Naive Bayes Classifier for Natural Language Process

Abstract

In 2012, hurricane Sandy ruined the whole American and killed 233 people along the path of the storm in eight countries. During the hurricane season, there is a topic #HurricaneSandy where people shared the information about Sandy and posted needs and help. To spread the message and help these tweet authors timely and effectively, the tweets could be classified into five categories—*Energy, Food, Medical, Water* and *None*. In this paper, a Naive Bayes Classifier was proposed to do the classification. We will make a basic Naive Bayes Classification and try to make some improvement on the accuracy from different aspects. We will also give a recommendation of the best-performed modified classification and show the source code.

Introduction

Naive Bayes Classification is regarded as one of the simplest and most powerful algorithms. In spite of the great advances of the Machine Learning in the recent years, it has proven to not only be simple but also fast, accurate and reliable. Here we applied it in a natural language process problem, which requires to classify tweets into five categories—*Energy*, *Food*, *Energy*, *Water* and *None*.

We began with the introduction of how Naive Bayes Classification works on natural process and then present the implementation. In the implementation part, we built up a basic Naive Bayes Classifier by Python and give a prediction to the testing dataset. After that, we attempted to improve the classifier through three different aspects—algorithm, language model and strategy.

All the datasets are from https://github.com/abisee/sailors2017. We used the training data sheet to train our classifier and the test data sheet to calculate the accuracy.

Naive Bayes Classification

The classification is based on the Bayes theorem:

$$P(C|A) = \frac{P(C)P(A|C)}{P(A)} - - - Eq.(1)$$

Here we apply the theorem into natural language process scenario. Event C will be the category of the tweet while event A is the target tweet. We tokenize the tweet into different components as the sub events that cause the event A. The revised formula is as below:

$$P(c|a_1, a_2, \dots, a_n) = \frac{P(c)P(a_1, a_2, \dots, a_n|c)}{P(a_1, a_2, \dots, a_n)} - - - Eq.(2)$$

Since we are concerned about if the probability of this tweet labeled as this category is relatively larger than the other categories, we only focus on the numerator. Then we assume that every word in a sentence is independent so to make the classifier *Naive*.

$$P(c)P(a_1, a_2, \dots, a_n|c) = P(c)\prod_{i=1}^{n} P(a_i|c) = P(c)\frac{\prod_{i=1}^{n} D(a_i, c)}{D(c)} - - - Eq.(3)$$

Where *D* represents counting the number of the events.

For example, a tweet "I like cheese" will be tokenized to "I", "like", "cheese". To calculate the probability of it is in *Food* category, we simplify the formula to be:

$$P(\text{"I like cheese"}|Food) = P(Food)\frac{D(\text{"}I\text{"}, Food)D(\text{"}like\text{"}, Food)D(\text{"}cheese\text{"}, Food)}{D(Food)}$$

After calculating all the probability under different category, we find the max one and classify this tweet as relevant category.

First Training and Testing

For the very beginning, we just build up a unigram language model to process the training data. Unigram model means using word frequency in this category to represent $P(a_i|c)$ without considering sentence condition or words order. In the following part, we will try to make improvement from this point. In addition, we use the TweetTokenizer from the module NLKT to do the tokenization. Based on this simple model, the accuracy of the prediction on testing data is only 40.21% The following is the first five lines of our prediction on the test data. See Appendix A for

the source code.

	Tweets	category	prediction
0	I have a lot of canned goods and some clothing , but I can also buy and bring things as needed . Please let me know what you need most .	Food	Food
1	How the **** am I supposed to get @ MeekMill new album when I ai n't got power ? **** outaaa here sandy !	Energy	Medical
2	Frankenstorm wo n't stop the bean ! Thx for staying open for the neighbors who need coffee and treat ! (@ The Bean) http : //t.co/Zw7oA0Tq	Food	Medical
3	Deodorant Toothpaste Shampoo/Conditioner Baby Shampoo Kids Toothbrush Bar Soap Mouth Wash Q-tips painkillers	Medical	Energy
4	clothes for baby kids woman men , food , non perishable food , tools , toys , paper , furniture , any products ,	Food	Food

Table 1: First Prediction of Testing Data

Here is a probability list of the first five tweets in the testing dataset. We shuffle the list before the sorting to remove the data biases.

```
[('Food', 1.192e-70), ('Medical', 0), ('Energy', 0), ('Water', 0), ('None', 0)]
[('Medical', 0), ('Food', 0), ('None', 0), ('Water', 0), ('Energy', 0)]
[('Medical', 0), ('Energy', 0), ('Food', 0), ('None', 0), ('Water', 0)]
[('Energy', 0), ('Food', 0), ('Medical', 0), ('Water', 0), ('None', 0)]
[('Food', 9.775e-53), ('Water', 0), ('Medical', 0), ('None', 0), ('Energy', 0)]
```

List 1: Probability Rank of the First Five Tweets with No Correction

Laplace Smoothing

As what we see in List 1, there are many probabilities 0. The reason is, in the calculation of probability, if there exists a word that has never appeared in the training data, $D(a_i, c)$ will be 0 and the whole multiplication will get 0 as the result. The classifier will not choose this category only because there exits a word that not in the training data. However, the loss of this word can be caused by the deficiency of the data collecting rather than it does not belong to this category. To solve this problem, we apply Laplace Smoothing, which makes some correction in the calculation of $P(a_i|c)$ and $P(c)$. See Appendix B for the source code.

$$\hat{P}_k(c) = \frac{|D(c)| + k}{|D| + k \cdot N} - - - Eq.(4)$$

$$\hat{P}_k(a_i|c) = \frac{|D(a_i, c)| + k}{|D(c)| + k \cdot N'} - - - Eq.(5)$$

Where k represents the correction parameter, N and N' represents the number of distinct category and word respectively, e.g. $\hat{P}_1(Food) = \frac{|D(Food)|+1}{|D|+1\cdot 5}$, here are five categories hence $N = 5$. To enhance the performance of classifier, we can use Grid Search method to find out the most suitable correction, however, we did not implement it and set the default correction parameter to be 1, because this method is a common way in Machine Learning to find suitable parameters, which is beyond our topic. The corrected predictions for the first five tweets are below, and the accuracy increases to 74.37%. See Appendix B for the source code.

[('Food', 3.462e-73), ('Energy', 3.113e-84), ('Water', 2.546e-87), ('None', 7.276e-91), ('Medical', 1.69e-92)]
[('Energy', 1.403e-76), ('Food', 1.084e-78), ('None', 9.984e-82), ('Water', 1.478e-84), ('Medical', 3.818e-89)]
[('None', 3.305e-87), ('Energy', 4.041e-90), ('Food', 1.144e-93), ('Water', 4.196e-96), ('Medical', 5.094e-99)]
[('Medical', 6.471e-53), ('Water', 2.443e-54), ('Energy', 4.653e-55), ('Food', 5.47e-56), ('None', 2.381e-58)]
[('Food', 1.428e-54), ('Water', 8.356e-65), ('Energy', 3.884e-67), ('Medical', 4.134e-70), ('None', 2.334e-73)]

List 2: Probability Rank of the First Five Tweets with Laplace Correction

Bigram Language Model

As what we mentioned above, the basic one we used unigram language model to tokenize sentences and calculate probability without considering the condition of the sentence and the order of the words. Hence, we replace unigram model by bigram model. Although there can be N-gram model to find the most suitable model for the classification, for the limited space, we only talk about bigram here. As what mentioned in Laplace Smoothing part, Grid Search method is still available for this situation.

As for calculating the probability in bigram model, there are two different methods. First, we simply tokenize the sentence into the combination of two words and replace the word vector in unigram by this combination. For example, "I like cheese" was tokenized as ["I", "like", "cheese"] in unigram model, while in this method of bigram model, it will be ["I like", "like cheese"]. The previous formula will be:

$$P(\text{"I like cheese"}|Food) = P(Food)\frac{D(\text{"}I\ like\text{"}, Food)D(\text{"}like\ cheese\text{"}, Food)}{D(Food)}$$

Also, we apply Laplace Smoothing into the calculation, however the accuracy reduces to 68.33%.

The second method is to calculate the occurrence in a consecutive way. Given the first word and category, we calculate the probability of the second word to be next word, and iteratively generate the probability of the whole sentence classified into this category.

$$P(a_n|a_{n-1}) = \frac{D(a_n, a_{n-1})}{D(a_{n-1})} - - - Eq.(6)$$

$$P(a_1, \dots, a_n|c) = P(a_1|c)\prod_{i=2}^{n} P(a_i|a_{i-1}, c) = \frac{D(a_1, c)}{D(c)}\prod_{i=2}^{n} \frac{D(a_i, a_{i-1}, c)}{D(a_{i-1}, c)} - - - Eq.(7)$$

However, after several attempts, the accuracy in this method is still about 68%. See Appendix C for the source code.

There are many causes that make bigram model perform worse than the unigram model. Here we concluded two major reasons. First, the dataset is special. The contents of the dataset are tweets instead of other formal texts. People type tweets really casually and use lots of emoji which even can not be encoded by the program language, like “outaaa” and “”. There are many messy punctuations and incoherent words, hence, the bilingual model did not fit the dataset. Second, the scenario is special. Here we want to classify the tweets into five categories to assist the author after Hurricane Sandy’s destruction. Therefore, a single key word is more important than the coherence of the words. For example, a key word “bread” is sufficient to label this tweet as *Food*.

Strategy

There are two strategies to make a classification system—one-versus-all (OvA) and one-versus-one (OvO). Up to now, we have established a multiclass Naive Bayes Classifier, and use OvA strategy to make the classification. In other words, we classify the tweets into five classes is to train 5 binary classifiers, one for each category (a *Food*-detector, an *Energy*-detector, a *Water*-detector, and so on). Then when we classify a tweet, we get the decision probability from each classifier for that

tweet and you select the category whose classifier outputs the highest probability. Another strategy is to train a binary classifier for every pair of categories: one to distinguish *Energy* and *Food*, another to distinguish *Energy* and *Water*, and so on. If there are N classes, we need to train $N \times (N-1)/2$ classifiers. For the tweet classification, this means training 10 binary classifiers, while each classifier only needs to be trained on the part of the training set for the two categories that it must distinguish.

In this part, we try to use OvO strategy to make an improvement. First, we build a binary classifier only to classify two given categories. If we want to classify a tweet into *Energy* or *Food*, we only extract the dataset labeled as *Energy* and *Food* from the whole training data. Then, the binary classify has to choose one category to label this tweet with Naive Bayes Classification and Laplace Smoothing. Second, after sending this tweet to total 10 classifiers, they make a classification of every pair of categories. We take down these results into one table like this:

	Energy	Food	Medical	Water	None
Energy	0.0	1.0	-1.0	-1.0	-1.0
Food	-1.0	0.0	-1.0	-1.0	-1.0
Medical	1.0	1.0	0.0	1.0	-1.0
Water	1.0	1.0	-1.0	0.0	-1.0
None	1.0	1.0	1.0	1.0	0.0

Table 2: OvO Classification Result of First Testing Tweet

Here we simply stipulate that winner scores 1 point and loser lost 1 point, which named as Victory Count Method. Using G to represent Table 2 and i, j $(i \neq j)$ to represent different categories, $G_{i,j} = 1$ means category j was chosen after the classification between category i and j, $G_{i,j} = -1$ means category j was not chosen after the classification between category i and j. We add up the entries for each column and generate the final score of each category, e.g., the score of *Energy* in this table is 2. Finally, we predict the category by ranking the scores.

```
[('Food', 4.0), ('Energy', 2.0), ('Water', 0.0), ('Medical', -2.0), ('None', -4.0)]
[('Energy', 4.0), ('None', 2.0), ('Food', 0.0), ('Water', -2.0), ('Medical', -4.0)]
[('None', 4.0), ('Food', 2.0), ('Energy', 0.0), ('Water', -2.0), ('Medical', -4.0)]
[('Medical', 4.0), ('Water', 2.0), ('Food', 0.0), ('Energy', -2.0), ('None', -4.0)]
[('Food', 4.0), ('Water', 2.0), ('Medical', 0.0), ('Energy', -2.0), ('None', -4.0)]
```

List 3: Scores Rank of the First Five Tweets under OvO

Although the OvO strategy took 185 times longer than the running time of OvA strategy, the performance (accuracy) increased to 89.68%! If the running time is not concerned, OvO strategy can be considered. See Appendix D for the source code.

In addition to using the table like Table 2 to calculate score, we can also use Massey Method to rank the categories after every binary classification finished. We convert every binary classification into a round of competition, and convert the probability difference into net point difference. For example, the data of Table 2 will be converted into the following round robin tournament graph as Figure 1, where v_i represents the probability difference in *i*-th classification round. Therefore, the problem is converted to calculating the rank vector *r* as shown in Eq.(8). Then, we can use Massey Method, that is, using graph matrix to form a Gram matrix, replacing any one row of Gram matrix by the all-one vector and zeroing out the corresponding *v* element. The ***r*** vector of first five testing sheets are shown in List 4. Although the running time is close to the former method, the accuracy is about 77.58%. See function *OvO_predict_Massey* in Appendix D for the source code.

$$\begin{bmatrix} -1 & 1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{bmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \\ v_9 \\ v_{10} \end{pmatrix} --- Eq.(8)$$

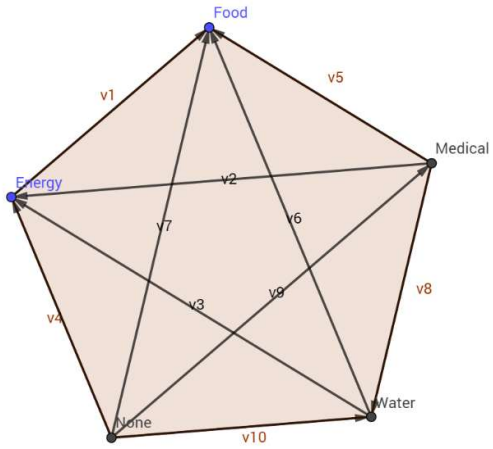

Figure 1: Round Robin Category Tournament

```
[-0.022    0.199 -0.083 -0.09    -0.004]
[ 2.000e-01 -4.735e-05 -9.186e-02 -1.078e-01 -2.405e-04]
[ 7.045e-04 -8.566e-06 -1.726e-01  1.254e-01  4.652e-02]
[-1.694e-03 -1.274e-05  2.000e-01 -1.982e-01 -6.495e-05]
[-0.027    0.2      -0.08    -0.085 -0.008]
```

List 4: ***r*** Vectors of First Five Testing Tweets

Here are some thoughts for the demerits of using Massey Rating Method here. The probability calculated by the Eq.(3) and Laplace Smoothing is too small for the matrix computation. Many probabilities are even smaller than 2.2250738585072014e-308, which is the minimum of float in Python. They are recorded as 0 in the ***v*** vector and influence the final ranking result.

Conclusion

In this paper, we first built up a Naive Bayes Classifier and improve its performance from three aspects—algorithm, language model and strategy. For algorithm part, we used Laplace Smoothing to correct the probability 0 and reduce the influence from the deficiency of training data. For language model part, we attempted to consider the order and coherence of words and used two methods to implement bigram model. However, these two methods, word vector and occurrence probability, did not improve the performance. We concluded the reason to be the particularity of tweets. For strategy part, we changed the former strategy OvA to OvO. We used two methods, including Victory Count Method and Massey Rating Method, to rank the

categories. The performance is improved while it increases the running time. In this situation, simple is powerful. A Naive Bayes Classification with Laplace Smoothing is sufficient to classify the tweets precisely and quickly. If a higher accuracy is demanded, OvO strategy with Victory Count Method is recommended.

P.S.

If you want to run the whole program instead of reading the following messy codes, you can use git to clone the whole program from my GitHub link. There will be a document help you configure the environment.

Clone Link: https://github.com/ZavierLyu/Naive-Bayes-Classifier-for-Tweets.git

Appendices

Appendix A:

This is the source code for the first Naive Bayes Classification with no Laplace Smoothing using unigram model and one-versus-all (OvA) strategy. Running Time: 0.3672s, Accuracy: 40.21%

```
1.  import pandas as pd
2.  import random
3.  from collections import Counter
4.  from nltk.tokenize import TweetTokenizer
5.  import datetime as dt
6.  random.seed(42)
7.
8.
9.  def read_data(filename):
10.     raw_df = pd.read_csv(filename,
11.                          names=["index", "tweets", "category", "type"])
12.     raw_df = raw_df[["tweets", "category", "type"]]
13.     return raw_df
14.
15.
16. def categorize_tweets(df):
17.     df_energy = df[df["category"] == "Energy"]
18.     energy_list = list(df_energy.tweets)
19.     df_energy = df[df["category"] == "Food"]
20.     food_list = list(df_energy.tweets)
21.     df_energy = df[df["category"] == "Medical"]
22.     medical_list = list(df_energy.tweets)
23.     df_energy = df[df["category"] == "Water"]
24.     water_list = list(df_energy.tweets)
25.     df_energy = df[df["category"] == "None"]
26.     none_list = list(df_energy.tweets)
27.     tweets_list = [energy_list, food_list,
28.                    medical_list, water_list, none_list]
29.     return tweets_list
30.
31.
32. def tokenize(tweets_list):
33.     '''''
34.     Return a dictionary {"Energy":[words], "Food":[words]}
35.     '''
36.     tknzr = TweetTokenizer(preserve_case=False)
37.     energy_words, food_words, medical_words, water_words, none_words = [], [], [], [], []
38.     words_list = [energy_words, food_words,
39.                   medical_words, water_words, none_words]
40.     for i in range(len(words_list)):
```

```
        for j in tweets_list[i]:
            words_list[i] += tknzr.tokenize(j)
    dic = {"Energy": energy_words, "Food": food_words,
           "Medical": medical_words, "Water": water_words,
           "None": none_words}
    return dic


def unigram(filename):
    df = read_data(filename)
    tweets_list = categorize_tweets(df)
    words_dic = tokenize(tweets_list)

    train_dic = {}
    for key, value in words_dic.items():
        cnt = Counter(value)
        len_cnt = len(value)
        for word in cnt:
            cnt[word] /= float(len_cnt)
        train_dic[key] = cnt
    # print(train_dic["Energy"])
    return train_dic


def calculate_priori_prob(filename):
    df = read_data(filename)
    len_ = len(df["category"])
    cnt = Counter(list(df["category"]))
    for i in cnt:
        cnt[i] /= len_
    # print(cnt)
    return cnt


def calculate_prob(words, category, train_dic, priori_dic):
    prob = priori_dic[category]
    for i in words:
        if i in train_dic[category]:
            prob *= train_dic[category][i]
        else:
            prob = 0  # without Laplace Smoothing
            break
    prob = float("{0:.3e}".format(prob))
    return (category, prob)


def predict(tweets, train_dic, priori_dic, calculate_prob):
```

```
    tknzr = TweetTokenizer(preserve_case=False)
    words = tknzr.tokenize(tweets)
    energy_prob = calculate_prob(words, "Energy", train_dic, priori_dic)
    food_prob = calculate_prob(words, "Food", train_dic, priori_dic)
    medical_prob = calculate_prob(words, "Medical", train_dic, priori_dic)
    water_prob = calculate_prob(words, "Water", train_dic, priori_dic)
    none_prob = calculate_prob(words, "None", train_dic, priori_dic)
    prob_list = [none_prob, energy_prob, food_prob, medical_prob, water_prob]
    # Randomize the default prediction if all probability is 0
    # to test the robustness of the algorithm.
    random.shuffle(prob_list)

     prob_list = sorted(prob_list, key=lambda x: x[1], reverse=True)
     # print(prob_list)
     return prob_list[0][0]


def testing(filename, train_dic, priori_dic, calculate_prob_func, output_file):
    test_df = read_data(filename)
    test_df["prediction"] = test_df["tweets"].apply(
        lambda row: predict(row, train_dic, priori_dic, calculate_prob_func))
    # print(test_df.head(10))
    total_amount = len(test_df.tweets)
    correct_amount = len(test_df[test_df["category"] == test_df["prediction"]])
    print("Accuracy:", correct_amount/total_amount)
    test_df.drop("type", axis=1, inplace=True)
    test_df.to_csv(output_file)


def main():
    timeStart = dt.datetime.now()
    print("UNIGRAM NO LAPLACE")
    priori_dic_1 = calculate_priori_prob(
        "data\labeled-data-singlelabels-train.csv")
    train_dic_1 = unigram("data\labeled-data-singlelabels-train.csv")
    testing("data\labeled-data-singlelabels-
test.csv", train_dic_1, priori_dic_1, calculate_prob,
            "unigram_no_laplace.csv")
    timeEnd = dt.datetime.now()
    print("Running Time:", str(timeEnd-timeStart))
```

Appendix B:

This is the source code for the second Naive Bayes Classification with Laplace Smoothing using unigram model and one-versus-all (OvA) strategy. Running Time: 0.3341s, Accuracy: 74.38%

```
import pandas as pd
import random
from collections import Counter
from nltk.tokenize import TweetTokenizer
import datetime as dt
random.seed(42)


def read_data(filename):
    raw_df = pd.read_csv(filename,
                         names=["index", "tweets", "category", "type"])
    raw_df = raw_df[["tweets", "category", "type"]]
    return raw_df


def tokenize(tweets_list):
    '''''
    Return a dictionary {"Energy":[words], "Food":[words]}
    '''
    tknzr = TweetTokenizer(preserve_case=False)
    energy_words, food_words, medical_words, water_words, none_words = [], [], [], [], []
    words_list = [energy_words, food_words,
                  medical_words, water_words, none_words]
    for i in range(len(words_list)):
        for j in tweets_list[i]:
            words_list[i] += tknzr.tokenize(j)
    dic = {"Energy": energy_words, "Food": food_words,
           "Medical": medical_words, "Water": water_words,
           "None": none_words}
    return dic


def categorize_tweets(df):
    df_energy = df[df["category"] == "Energy"]
    energy_list = list(df_energy.tweets)
    df_energy = df[df["category"] == "Food"]
    food_list = list(df_energy.tweets)
    df_energy = df[df["category"] == "Medical"]
    medical_list = list(df_energy.tweets)
    df_energy = df[df["category"] == "Water"]
    water_list = list(df_energy.tweets)
    df_energy = df[df["category"] == "None"]
    none_list = list(df_energy.tweets)
    tweets_list = [energy_list, food_list,
                   medical_list, water_list, none_list]
    return tweets_list

```

```
48.
49. def testing(filename, train_dic, priori_dic, calculate_prob_func, output_file):
50.     test_df = read_data(filename)
51.     test_df["prediction"] = test_df["tweets"].apply(
52.         lambda row: predict(row, train_dic, priori_dic, calculate_prob_func))
53.     # print(test_df.head(10))
54.     total_amount = len(test_df.tweets)
55.     correct_amount = len(test_df[test_df["category"] == test_df["prediction"]])
56.     print("Accuracy",correct_amount/total_amount)
57.     test_df.drop("type", axis=1, inplace=True)
58.     test_df.to_csv(output_file)
59.
60.
61. def predict(tweets, train_dic, priori_dic, calculate_prob):
62.     tknzr = TweetTokenizer(preserve_case=False)
63.     words = tknzr.tokenize(tweets)
64.     energy_prob = calculate_prob(words, "Energy", train_dic, priori_dic)
65.     food_prob = calculate_prob(words, "Food", train_dic, priori_dic)
66.     medical_prob = calculate_prob(words, "Medical", train_dic, priori_dic)
67.     water_prob = calculate_prob(words, "Water", train_dic, priori_dic)
68.     none_prob = calculate_prob(words, "None", train_dic, priori_dic)
69.     prob_list = [none_prob, energy_prob, food_prob, medical_prob, water_prob]
70.     # Randomize the default prediction if all probability is 0
71.     # to test the robustness of the algorithm.
72.     random.shuffle(prob_list)
73.
74.     prob_list = sorted(prob_list, key=lambda x: x[1], reverse=True)
75.     # print(prob_list)
76.     return prob_list[0][0]
77.
78.
79. def calculate_priori_laplace(filename, k=1):
80.     df = read_data(filename)
81.     len_number = len(df["category"])
82.     len_kind = len(set(list(df["category"])))
83.     cnt = Counter(list(df["category"]))
84.     for i in cnt:
85.         cnt[i] = (cnt[i]+k) / (len_number+k*len_kind)
86.     # print(cnt)
87.     return cnt
88.
89.
90. def calculate_prob_laplace(words, category, train_dic, priori_dic):
91.     prob = priori_dic[category]
92.     for i in words:
93.         if i in train_dic[category]:
94.             prob *= train_dic[category][i]
```

```
        else:
            prob *= train_dic[category][None]
    prob = float("{0:.3e}".format(prob))
    return (category, prob)


def unigram_laplace(filename, k=1):
    df = read_data(filename)
    tweets_list = categorize_tweets(df)
    words_dic = tokenize(tweets_list)

    train_dic = {}
    len_set = len(set([x for j in words_dic.keys() for x in words_dic[j]]))
    for key, value in words_dic.items():
        cnt = Counter(value)
        len_cnt = len(value)
        for word in cnt:
            cnt[word] = (cnt[word]+k) / (float(len_cnt)+k*len_set)
        cnt[None] = k / (float(len_cnt) + k*len_set)
        train_dic[key] = cnt
    # print(train_dic["Energy"])
    return train_dic


def main():
    timeStart = dt.datetime.now()
    print("UNIGRAM WITH LAPLACE")
    priori_dic_2 = calculate_priori_laplace(
        "data\labeled-data-singlelabels-train.csv", k=1)
    train_dic_2 = unigram_laplace(
        "data\labeled-data-singlelabels-train.csv", k=1)
    testing("data\labeled-data-singlelabels-
test.csv", train_dic_2, priori_dic_2, calculate_prob_laplace,
            "unigram_with_laplace.csv")
    timeEnd = dt.datetime.now()
    print("Running Time:", str(timeEnd-timeStart))


if __name__ == "__main__":
    main()
```

Appendix C:

This is the source code for the second Naive Bayes Classification with Laplace Smoothing using bigram model and one-versus-all (OvA) strategy.

1. First method: Running Time: 0.3087s, Accuracy: 69.04%

```
from nltk import bigrams
import string
import random
from basis import read_data, categorize_tweets
from nltk.tokenize import TweetTokenizer
from collections import defaultdict, Counter
from laplace import calculate_priori_laplace, calculate_prob_laplace
import datetime as dt


def removePunctuation(tweet):
    translator = tweet.maketrans('', '', string.punctuation)
    return tweet.translate(translator)


def bigramReturner(tweetString):
    # tweetString = tweetString.lower()
    # tweetString = removePunctuation(tweetString)
    tknzr = TweetTokenizer(preserve_case=False)
    words = tknzr.tokenize(tweetString)
    bigramFeatureVector = []
    for item in bigrams(words):
        bigramFeatureVector.append(' '.join(item))
    return bigramFeatureVector


def bigram_tokenize(tweets_list):
    '''''
    Return a dictionary {"Energy":[words], "Food":[words]}
    '''
    energy_words, food_words, medical_words, water_words, none_words = [], [], [], [], []
    words_list = [energy_words, food_words,
                  medical_words, water_words, none_words]
    for i in range(len(words_list)):
        for j in tweets_list[i]:
            words_list[i] += bigramReturner(j)
    dic = {"Energy": energy_words, "Food": food_words,
           "Medical": medical_words, "Water": water_words,
           "None": none_words}
    return dic


def bigram_laplace(filename, k=1):
    df = read_data(filename)
    tweets_list = categorize_tweets(df)
    words_dic = bigram_tokenize(tweets_list)

```

```
    train_dic = {}
    len_set = len(set([x for j in words_dic.keys() for x in words_dic[j]]))
    for key, value in words_dic.items():
        cnt = Counter(value)
        len_cnt = len(value)
        for word in cnt:
            cnt[word] = (cnt[word]+k) / (float(len_cnt)+k*len_set)
        cnt[None] = k / (float(len_cnt) + k*len_set)
        train_dic[key] = cnt
    # print(train_dic["Energy"])
    return train_dic


def bigram_predict(tweets, train_dic, priori_dic, calculate_prob):
    words = bigramReturner(tweets)
    energy_prob = calculate_prob(words, "Energy", train_dic, priori_dic)
    food_prob = calculate_prob(words, "Food", train_dic, priori_dic)
    medical_prob = calculate_prob(words, "Medical", train_dic, priori_dic)
    water_prob = calculate_prob(words, "Water", train_dic, priori_dic)
    none_prob = calculate_prob(words, "None", train_dic, priori_dic)
    prob_list = [none_prob, energy_prob, food_prob, medical_prob, water_prob]
    # Randomize the default prediction if all probability is 0
    # to test the robustness of the algorithm.
    random.shuffle(prob_list)

    prob_list = sorted(prob_list, key=lambda x: x[1], reverse=True)
    # print(prob_list)
    return prob_list[0][0]


def bigram_testing(filename, train_dic, priori_dic, calculate_prob_func, output_file):
    test_df = read_data(filename)
    test_df["prediction"] = test_df["tweets"].apply(
        lambda row: bigram_predict(row, train_dic, priori_dic, calculate_prob_func))
    # print(test_df.head(10))
    total_amount = len(test_df.tweets)
    correct_amount = len(test_df[test_df["category"] == test_df["prediction"]])
    print("Accuracy:", correct_amount/total_amount)
    test_df.drop("type", axis=1, inplace=True)
    test_df.to_csv(output_file)


def main():
    timeStart = dt.datetime.now()
    print("BIGRAM WITH LAPLACE METHOD 1")
    priori_dic_3 = calculate_priori_laplace(
        "data\labeled-data-singlelabels-train.csv")
```

```
    train_dic_3 = bigram_laplace("data\labeled-data-singlelabels-train.csv")
    bigram_testing("data\labeled-data-singlelabels-test.csv", train_dic_3, priori_dic_3,
                   calculate_prob_laplace, "bigram_no_laplace.csv")
    timeEnd = dt.datetime.now()
    print("Running Time:", str(timeEnd-timeStart))

if __name__ == "__main__":
    main()
```

2. Second method: Running Time: 3.538s, Accuracy: 68.68%

```
from nltk import bigrams
import string
from collections import defaultdict
import pandas as pd
import random
from collections import Counter
from nltk.tokenize import TweetTokenizer
import datetime as dt
random.seed(42)


def removePunctuation(tweet):
    translator = tweet.maketrans('', '', string.punctuation)
    return tweet.translate(translator)


def read_data(filename):
    raw_df = pd.read_csv(filename,
                         names=["index", "tweets", "category", "type"])
    raw_df = raw_df[["tweets", "category", "type"]]
    return raw_df


def categorize_tweets(df):
    df_energy = df[df["category"] == "Energy"]
    energy_list = list(df_energy.tweets)
    df_energy = df[df["category"] == "Food"]
    food_list = list(df_energy.tweets)
    df_energy = df[df["category"] == "Medical"]
    medical_list = list(df_energy.tweets)
    df_energy = df[df["category"] == "Water"]
    water_list = list(df_energy.tweets)
    df_energy = df[df["category"] == "None"]
    none_list = list(df_energy.tweets)
    tweets_list = [energy_list, food_list,
                   medical_list, water_list, none_list]
```

```
37.     return tweets_list
38. 
39. 
40. def bigramPacker(tweetString):
41.     tknzr = TweetTokenizer(preserve_case=False)
42.     words = tknzr.tokenize(tweetString)
43.     bigram_list = list(bigrams(words, pad_right=True, pad_left=True))[1:-1]
44.     return bigram_list
45. 
46. 
47. def bigram_tokenize_2(tweets_list):
48.     energy_words, food_words, medical_words, water_words, none_words = [], [], [], [], []
49.     words_list = [energy_words, food_words,
50.                   medical_words, water_words, none_words]
51.     for i in range(5):
52.         for j in tweets_list[i]:
53.             words_list[i] += bigramPacker(j)
54. 
55.     energy_counts = bigram_counter_2(words_list[0])
56.     food_counts = bigram_counter_2(words_list[1])
57.     medical_counts = bigram_counter_2(words_list[2])
58.     water_counts = bigram_counter_2(words_list[3])
59.     none_counts = bigram_counter_2(words_list[4])
60. 
61.     dic = {"Energy": energy_counts, "Food": food_counts,
62.            "Medical": medical_counts, "Water": water_counts,
63.            "None": none_counts}
64.     return dic
65. 
66. 
67. def bigram_counter_2(bigram_list):
68.     bigram_counts = defaultdict(lambda: Counter())
69.     for w1, w2 in bigram_list:
70.         bigram_counts[w1][w2] += 1
71.     return bigram_counts
72. 
73. 
74. def calculate_priori_laplace(filename, k=1):
75.     df = read_data(filename)
76.     len_number = len(df["category"])
77.     len_kind = len(set(list(df["category"])))
78.     cnt = Counter(list(df["category"]))
79.     for i in cnt:
80.         cnt[i] = (cnt[i]+k) / (len_number+k*len_kind)
81.     # print(cnt)
82.     return cnt
83. 
```

```
84.
85. def output_train_dic(filename):
86.     df = read_data(filename)
87.     tweets_list = categorize_tweets(df)
88.     train_dic = bigram_tokenize_2(tweets_list)
89.     return train_dic
90.
91.
92. def calculate_prob_bigram_2(words, category, train_dic, priori_dic):
93.
94.     prob = priori_dic[category]
95.     start_word = words[0][0]
96.     total_amount = 0
97.     start_amount = 0
98.     for c in train_dic.keys():
99.         for w1 in train_dic[c].keys():
100.             total_amount += sum(train_dic[c][w1].values())
101.             if (w1 == start_word) and (c == category):
102.                 start_amount += sum(train_dic[c][w1].values())
103.
104.     P0 = (start_amount + 1) / total_amount
105.     prob = prob * P0
106.     for bigram_tuple in words:
107.         w1 = bigram_tuple[0]
108.         w2 = bigram_tuple[1]
109.         N = 1000
110.         if w1 in train_dic[category].keys():
111.             prob *= (train_dic[category][w1][w2] + 1) / \
112.                 (sum(train_dic[category][w1].values())+N)
113.         else:
114.             prob *= 1 / N
115.     prob = float("{0:.3e}".format(prob))
116.     return (category, prob)
117.
118.
119. def bigram_predict_2(tweets, train_dic, priori_dic, calculate_prob):
120.     words = bigramPacker(tweets)
121.     energy_prob = calculate_prob(words, "Energy", train_dic, priori_dic)
122.     food_prob = calculate_prob(words, "Food", train_dic, priori_dic)
123.     medical_prob = calculate_prob(words, "Medical", train_dic, priori_dic)
124.     water_prob = calculate_prob(words, "Water", train_dic, priori_dic)
125.     none_prob = calculate_prob(words, "None", train_dic, priori_dic)
126.     prob_list = [none_prob, energy_prob, food_prob, medical_prob, water_prob]
127.     # Randomize the default prediction if all probability is 0
128.     # to test the robustness of the algorithm.
129.     random.shuffle(prob_list)
130.     prob_list = sorted(prob_list, key=lambda x: x[1], reverse=True)
```

```
    # print(prob_list)
    return prob_list[0][0]


def testing(filename, train_dic, priori_dic, calculate_prob_func, output_file):
    test_df = read_data(filename)
    test_df["prediction"] = test_df["tweets"].apply(
        lambda row: bigram_predict_2(row, train_dic, priori_dic, calculate_prob_func))
    total_amount = len(test_df.tweets)
    correct_amount = len(test_df[test_df["category"] == test_df["prediction"]])
    print("Accuracy:", correct_amount/total_amount)
    test_df.drop("type", axis=1, inplace=True)
    test_df.to_csv(output_file)


def main():
    timeStart = dt.datetime.now()
    print("BIGRAM WITH LAPLACE METHOD 2")
    priori_dic = calculate_priori_laplace(
        "data\labeled-data-singlelabels-train.csv")
    train_dic = output_train_dic("data\labeled-data-singlelabels-train.csv")
    testing("data\labeled-data-singlelabels-test.csv", train_dic, priori_dic,
            calculate_prob_bigram_2, "bigram_2_laplace")
    timeEnd = dt.datetime.now()
    print("Running Time:", str(timeEnd-timeStart))

if __name__ == "__main__":
    main()
```

Appendix D:

This is the source code for the second Naive Bayes Classification with Laplace Smoothing using unigram model and one-versus-one (OvO) strategy.

Victory Count Method (`OvO_predict_1`): Running Time: 43.41s, Accuracy: 89.68%

Masse Method (`OvO_predict_Massey`): Running Time: 39.28s, Accuracy: 77.58%

```
import pandas as pd
import random
from collections import Counter
from nltk.tokenize import TweetTokenizer
import numpy as np
import datetime as dt
random.seed(42)
np.set_printoptions(precision=3)


```

```
def read_data(filename):
    raw_df = pd.read_csv(filename,
                         names=["index", "tweets", "category", "type"])
    raw_df = raw_df[["tweets", "category", "type"]]
    return raw_df


def binary_classifier(c1, c2, df, tweet):
    sub_df = df[df["category"].isin([c1, c2])]
    c1_df = df[df["category"] == c1]
    c2_df = df[df["category"] == c2]
    c1_sentences = list(c1_df["tweets"])
    c2_sentences = list(c2_df["tweets"])
    tknzr = TweetTokenizer(preserve_case=False)
    c1_words = []
    c2_words = []
    for sent in c1_sentences:
        c1_words += tknzr.tokenize(sent)
    for sent in c2_sentences:
        c2_words += tknzr.tokenize(sent)
    c1_count = Counter(c1_words)
    c2_count = Counter(c2_words)

    N_word = len(set(c1_words+c2_words))
    for w in c1_count.keys():
        c1_count[w] = (c1_count[w]+1) / (len(c1_words)+N_word)
    c1_count[None] = 1/(len(c1_words)+N_word)
    for w in c2_count.keys():
        c2_count[w] = (c2_count[w]+1) / (len(c2_words)+N_word)
    c2_count[None] = 1/(len(c2_words)+N_word)

    tweet_words = tknzr.tokenize(tweet)

    len_lines = len(sub_df["category"])
    N = 2
    cnt = Counter(list(df["category"]))
    for i in cnt:
        cnt[i] = (cnt[i]+1) / (len_lines+1*N)

    c1_prob = calculate_prob(tweet_words, c1, c1_count, cnt)
    c2_prob = calculate_prob(tweet_words, c2, c2_count, cnt)

    # assert c1_prob != c2_prob, "IMPOSSIBLE!"
    if c1_prob > c2_prob:
        return (c1, c1_prob-c2_prob)
    else:
        return (c2, c2_prob-c1_prob)
```

```
58.
59.
60. def calculate_prob(tweet_words, category, count, cnt):
61.     prob = cnt[category]
62.     for word in tweet_words:
63.         if word in count:
64.             prob *= count[word]
65.         else:
66.             prob *= count[None]
67.     return prob
68.
69.
70. def OvO_predict_1(df, tweet):
71.     tags = ["Energy", "Food", "Medical", "Water", "None"]
72.     result_df = pd.DataFrame(np.zeros([5, 5]), columns=tags, index=tags)
73.     round_score_list = []
74.     for c_1 in range(len(tags)):
75.         for c_2 in range(c_1+1, len(tags)):
76.             category_1 = tags[c_1]
77.             category_2 = tags[c_2]
78.             prediction = binary_classifier(category_1, category_2, df, tweet)
79.             if prediction[0] == category_1:
80.                 result_df[category_1][category_2] = 1
81.                 result_df[category_2][category_1] = -1
82.             elif prediction[0] == category_2:
83.                 result_df[category_2][category_1] = 1
84.                 result_df[category_1][category_2] = -1
85.             round_score_list.append(prediction[1])
86.     category_score = [(tags[i], sum(list(result_df[tags[i]])))
87.                       for i in range(5)]
88.     random.shuffle(category_score)
89.     category_score = sorted(category_score, key=lambda x: x[1], reverse=True)
90.     # print(category_score)
91.     return category_score[0][0]
92.
93.
94. def OvO_predict_Massey(df, tweet):
95.     tags = ["Energy", "Food", "Medical", "Water", "None"]
96.     round_score_list = []
97.     B = np.zeros([10, 5])
98.     cnt = 0
99.     for c_1 in range(len(tags)):
100.         for c_2 in range(c_1+1, len(tags)):
101.             category_1 = tags[c_1]
102.             category_2 = tags[c_2]
103.             prediction = binary_classifier(category_1, category_2, df, tweet)
104.             if prediction[0] == category_1:
```

```
                B[cnt][c_1] = 1
                B[cnt][c_2] = -1
            elif prediction[0] == category_2:
                B[cnt][c_1] = -1
                B[cnt][c_2] = 1
            round_score_list.append(prediction[1])
            cnt += 1
    round_score_array = np.array(round_score_list)
    if np.linalg.norm(round_score_array) == 0:
        return tags[random.randint(0, 4)] # When all the elements of v vector are 0
    else:
        round_score_array = round_score_array / \
            np.linalg.norm(round_score_array, ord=1)  # Normalization

    v = round_score_array
    # Using Massey method to rank the categories.
    G = np.dot(B.T, B)
    P = np.dot(B.T, v)
    G[-1] = 1
    P[-1] = 0
    G_I = np.linalg.inv(G)
    r = np.dot(G_I, P) # Get the index of the max value.
    index = np.where(r == r.max())[0][0]
    return tags[index]


def testing(test_file, train_file, output_file, predict_func):
    print("OvO strategy")
    timeStart = dt.datetime.now()
    test_df = read_data(test_file)
    train_df = read_data(train_file)
    test_df["prediction"] = test_df["tweets"].apply(
        lambda row: predict_func(train_df, row))
    total_amount = len(test_df.tweets)
    correct_amount = len(test_df[test_df["category"] == test_df["prediction"]])
    print("Accuracy:", correct_amount/total_amount)
    test_df.drop("type", axis=1, inplace=True)
    test_df.to_csv(output_file)
    timeEnd = dt.datetime.now()
    print("Running Time:", str(timeEnd-timeStart))


if __name__ == "__main__":
    testing(
        test_file="data\labeled-data-singlelabels-test.csv",
        train_file="data\labeled-data-singlelabels-test.csv",
```

```
        output_file="OvO_laplace.csv",
        predict_func=OvO_predict_1 #Change function to OvO_predict_Massey to use graph ranking
    )
```