https://github.com/rfliegerallison/kaggle-whats-cooking/blob/master/What's%20Cooking%20-%20Kaggle%20Competition.ipynb
3. A Walkthrough – EDA + Vizualizations + Unigram Model. Link: https://www.kaggle.com/gloriahristova/a-walkthrough-eda-vizualizations-unigram-model
4. What Is The Rock Cooking?? (Ensembling & Network). Link: https://www.kaggle.com/ash316/what-is-the-rock-cooking-ensembling-network
5. Think Differently What's Cooking. Link: https://www.kaggle.com/ashishpatel26/think-differently-what-s-cooking
6. A Detailed Explanation of Keras Embedding Layer. Link: https://www.kaggle.com/rajmehra03/a-detailed-explanation-of-keras-embedding-layer