What’s Cooking – Categorize the Cuisine using ML Classification

Nayeem Ahmed A S[MCVGRB]

Eötvös Loránd University, Budapest, Hungary

nayeemahmedas@gmail.com

Abstract. In this paper, we describe our methodology for training an algorithm to make such predictions. We have “What’s Cooking” dataset which is taken from Kaggle open contest (Kaggle is an online platform for hosting data science competitions). We perform two tokenization method and applied eight different model in both tokenized data and Compared their respective accuracy.

Keywords: food, recipe, machine learning, classification, scikit-learn

1 Introduction

The competition (https://www.kaggle.com/c/whats-cooking) as mentioned before was hosted on the Kaggle platform. Kaggle arranged the dataset for 20 international cuisines from Yummly. Yummly is a free smartphone app and website that provides recipe recommendations personalized to the individual’s tastes, semantic recipe search, a digital recipe box, shopping list and one-hour grocery delivery. While there were many competitions to choose from, I decided to do this cooking challenge because I was interested in learning natural language processing, which I felt like I hadn't really been able to practice. Although the competition did not require much real language processing, it was still appealing and interesting to think about how different ingredients could be used to predict the cuisine.

I wanted to understand what I was doing with the algorithms I was using and I wanted to make educated decisions about my modeling choices. As a result, I spent lots of time learning by reading and tinkering. I used Scikit learn library to implement data split, preprocessing, tokenization, algorithms and matplot library for visualization.

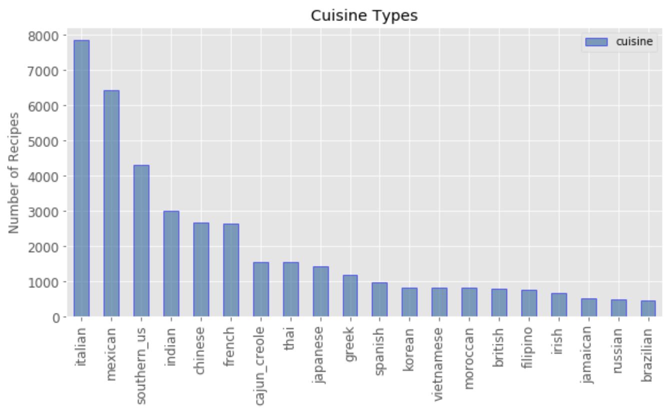
My project is all about implementing 8 classification model using scikit learn apply these 8 models on count vectorized data and tf-idf vectorized data respectively and comparing the result and come up with best model in each vectorizer methods.

2 Data Exploration

The dataset, made available by recipe index Yummly [4] through data science competition host Kaggle [5], consists of 39,774 recipes. Each recipe comprises a list of ingredients, a unique identifier, and a cuisine label. train.json – 39774 records containing recipe id, type of cuisine and list of ingredients. test.json – 9942 records containing recipe id and list of ingredients. Our task is to predict cuisine for a particular dish from test data given its ingredient set. There are 6714 unique ingredients in the dataset. Except the frequent ingredients in specific cuisines, there are some ingredients with high degree of uniqueness like Soy sauce (Asian cuisine), Sesame oil (Asian cuisine), Sake (Japanese cuisine), Garam masala (Indian cuisine), Ground ginger (Moroccan cuisine), Avocado (Mexican cuisine).

20 types of cuisines:

Weight	**Recipe**	**Cuisine**
19.71 %	7838	Italian
16.19 %	6438	Mexican
10.86 %	4320	Southern US
7.55 %	3003	Indian
6.72 %	2673	Chinese
6.65 %	2646	French
3.89 %	1546	Cajun Creole
3.87 %	1539	Thai
3.58 %	1423	Japanese
2.95 %	1175	Greek
2.49 %	989	Spanish
2.09 %	830	Korean
2.07 %	825	Vietnamese
2.06 %	821	Moroccan
2.02 %	804	British
1.90 %	55	Filipino
1.68 %	667	Irish
1.32 %	526	Jamaican
1.23 %	489	Russian
1.17 %	467	Brazilian



3 Data Preprocessing

Text mining and Natural Language Processing techniques could be used to derive useful information from raw data. Created a function names pre_processing in which I perform converting the text case to lower case. Data is often received in irregular formats. E.g. 'Salt' and 'salt' both means the same ingredient.

Within the Pre_processing function created delete_brand function which is helpful to remove all brand names in the rows. Brand name with ingredients will result as differend ingredient though "red gold salt" and "salt" is same. I removed brands names like country crock, red gold, wish-bone, etc.

Created delete_state function within pre_processing function which is helpful to remove all state of ingredient. It is not important for us whether the ingredient is frozen, chopped, ground, etc. we will predict cuisine based on the ingredients. Deal with the punctuation, created an original function to remove all the special character in the data. Python would read 'data mining' and 'datamining' as two different words.

After performing all the above steps on ingredient column (dependent column) I created a new column all_ingredient which I stored all the ingredient as a single string with delimitation using semicolon (;).

Tokenization

I chose two process to set up my training data to extract features from the data.

- **Count Vectorizer**, which converts the words in a document (or in this case recipe) into counts or...

- **tfidf (Term frequency inverse document frequency) vectorizer** to not only count but also return the normalized count based on how many times an ingredient appears in all the recipes

Using scikit learn first I performed count vectorizer on all_ingredient column (dependent values) and using Label Encoder on cuisine type I got y value. In this project I implemented all the models using scikit learn library for which we require X_train and y_train. Where X_train is dependent value pass through vectorizer method and got x_train and by encoding label got cuisine type as y_train. We can conveniently tell the count vectorizer which features it should accept and let him build the matrix with 1s and 0s when ingredients are present in a single step. We see that the vectorizer has retained 3010 ingredients and processed the 40 000 recipes in the training dataset. We can easily access the features to check them using the vectorizers properties.

I found that by tf-idf vectorizing the texts into matrices as inputs are much more efficient. After cleaning up the input data (train.json and test.json) by removing units and some of the non-letter characters such as numbers and whitespaces. It still needs to be configured into a tf-idf matrix before feeding into different models. Instead of using strings as input, I will be using the tf-idf vectorizer from Scikit-learn to configure the strings into tf-idf features as the input. The term-frequency inverse-document frequency (tf-idf), which involves calculating the relative frequency of an ingredient in a particular cuisine compared to all other cuisines. Doing so would reveal the “indicative” ingredients that could be used to, with relatively high confidence, predict the cuisine of a recipe. For example, the ingredient garam masala is an “indicative” ingredient i that is often associated with cuisine c; a higher tf-idf for i for a cuisine c score means that i occurs more frequently in cuisine c than in any other cuisines.

$$\text{tf}(i, c) = \text{number of times ingredient i appears in cuisine c}$$

$$\text{idf}(i, C) = \frac{\text{number of cuisines in C}}{\textit{number of cusine that contain ingredient i}}$$

$$\text{tf-idf}(i, c, C) = \text{tf}(i, c) * \text{idf}(i, C)$$

Split Data:

Using Scikit learn model_selection library imported train_test_split using which we can split X_train and y_train into train and validation form as X_train, y_train, X_validation and y_validation.

Please note that the first paragraph of a section or subsection is not indented. The first paragraphs that follows a table, figure, equation etc. does not have an indent, either.

Subsequent paragraphs, however, are indented.

4 Modeling

The multiclass classification problem can be decomposed into several binary classification tasks that can be solved efficiently using binary classifiers, I will be using some of the scikit-learn modules as follows:

Logistic regression:

Logistic regression is one of the most used algorithms for binary classification for its simplicity and efficiency to implement. It measures the relationship between the labels and features by estimating the probabilities using its underlying logistic function. Fundamentally, it is a linear method but it uses the logistic function to transform the predictions. Scikit-learn has the logistic regression with multinomial built-in. I tried to change the parameter like multi_class as multinomial and solver as lbfgs which supports L2 regularization in logistic regression. But it results very low accuracy where as the default arguments gave me better results.

With the following Default arguments:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

model	Time Taken	Accuracy
Logistic Regression, with count vectorizer	0.182 min	78.6%
Logistic Regression, with tf-idf Vectorizer	0.079 min	78.3%

Random Forest:

Random Forest is an ensemble of a group of decision trees (features) and trains each individual decision tree with different subsets of the training data. Each node of the decision tree is split using a randomly selected feature from the data, so the models created by the algorithm would not be correlated with each other. Eventually, the outliers would be eliminated through majority voting. Random forest is a good model to reduce overfitting and no need for feature normalization. Each individual decision trees can be trained in parallel. It can be used for multiclass classification out of the box. A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.

With the following Default arguments:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=None, max_features='auto',
                       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, n_estimators=10, n_jobs=None, oob_score=False, random_state=None, verbose=0, warm_start=False)

Tuned parameters for better results:
criterion='entropy', max_features=3, n_estimators=1000, n_jobs=2, oob_score=True, random_state=0

Criterion: The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain.
Max_features: The number of features to consider when looking for the best split
N_estimators: The collection of fitted sub-estimators.
Oob_score: Score of the training dataset obtained using anra out-of-bag estimate.
Random_state: If int, random_state is the seed used by the random number generator.
N_jobs: The number of jobs to run in parallel for both fit and predict.

model	Time Taken	Accuracy
Random Forest, with count vectorizer	5.808 mins	75.4%
Random Forest, with tf-idf Vectorizer	5.533 mins	75.4%

XGBoost:

XGBoost is also an ensemble method. It is a start-of-the-art model that became popular not long ago. It is an advanced implementation of gradient boosting algorithm. It is also an ensemble method that creates a strong classification model by stacking "weaker" ones. By adding models on top of each other iteratively, the errors of the previous model are corrected by the next predictor until the training data is accurately predicted or reproduced by the model. It also adds an additional custom regularization term in the objective function. XGBoost is also known as ‘**regularized boosting**’ technique. XGBoost has an in-built routine to handle missing values. Imported XGBClassifier from sklearn and implemented by calling function and fit the train data. Tuning its arguments doesn’t give any better result so used same default arguments.

With the following Default arguments:
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
colsample_bynode=1, colsample_bytree=1, gamma=0,
learning_rate=0.1, max_delta_step=0, max_depth=3,
min_child_weight=1, missing=None, n_estimators=100, n_jobs=1,
nthread=None, objective='multi:softprob', random_state=0,
reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
silent=None, subsample=1, verbosity=1)

model	Time Taken	Accuracy
XGBoost classification, with count Vectorizer	0.875 mins	73.4%
XGBoost classification, with tf-idf Vectorizer	2.063 mins	73.3%

Support Vector Machines

Support Vector Machine is a classification algorithm that uses the support vectors (data points nearest to the hyperplane, which separates and classifies a set of data) to classify instances into two classes (binary). The goal is to choose a hyperplane with the greatest possible margin between hyperplane and any point within the training set, given a greater change of new data being classified correctly. Same Scikit-learn's "one versus all" scheme can be applied on this algorithm to turn this binary classification method into multi-class classification. SVC with parameter kernel='linear', but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples. Imported LinearSVC from sklearn.svm and implemented by calling function and fit the train data.

With the following Default arguments:

```
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
          intercept_scaling=1, loss='squared_hinge', max_iter=1000,
          multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
          verbose=0)
```

Tuned parameters for better results:

LinearSVC(multi_class='ovr')

Multi_class: Determines the multi-class strategy if y contains more than two classes. "ovr" trains n_classes one-vs-rest classifiers.

model	Time Taken	Accuracy
S**VM** classification, with count Vectorizer	0.211 min	77.7%
S**VM** classification, with tf-idf Vectorizer	0.023 min	79.0%

Neural Network by Scikit-learn :

Multi-layer perceptron classifier (MLP Classifier) is a feedforward neural network model (connections between nodes do not form a cycle unlike recurrent neural networks) that has been adopted by Scikit-learn. It optimizes the log-loss function using LBFGS or stochastic gradient descent.

Mathematically, neural nets are nonlinear. Each layer represents a non-linear combination of non-linear functions from the previous layer. Each neuron is a multiple-input, multiple-output (MIMO) system that receives signals from the inputs, produces a resultant signal, and transmits that signal to all outputs. Practically, neurons in an ANN are arranged into layers. The first layer that interacts with the environment to receive input is known as the input layer. The final layer that interacts with the output to present the processed data is known as the output layer. Layers between the input and the output layer that do not have any interaction with the environment are known as hidden layers. Increasing the complexity of an ANN, and thus its computational capacity, requires the addition of more hidden layers, and more neurons per layer.

Artificial neural nets are useful for situations where there is an abundance of data, but little underlying theory. Overtraining is a problem that arises when too many training examples are provided, and the system becomes incapable of useful generalization. This can also occur when there are too many neurons in the network and the capacity for computation exceeds the dimensionality of the input space. During training, care must be taken not to provide too many input examples and different numbers of training examples could produce very different results in the quality and robustness of the network.

There are a number of different parameters that must be decided upon when designing a neural network. Among these parameters are the number of layers, the number of neurons per layer, the number of training iterations, et cetera. Some of the more important parameters in terms of training and network capacity are the number of hidden neurons, the learning rate and the momentum parameter.

- Having too many hidden neurons is analogous to a system of equations with more equations than there are free variables: the system is over specified, and is incapable of generalization.
- Having too few hidden neurons, conversely, can prevent the system from properly fitting the input data, and reduces the robustness of the system.
- There are typically 8 neurons in the hidden layer.
- The learning weight is typically 0.3.
- The typical momentum value is 0.9.
- Typical value for epoch: 5000000
- Typical minimum error: 0.01

With the following Default arguments:

```
MLPClassifier(activation='relu', alpha=0.0001, batch_size='auto',
              beta_1=0.9, beta_2=0.999, early_stopping=False, epsilon=1e-08,
              hidden_layer_sizes= (400, 500, 400), learning_rate='constant',
              learning_rate_init=0.001, max_iter=200, momentum=0.9,
              n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
              random_state=None, shuffle=True, solver='adam', tol=0.0001,
              validation_fraction=0.1, verbose=False,
      warm_start=False)
```

model	Time Taken	Accuracy
Neural Network - MLP, with count vectorizer	7.187 mins	78.1%
Neural Network - MLP, with tf-idf Vectorizer	13.785 mins	76.6%

Multinomial Naive Bayes

Naive Bayes assumes the Bayes' theorem, that independence among predictors (a particular feature in one class is unrelated to other features). The algorithm will calculate the likelihoods of each class using Bayes theorem and the class with the highest probability will be the output. Multinomial Naive Bayes algorithm assumes the input variables follows the bell curve (normal distribution) and works very well in classifying multiclass data and text classification in practice. It is easy and fast to predict class of test data set and designed for multi-class prediction. However, the challenge is that the assumption of independence does not hold well in most of the cases in real life. In this project, I use tf-idf to vectorize and highlight the feature importance (based on the count of input variables) before feeding into the models. Though Naive Bayes is not typically regarded as a strong text classifier, it served us well as a foundation for further studies.

With the following Default arguments:

```
MultinomialNB (alpha=1.0, class_prior=None, fit_prior=True)
```

model	Time Taken	Accuracy
Naive Bayes Classification, with count vectorizer	0.001 min	72.8%
Naive Bayes Classification, with tf-idf Vectorizer	0.000 min	67.7%

Stochastic Gradient Descent Classifier

Stochastic gradient descent is an iterative method for optimizing a differentiable objective function (a stochastic approximation of gradient descent optimization). Traditional gradient descent uses least square approach for fitting by calculating the entire data set, which is very expensive and slow to train. Stochastic gradient descent only uses a small sample of training set (selected stochastically) to calculate the parameters, which is faster. In Scikit-learn, stochastic gradient descent is a binary classification. A linear classifier optimized using stochastic gradient descent. scikit-learn’s SGDClassifier allows for many different loss functions.

With the following Default arguments:

```
SGDClassifier (alpha=0.0001, average=False, class_weight=None,
               early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
               l1_ratio=0.15, learning_rate='optimal', loss='hinge',
               max_iter=1000, n_iter_no_change=5, n_jobs=None, penalty='l2',
               power_t=0.5, random_state=None, shuffle=True, tol=0.001,
               validation_fraction=0.1, verbose=0, warm_start=False)
```

model	Time Taken	Accuracy
SGD Classification, with count vectorizer	0.017 min	77.7%
SGD Classification, with tf-idf Vectorizer	0.009 min	78.3%

Gradient Boosting Classifier:

Gradient Boosting Classifier is an ensemble method. It builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage n_classes_ regression trees are fit on the negative gradient of the binomial or multinomial deviance loss function. Binary classification is a special case where only a single regression tree is induced. Imported gradient boosting classifier from sklearn.ensemble and trained data by calling the imported function and fit and score the accuracy in this model. In this we used the below default argument which result a better result.

With the following Default arguments:

```
GradientBoostingClassifier(criterion='friedman_mse', init=None,
               learning_rate=0.1, loss='deviance', max_depth=3,
               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, n_estimators=100,
               n_iter_no_change=None, presort='auto',
               random_state=None, subsample=1.0, tol=0.0001,
               validation_fraction=0.1, verbose=0,
                   warm_start=False)
```

model	Time Taken	Accuracy
Gradient Boosting, with count vectorizer	3.484 min	75.2%
Gradient Boosting, with tf-idf Vectorizer	8.530 min	74.4%

Data Visualization:

Collected accuracy results from all the model and appended it in a list. Using matplot library, created bar graph and plotted the results which is in the list.

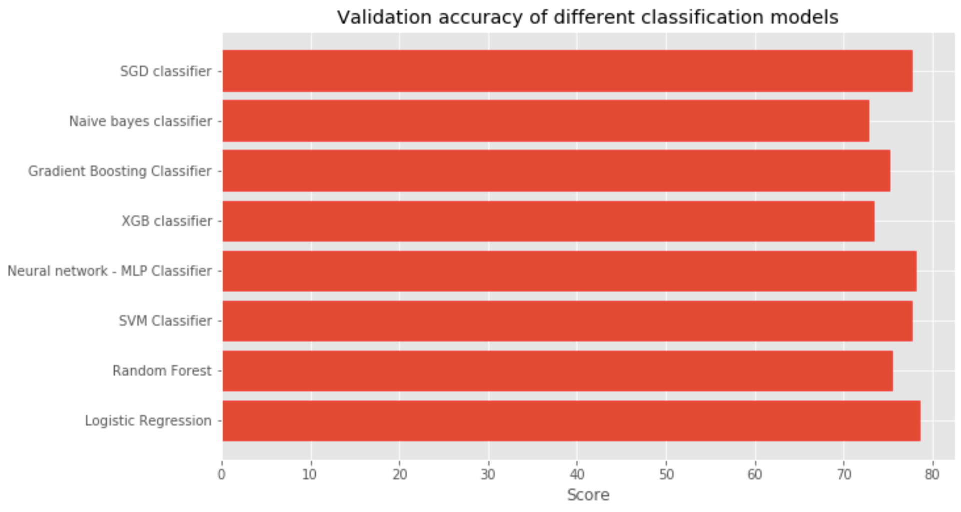
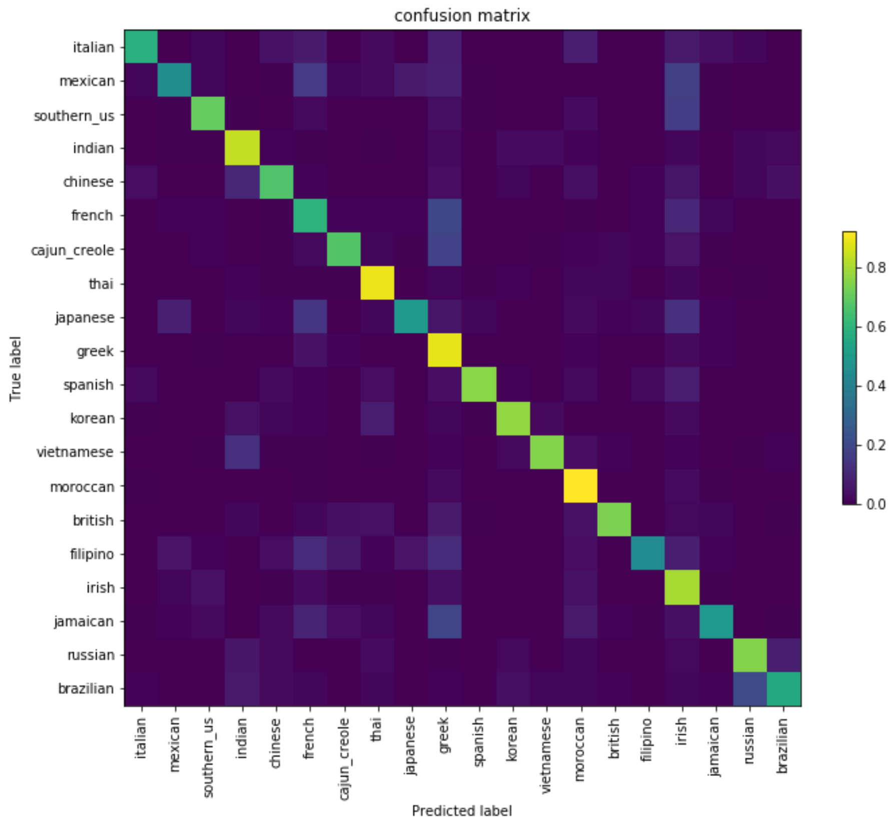


Fig [1]: This Bar graph shows the accuracy of model which used count vectorizer.

Confusion Matrix:

A confusion matrix allows us to see the confusion the classifier makes. It should be read column by column. In each column, one sees the recipes the classifier considered to be one cuisine. Looking at the color in each square one can see the relative accuracy of that classification. Perform confusion matrix on validation set using best model (Logistic Regression).



Confusion matrix using logistic regression.

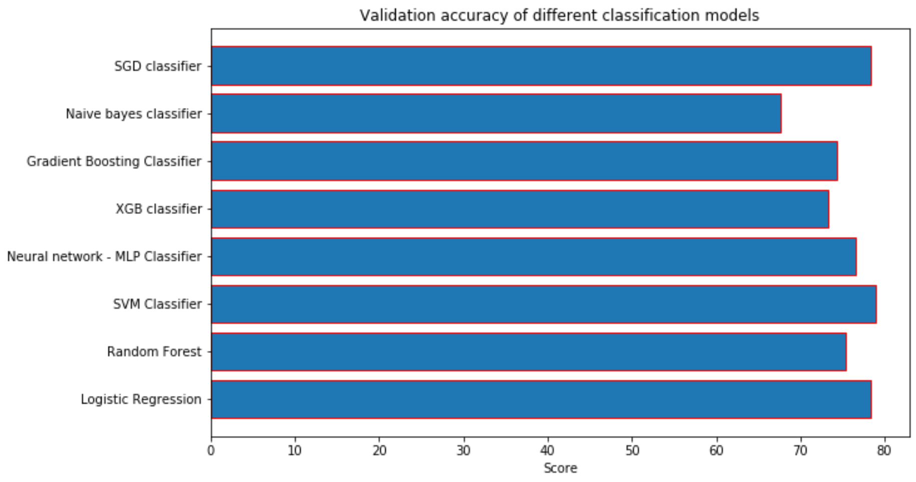


Fig [2]: This Bar graph shows the accuracy of model which used tf-idf vectorizer.

Conclusion:

Finally, we can come to conclusion that using Count vectorization method Logistic Regression gives best accuracy but using tf-idf vectorization method SGD Classification gives best accuracy. Both models are best models in terms of time consumption and accuracy. In Colab, I have performed prediction on test data using these two models.

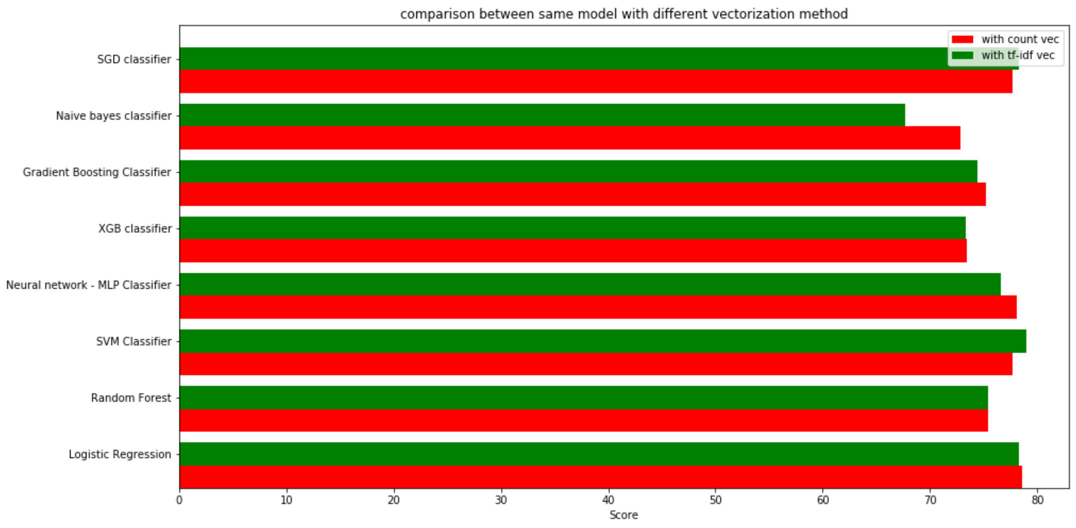


Fig [3]: Comparison between same model accuracy's with different vectorization method

References

1. SciKit-learn models documentation from https://scikit-learn.org.
2. Analytics Vidhya - Kaggle Solution: What's Cooking ? (Text Mining Competition) - https://www.analyticsvidhya.com/blog/2015/12/kaggle-solution-cooking-text-mining-competition/
3. https://flothesof.github.io/kaggle-whats-cooking-machine-learning.html
4. http://courseprojects.souravsengupta.com/cds2015/whats-cooking/
5. http://jeffwen.com/2015/12/19/whats_cooking