Determining the Spiciness of Food Recipes by Using NLP Methods

by
Erwin Mazwardi
jc828371@my.jcu.edu.au

Assessment 3: Web Crawler and NLP System
28/08/2022

1 ABSTRACT

Many recipe sites do not provide information about how spicy their recipes are. Comparing the recipes manually is not possible for users. Although, there are many recipe recommender systems based on recipe titles, ingredients, or contents, they ignore the quantities of the ingredients in the decision making. In this project, the quantities of the ingredients were included in the making of a recipe dataset based on the NLP method. Since the NLTK tagger is not able to recognize word pattern in an ingredient list, a custom tagger was constructed with the help of several dictionaries. The TfIdf vectorizer created features out of the ingredient lists. An unsupervised machine learning (ML) algorithm used the vectorized dataset to train the ML model. Whilst the model accuracy was not really good, the NLP methods used in this project can be improved for similar projects.

2 INTRODUCTION

People who like to cook usually turn to recipe books or online recipes to get an idea of what to cook. However, choosing the right recipes suitable for users' tastes are not easy. The recipes might win popular votes, but it does not mean that those recipes are intended for a particular user. For example, the beef curry recipe has many variations of the same recipe. However, the recipe books or online recipes usually do not provide accurate information about the food taste, whether it is sour, spicy, sweet, or mix of them. As a result, users are forced to rely on popular vote or to try the recipes themselves. Therefore, to ease the user activities, artificial intelligence (AI) software can be constructed using natural language processing (NLP) approach.

To understand why NLP approach can help the creation of AI recipe software, let us consider the following two butter chicken recipes. Users can compare these recipes based on the number or quantity of spices. However, the manual comparison would lead the inaccuracy of

the result, especially more than two recipes are being compared. NLP approach can help the comparison process by extracting ingredients from ingredient lists and transforming the extracted data to format that can be understood by machine learning (ML) algorithms. The ML algorithms will classify the recipes to different tastes, such very spicy, moderate spicy, very sour, moderate sour, and so on.

Using NLP combined with ML or even neural network (NN) to classify food recipes is nothing new. Hanan etc (2015), created a method of classifying closely similar recipes in order to extract spam recipes. The method relies on two-step classifications. In the first, they used K-Means approach to classify the recipes based on food names and cooking methods in the recipe title. In the second step, they utilized Repeated Bisection approach to classify recipes based on ingredients. Both approaches used unsupervised ML algorithms.

On other hand, Campese etc (2021) relied on supervised methods when classifying based food ingredients found in food labels. The intention was to classify foods into the right intolerance group given their ingredients. The way they did it was by extracting correlations and/or semantic relationships among the words used in a food ingredient label. The goal is to predict if a certain food belongs to one of this five main clusters: milk, yeast, salicylates, nickel, and wheat. They used five NLP algorithms to create a benchmark: Convolutional Neural Networks (CNN), Long short-term memory (LSTM), Dense Neural Network (Dense NN), and Linear Support Vector Machine (Linear SVM).

Hanai et. al. (2015) concluded there are four points for clustering similar recipes. First, images are not necessary to find out recipe similarity. Second, titles that include food names, ingredients or cooking methods, are important words. Third, ingredients in an ingredient list play important roles for similarity. Finally, sizzle words in a title are not important to find the similarity.

Both research mentioned previously ignored cardinal numbers related to ingredients. They were only interested with the words. Based on their goals, the results are good enough to recognise the scam recipes and to classify the food labels based on the intolerance levels. However, to determine the spiciness of the food recipe, more information are needed. Therefore, in this project, ingredient quantities will be included in the learning process using ML.

This report is composed of six parts. In Section 2, the problem of classifying recipes based on the spiciness if the ingredients is outlined. The limitation is shown to recognize the difficulty of comparing recipes manually. Few research has been done in this area and these are explored in this section. In Section 3, the methodology to achieve the project is explained in detail. This includes the explanation of the scrapping method, the text pre-processing using NLP methods, the ML model creation and training, the visualization, and the evaluation. In Section 4, implementation details and results of the experimentation are explained in detail. Section 5 presents detailed discussion of experimental results. In Section 6, conclusions are presented. Finally, Section 7 presents the future works to improve the current project result.

3 METHODOLOGY

The first step is to get a clean dataset containing recipes with their ingredient lists. Web scrapping method is used to get the data from online food recipes. There are two ways to get the data: manual coding and scrapping tools. Manual coding using Python will consume a lot of time, which is not recommended for this two-weeks project. The more reasonable approach is by using scrapping tools. There are several food scrapping tools available. The easiest one to be used is a package provided by Python itself namely recipe-scrappers. The goal is to create a table with two columns intended for recipe titles and ingredients.

The second step is to pre-process the ingredient words including removing punctuations, applying stemmer or lemmatizer, and applying stopwords. All punctuations are to be removed except the cardinal numbers. The easiest way to remove punctuations is by utilizing a regular expression package from Python called *re*. Once the unwanted punctuations are removed, the next step is convert words to the root forms. For this case, PorterStemmer or Lemmatizer packages from Python will be used. Unlike *PostStemmer*, *Lemmitizer* process will keep the root words without cutting. Finally, stopwords will be used to remove unwanted words, such as *for*, *or*, *and*, etc.

The third step is to use an NLP approach to tag ingredient words. The problem is the NLTK tagging system does not support tagging the ingredient words. For example, there is no way that the NLTK tagging system will be able to tag the following ingredient list.

1 tablespoon cumin seeds
1 ½ teaspoons ground turmeric

Therefore, a custom tagging will created using the NLTK Unigram Tagger. The tagger needs a dictionary containing words with their own tagging names: CARD (for cardinals), UNIT (for units), and INGR (for ingredients). The dictionary should look like the following.

```
taggedWordDict = {"1": "CARD", "tablespoons": "UNIT", "cumin_seeds": "INGR"
```

After applying the custom tagger function, an ingredient line should look like this. At this time, this the only format allowed for simplification. Other than this will be removed from the ingredient lines.

[("1", "CARD"), ("tablespoon", "UNIT"), ("cumin seeds", "INGR")]
[("1 ½", "CARD"), ("teaspoons", "UNIT"), ("ground turmeric", "INGR")]

The fourth step is to extract features out of the recipe ingredients. The easiest method is by using Bag of Words (BoW). BoW method creates a huge sparse matrix containing counts of all words found in all ingredient lists. In Python, Scikit-learn provides a function namely *CountVectorizer* to perform BoW. Although, BoW performs well, but another method called Term Frequencies-Inverse Document Frequency (TF-IDF) outperforms BoW's performance marginally. Scikit-learn provides a function namely *TfidVectorizer* for this purpose.

Since the goal is to classify the recipes into undetermined clusters, there are several unsupervised ML algorithms including K-means clustering, Hierarchical Clustering, Probabilistic Clustering, Apriori Algorithms, Principal Component Analysis (PCA), and Singular Value Decomposition. Neural Networks are also popular especially for image classifications. In this project, K-Means algorithm will be used, since the dataset contains numerical values only and a K-Means model is fast to be trained.

To use a K-Means model, an optimal number of clusters has to be determined using one of the following methods: Elbow, Average Silhouette, and Gap Statistic. The Elbow method is the simple method to use. Beside determining the number of cluster, K-Means algorithm needs centroids to be initialized. Once, the number of cluster and the initial number of centroids have been determined, the next step is to train a K-Means model.

To observe the prediction performed by the trained K-Means model, PCA method will be used. PCA is a type of dimensionality reduction that can be used to compress datasets. The

PCA is able to reduce the high dimension features to 2D dimensions. The PCA outputs are then plotted using a scatter plot provided by *pyplot* of Python.

The last step is to evaluate the model. Evaluating for unsupervised ML algorithms is not easy and most of the time requires human judgement. However, there are two methods to evaluate unsupervised models. The first one is by using labelled dataset as a reference and combining with the *homogeneity_score* metric supplied by *sklearn.metrics*. The second method is by using unlabelled dataset. For this purpose, the *silhouette_score* metric from *sklearn.metrics* can be used.

4 EXPERIMENTAL RESULT

4.1 Scrapping Online Recipes

Scrapping recipes from online recipe sites was quite straight forward. A function called *scrape_me* from *recipe_scrappers* was used for that purpose. Initial plan was to gather 200 links a minimum of two food websites. However, at the end a total of 57 recipe links was collected from a food website called Foodnetwork.com. This reduction was caused by the hardship of collecting and choosing the links one by one manually. Furthermore, only one website was considered to prevent different standards in writing ingredient lists. Again, the short time given to finish the project was the consideration.

After scrapping the data and store them into a two-columns table containing recipe titles and ingredients, the table then was saved into a file using *json.dumps* function. Unlike saving a table using the *to_csv* function, the *json.dumps* function will maintain the table structure. If the table is visualized using the Pandas dataframe, it will looks like the one shown in Table 1.

Table 1. A snapshot of Pandas dataframe containing title and ingredient columns

	title	ingredient
28	Spicy Stewed Beef	lo_butter lo_cumin lo_chili_powder lo_kosher_s...
26	Spicy Bean Soup	sao_chili_powder lo_coriander lo_cumin
42	Spicy Corn Soup	lo_garlic lo_chili_powder sao_black_pepper lo_...

4.2 Insert (_) for Ingredients with Multiple Words

Since the tagger that will be used in later stages only looks for individual words to be tagged, then to tag multiple words, such as *ground turmeric*, an underscore was used to replace the

blank space between two words. For this purpose a dictionary containing word patterns to be found and replaced with "_" was created and can be found in a file called *generateIngrTags*. The dictionary structure can be seen in following code. In addition, the dictionary facilitates the changing of the original words to different words. For example, the word "kaffir lime leaves" becomes "lime_leaves". The directory then saved into a Jason file.

```
ingrDict = {"ground turmeric": "ground turmeric",
            "kaffir lime leaves": "lime_leaves",
            ...}
```

Back to the main Jupiter file, the process of finding word patterns to be replaced was done with the help of the Python regular expression function, *re.sub*, from the *re* library. The first, second, and third parameters of this function is the pattern to be found, the replacement word(s), and the sentence containing the pattern correspondingly.

```
lineList.append(re.sub(ingrKeys[x], # Pattern to be found
                       ingrRepDict[ingrKeys[x]], # Replacement words
                       rawRecipeLists[i][1][y])) #
```

4.3 Create a Custom Tagger

To support the custom tagger, three dictionaries were created. The first dictionary was called as *ingrTagDict*. It had ingredient words as keywords and *INGR* as the tagging values. The second dictionary was called as *unitTagDict*. It listed all units that are needed to be tagged. The name of the tag was *UNIT*. The third dictionary was called as *cardTagDict*. It tagged all cardinals with *CARD*. The snapshots of these three dictionaries can be seen in the following table. The content of all dictionaries were created manually. Due to the time limitation, many ingredients were not included into the dictionary. This would lead to the inaccuracy of ML models. These three dictionaries were coded in a file called *generateIngrTags.ipynb*. and a snapshot of them can be seen in Table 2.

Table 2. A snapshot of the structure of the three dictionaries

ingrTagDict = { "fish_sauce": 'INGR', "garam_masala": 'INGR', ...}	**unitTagDict** = { "can": 'UNIT', "cups": 'UNIT ...}	**cardTagDict** = { 'one': 'CARD', 'two': 'CARD', ...}

In the main file, these three dictionaries were combined into a dictionary called *tagDict*, as it can be seen in Sec 3 of the main code. The combined dictionary became an input parameter to a function called *nltk.UnigramTagger()* that would create a custom tagger. The part of the code can be seen below.

```
customTagger = nltk.UnigramTagger(model=tagDict)
```

Once the custom tagger ready, the text pre-processing step was started. The code for this step can be found in Sec 4 of the main file. What the code did was to apply the custom tagger, remove untagged words, and convert cardinals and units to levels. For the third activity, another dictionary was created. The dictionary was intended to convert cardinals and units to levels. Due to the time limitation, the level types were determined not based on proper research, but was based on the author's judgement. Those level types and their short versions were as follows:

```
tiny amount of = tao_
small amount of = sao_
medium amount of = mao_
lot_of_ = lo_
```

The level types were then paired with corresponding cardinals and units. Again, the pairing was not based on proper research, but was based on the author's judgement. This might lead to the inaccuracy of the ML model be built in the later step. The content of the dictionary would look like this.

```
unitConvDict = {
 "0.125 teaspoon": "tao_",
 "1.50 tablespoon" : "lo_",
 "3.50 cups": "lo_",
 ...}
```

The original recipe dataset had three levels of list. The first list (**[ ]**) consisted of all recipes, the second list (**[ ]**) was for one recipe, and the third list (**[ ]**) contained the recipe title and the ingredients. The print out of the list is shown in Figure 1.

```
[['Hyderabadi Biryani',
   ['2 pounds boned leg of lamb, cut into 0.75-inch cubes',
    '1 tablespoons papaya_paste (tenderizing agent)' ...]]
 ['Yogurt Marinated Lamb with Tri-Masala',
   ['1 pound lamb, cut into 1-inch cubes',
    '1 pint plain yogurt', ...]]
 ["Mrs. Ayyer's Vegetable Biryani with Tempered Onion-Cucumber Raita",
   ['1 large cucumber',
    '0.25 cup plus 1 tablespoon ghee', ...]]
  ...]
```

Figure 1. A snapshot of the pre-processed recipe texts

Based on Figure 1, to get the ingredient list, three for-loops were required. The first loop is to go through each recipe. While in the first loop, the second loop is executed to get into the title and ingredients of each recipe.

In this project, *parsedRawRecipeLists[i][1][y]* was tagged by the *customTagger.tag()* function, where *i* was an index for looping the recipe list, *1* was an index to choose the ingredient list, and *y* was the index to the words in an ingredient line. The tagging code can be seen below.

```
tagIngr = customTagger.tag(parsedRawRecipeLists[i][1][y].split(' '))
```

A snapshot of the custom tagger output can be seen in Table 3. In this project the ingredient words that did not have the *INGR* tag would be removed after the tagging process. This was to speed up the finishing of the project.

Table 3. A snapshot of ingredient lines before and after tagging

Original Ingredient Lines	After Tagging
2 pounds boned leg of lamb, cut into 0.75-inch cubes	[('2', 'CARD'), ('pounds', 'UNIT')]
1 tablespoons papaya_paste (tenderizing agent)	[('1', 'CARD'), ('tablespoons', 'UNIT'), ('papaya_paste', 'INGR')]
2 tablespoons ginger	[('2', 'CARD'), ('tablespoons', 'UNIT'), ('ginger', 'INGR')]

While in the recipe loop block, convert the cardinals and units to levels. As can be seen from the code below, *recipeLevel[0][1]* represents the cardinal column, *recipeLevel[1][0]* represents the unit column, and *recipeLevel[2][0]* represents the ingredient column. The first

line of the code combines the first and second columns. The second line will use the combined columns as the key to get the corresponding level.

```
combCardUnit = recipeLevel[0][0] + ' ' + recipeLevel[1][0]
convUnitIngr = unitConvDict.get(combCardUnit) + recipeLevel[2][0]
```

The recipe list with the new ingredient values was created and saved into a Jason file for further processing. After getting replaced by levels, the dataset looked like the following. The first column contained recipe titles formatted in string, while the second column contained ingredient lists.

```
[['Hyderabadi Biryani', ['lo_papaya_paste', 'lo_ginger', ...]],
 ['Yogurt Marinated Lamb', ['lo_butter', 'sao_mint', ...]], ...]
```

4.4 Final Text Pre-Processing

Final text pre-processing as shown in Sec 5 is to convert the recipe list into a Pandas dataframe, so that it can be processed easily in the later steps. The dataset resulted from previous step had its ingredient list surrounded by an outer list. This outer list had to be removed and leaved the ingredient list in a string format. After this, recipes that have no ingredient lists were removed from the dataset. This had to be done to prevent any errors raised in further processing steps. As a result of the removal, the dataframe length was reduced from 57 to 50. Although, it was undesired, the final dataset was still above the minimum requirement.

4.5 Exploratory Text Data Analysis

To find out the inside of the dataset, the dataset was set by a function called *freq_words()*. This function was originally created the JCU team. The function uses the nltk.FreqDist function to calculate the distribution of words in the ingredient lists. The inputs to this function was the *recipeDF[1]* and 100 as the most frequent words.

The result demonstrated that the dataset was too unbalanced. There were ingredient lists that had maximum amount of ingredient such as *mmo_cumin*. On the other hand, there were ones that had too little ingredients, such as *mmo_black_pepper*. The distribution plot can be seen in Sec 6 of the main file.

4.6 Find Optimal Cluster (K) using the Elbow Method

Before a K-Means model can be trained, the cluster number has to be identified first. Sec 8 shows parts of the code for achieving this. Since the total row numbers of the recipe dataset was 50, then the maximum clusters used in the Elbow method was 50 as well. The result can be seen in Figure 2 that clearly shows that the process failed to find the expected number of clusters. It might caused by the inaccuracy of the dataset.

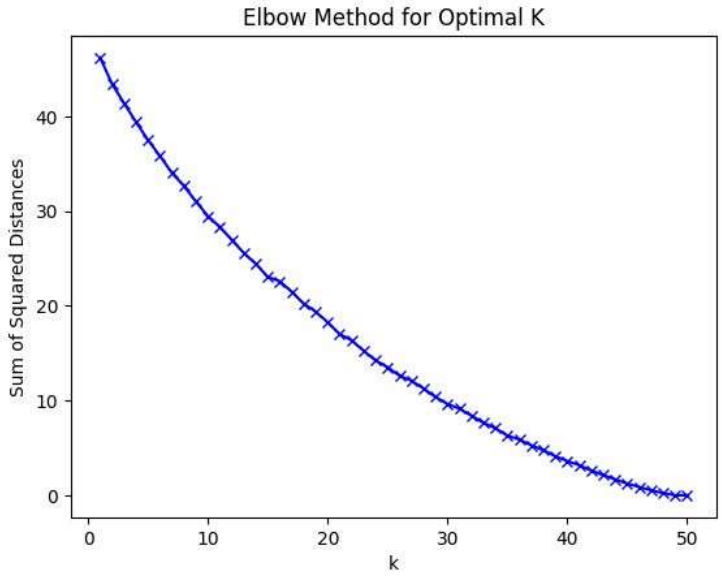

Figure 2. The Elbow plot

4.7 Train a K-Means Model

Since the Elbow activity failed to find the optimal cluster number, a number of 6 was chosen arbitrarily. The code is shown below. The first line was to train a K-Means model and the second line was to train the model with the vectorized dataset, *tfidRecipe*. The parameter *n_init* is used to define the number of initialization attempts for centroids. As a good of practice, default value of 10 generates good results without taking too much computational power. The parameter *random_state* was set to the same value all the time, so that the model would produce the same clusters over and over again. For the parameter *init*, *k-means++* was chosen, since the model had to be trained with a large amount of features. However, it was found out that, there were not different either *k-means++* be used or not.

```
kmeans = KMeans(n_clusters=6, n_init=1, random_state=1234)
kmeans.fit(tfidfRecipe)
```

The result from the trained model then was used to classify the recipes. Part of the Pandas table can be seen in Table 4.

Table 4. A snapshot updated dataframe with cluster column

	title	ingredient	cluster
25	Lentil Soup	mao_onion lo_kosher_salt sao_coriander sao_cumin	0
15	The Best Butter Chicken	mao_yogurt lo_kosher_salt lo_ginger lo_butter ...	0
14	Instant Pot Butter Chicken	mao_ginger mao_garam_masala mao_paprika sao_cu...	0
41	Chilled Corn Soup	lo_brown_sugar	0
37	Shrimp Ramen	mao_soy_sauce	0
33	Thai Chicken Soup	lo_curry_paste	0
3	Biryani Burger with Mango Salsa and Yogurt Sau...	sao_yogurt lo_kosher_salt lo_yogurt lo_mayonna...	0
2	Mrs. Ayyer's Vegetable Biryani with Tempered O...	sao_mustard mao_yogurt lo_lime_juice mao_cumin...	0
32	Thai Curry Chicken Noodle Soup	sao_shallots lo_curry_paste mao_curry_powder s...	1
39	Quick Beef Pho	lo_fish_sauce lo_sugar	1
38	Vietnamese Chicken Noodle Soup	sao_garlic lo_fish_sauce	1

4. 8 Visualization

There is no way to visualize datasets with large features using a 2D plotter, unless the size is reduced to a 2D space. PCA can help with this transformation. The code was written as shown below. The first line is intended to create a PCA model. In the second and third lines, the model are trained with the vectorized dataset and the cluster centres calculated by the K-Means model.

```
pca = PCA(n_components=2, random_state=1234)
reducedFeatures = pca.fit_transform(tfidfRecipe.toarray())
reduceClusterCentres = pca.transform(kmeans.cluster_centers_)
```

Once the features had been reduced to the 2D form, those points were plotted using a scatter plot. For the first plot, the first and second parameters were the X and Y values of the *reducedFeatures*, while the third parameter was the classification values resulted from the prediction values of *tfidRecipe*. The second plot was resulted from the *reduceClusterCentres* dimension. It plotted the locations of the centroids.

```
plt.scatter(reducedFeatures[:,0], # 1st parameter
            reducedFeatures[:,1], # 2nd parameter
            c=kmeans.predict(tfidfRecipe)) # 3rd parameter
plt.scatter(reduceClusterCentres[:,0], # 1st parameter
            reduceClusterCentres[:,1], # 2nd parameter
            marker='x', s=150, c='b') # 3rd parameter
```

From the scattered plot shown in Figure 3, it can be seen that there are three centroids located next to each other around the centre area. Therefore, a majority of cluster overlapping happened around the centre area. The scatter plot proved that the arbitrary chosen cluster number gave inaccurate classifications.

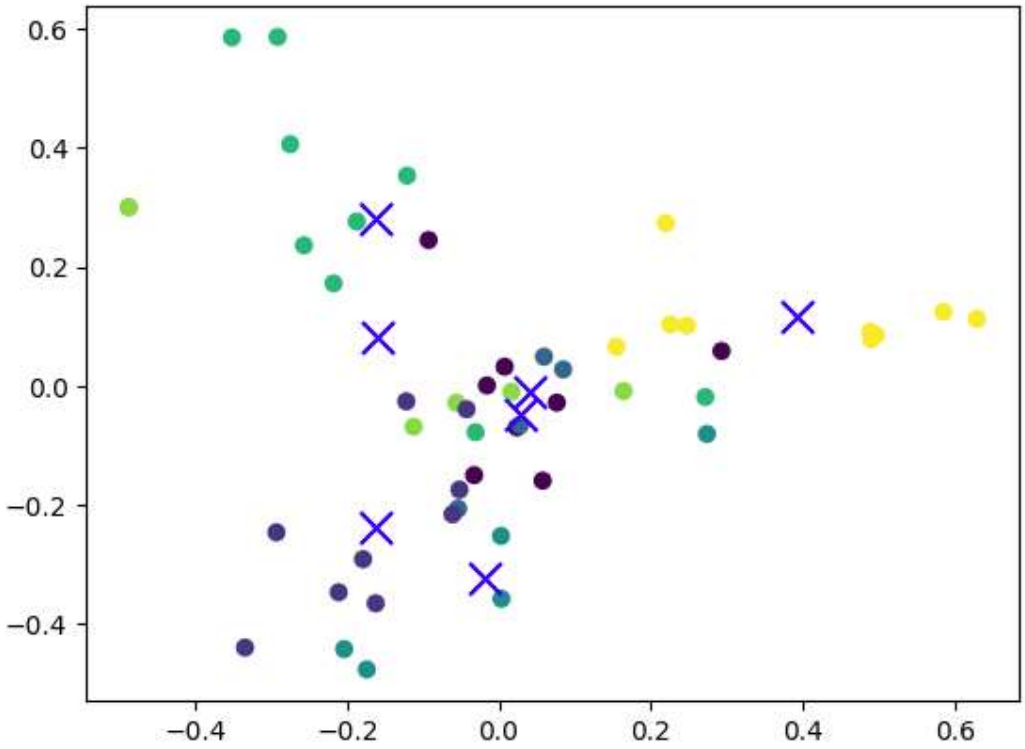

Figure 3. The scatter plot of PCA features

4.9 Evaluation

The following is the code written to evaluate the K-Means model. The result was 0.059, which indicated that there were too many overlapping. The best value is 1, and 0 indicated overlapping.

```
from sklearn.metrics import silhouette_score
silhouette_score(tfidfRecipe, labels=kmeans.predict(tfidfRecipe))
```

If the result of 0.059 is put at a line with a range from -1 to 1 as shown in Figure 4, it can be clearly seen that it was not a satisfactory result.

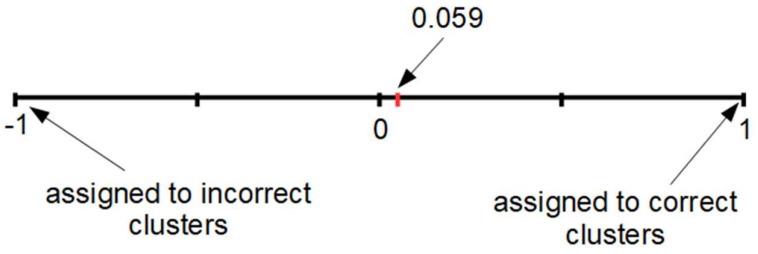

Figure 4. The position of the current result

Although the silhouette score can show the model accuracy, it does not show what the cause of the problem. Therefore, to investigate more detail one cluster was chosen. In this case, the chosen cluster in 1, because it had one of the largest number of recipes within a cluster. The result can be seen in Figure 5.

```
# Show cluster six only
for i in range(0, len(clusterSix)):
    print('Index:' + ' ' + str(clusterSix.index[i]))
    print('Title:' + ' ' + clusterSix.iloc[i][0])
    print('Ingr :' + ' ' + clusterSix.iloc[i][1])
    print('===')
```

```
Index: 28
Title: Spicy Stewed Beef
Ingr : lo_butter lo_cumin lo_chili_powder lo_kosher_salt lo_mustard lo_black_pepper
===
Index: 26
Title: Spicy Bean Soup
Ingr : sao_chili_powder lo_coriander lo_cumin
===
Index: 42
Title: Spicy Corn Soup
Ingr : lo_garlic lo_chili_powder sao_black_pepper lo_chicken_stock mao_heavy_cream
===
Index: 8
Title: Southern Indian Lamb Curry
Ingr : lo_cumin lo_coriander lo_turmeric lo_ginger sao_yogurt
===
Index: 1
Title: Yogurt Marinated Lamb with Tri-Masala and Basmati Rice (Hyderabadi Biryani)
Ingr : lo_butter sao_mint mao_cumin lo_coriander lo_fennel_seeds lo_poppy_seeds sao_yogurt
===
Index: 46
Title: Shrimp and Corn Chowder
Ingr : lo_butter
===
Index: 47
Title: New England Clam Chowder
Ingr : lo_butter mao_heavy_cream
===
Index: 29
Title: Laila's Stewed Beef with Creamy Cheese Grits
Ingr : lo_chili_powder lo_cumin lo_butter
===
Index: 49
Title: Corn Chowder
Ingr : lo_butter lo_heavy_cream
===
```

Figure 5. Recipes classified into cluster six

5. DISCUSSION

In terms of the dataset, a dataset containing 50 cleaned recipes were too small for any ML algorithms. As a result, the K-Means model could not be trained accurately. Campese et. al. (2021) for example used a total of 8500 recipes for classifying food labels. Kalra et. al. (2020) used a set of 2,482 recipes for calculating calorie information. Bien et. al. (2020) used a subset of 500 recipes for generating semi-structured text. Based on the research papers, at least 500 recipes should be collected, if an accurate model is wanted to be built.

Although, the *scrape_me* function did a really well job in extracting recipes from the internet, it does not have features to select food categories and to specify multiple sites. Imagine

selecting 500 recipes from 10 websites. It would be really a challenging task. Therefore, this kind of features should have been done by developers by creating a wrapper around the *scrape_me* function.

Another factor that needs to be considered is the creation of the custom tagger. There are hundred of food ingredients, but only 45 was recorded in the ingredient tag dictionary, *ingrTagDict*. As a result, many ingredients were omitted during the tagging process. The same case happened to the ingredient directory, where there were only 64 ingredients recorded to be converted to a single word or multiple words with underscore.

Unlike other research on NLP based food classifications, in this project cardinals were included into feature creations. Since there were many cardinals, several level classes were required to classify the cardinals. Unfortunately, the cardinal classification was done manually based on author's subjectivity. This lead to missing or inaccurate representations of level classes. For example, a large quantity of curry spices is not the same as a large quantity of onion chops. Table 5 shows examples of ingredient lines ignored in this project.

Table 5. Some examples of ignored ingredient lines

Ingredient Line	**Reasons for exclusion**
2 bay leaves	No unit
One 3-inch cinnamon_stick	Does not match (CARD)(UNIT) format
Kosher salt	No cardinal and unit
Chopped cilantro leaves	No cardinal and unit

As stated previously, the Elbow method failed to find the optimal number of clusters, even after it reached the sum of squired distance of 0 or corresponding to the cluster number of 50. It means that the algorithm failed to separate clusters. One of the answer for this problem is to do proper data pre-processing. The small size of the dataset and the incompleteness of the ingredient dictionaries might lead to the failing of the Elbow method. In addition, the K-Means algorithm may require further tuning, although it's not clear how it's going to be done. The last resort is to use other ML algorithms or even Neural Networks algorithms.

The K-Means algorithm still produced clusters with limitations mentioned previously. The high dimensionality features were converted to the 2D dimensions by using the PCA method. It worked well, since all clusters are can be viewed using a 2D scatter plot. Based on the plot,

it can be observed that there are several clusters existed. However, the plot did not give the legend information causing the difficulty to identify recipes that belon to certain clusters.

To evaluate the model, the *silhouette_score* metric was used. However, the result was not great, since it pointed out that there were many overlapping in the classification. The manual inspection can show where the problems exist. Figure 1 shows there are too many overlapping within a cluster. Let's investigate cluster 1 consisting recipe indexes of 28, 26, 42, 8, 1, 46, 47, 29, and 49. Table 6 shows some of ingredients contained in the recipes. Here, six of nine recipes contained *lo_butter*. Recipe 46, 47, and 49 did not contain any spicy ingredients, but they were still put into the same category of recipe 1 consisting the most species. The main reason why they were put in the same category is because they contained *lo_butter*. Take another example, *mao_heavy_cream*. It put recipe 42 and 47 into the same category, although recipe 42 contained more species and recipe 47 did not species at all. Based on these findings, two-step classifications may a good idea in order to separate the non-spicy recipes with the spicy recipes.

Table 6. Ingredients contained in recipes of cluster six

	28	26	42	8	1	46	47	29	49
lo_butter									
lo_chili_powder									
lo_cumin									
lo_kosher_salt									
lo_mustard									
lo_black_pepper									
lo_coriander									
lo_garlic									
lo_turmeric									
lo_ginger									
lo_heavy_cream									
mao_heavy_cream									

6. CONCLUSSION

This project proposed a classification method to determine the spiciness of a recipe based on its ingredient list. The purpose method is to utilize the ingredient quantities to judge the similarity between recipes.

Web scrapping was done using a recipe scrapper tool provided by a Python library. The tool only incorporates basic features. As result, it was a challenging task when it came of choosing online food recipes.

Two NLP methods was used here. The first one was the use of *nltk.UnigramTagger* for creating a custom tagger and the second one was the use of *TfidfVectorizer* for vectorize the ingredient words. In terms of training and test datasets, since the recipes were classified into unknown clusters, the test dataset was not required.

The custom tagger requires a dictionary, which paired tagging words with different ingredients, cardinals, and units. However, the dictionary was far from complete. Since, the tagger was defined manually, stemmer or lemmatization was not required. However, multiple words of an ingredient that had to be recognized by inserting underscores between them. This required the creation of another dictionary. This project also introduced level classes corresponding with the cardinals and units.

Based on the evaluation step, it seems that the K-Means algorithm failed to classify recipes based on the ingredients. There were too many overlapping recipes in one cluster. There are several problems been identified: the small size of the dataset, the small size of ingredient words recorded in the custom tagger, and the use of one-step classification. As a result, the model accuracy was quite low as pointed out by the *silhouette_score* metric.

The use of PCA in visualization of the predicted result was quite useful. Although, it was hard to understand, since there were no legends. However, PCA caused the higher dimension features to be plotted on a 2D area using a 2D scatter plot provided by *pyplot* of the Python library.

Although, the model accuracy was low, the proposed method is promising enough once the pre-processing part of the recipe dataset is improved. It must be accompanied by creating a modular function that is able to scrape the food websites in large amount of data.

7. FUTURE WORKS

Based on the finding during the experimentation phase, there are several things has to be done in order to get an accurate recipe classifier.

1. Enhance the scrapping process by creating a wrapper for scrape_me. The wrapper should have an option for listing different food websites and also has an ability to specify several food categories to be downloaded.
2. Enhance the tagging process by adding more ingredient tags and ingredients into the corresponding dictionaries. There should an automation approach to select the ingredient words from ingredient lists.
3. Fix the unit conversion dictionary to include more accurate class levels. Further research needs to be done to see whether this activity has been carried out by other researchers.
4. Add two-step classifications, where the first step is to separate recipes based their main taste (sour, sweet, hot, spicy, blunt, and so on) and the second step is to rank how much spiciness or sweetness in the same category.
5. Try to use other unsupervised ML or NN algorithms. Although, NNs are a little bit challenging to understand and require a computing machine with a graphic card, it is worthwhile to try in order to increase the model performance.

REFERENCES

Hanai, S., Nanba, H., and Nadamoto, A. 2015. **Clustering for Closely Similar Recipes to Extract Spam Recipes in User-generated Recipe Sites**. December 2015. ACM 978-1-4503-3491-4/15/12.

Campese, S. and Pozza D. 2021. **Food Classification for Inflammation Recognition Through Ingredient Label Analysis: A Real NLP Case Study**. DOI: 10.1007/978-3-030-55187-2_15.