Sentiment about China in Foreign Newspapers during COVID-19

Shu Zixuan
shuzx@shanghaitech.edu.cn

Wei Zhiyuan
weizhy@shanghaitech.edu.cn

Jiang Haoran
jianghr@shanghaitech.edu.cn

Li Xinni
lixn@shanghaitech.edu.cn

Lu Chenhao
luchh@shanghaitech.edu.cn

Abstract

In this project, we apply the knowledge of machine learning to classifying the emotions of news during the pandemic. To realize this, we take advantage of some external libraries for translating and summarizing the news to clean data, write and train the Naive Bayes Model on our own, and use F1 score for evaluation.

1. Introduction

During the outbreak of COVID-19, China's epidemic situation has attracted wide attention from around the world. Mainstream news media at home and abroad have covered China's epidemic response, prevention and control extensively, including both commendation and criticism.

1.1. Data Set

Overall, we have collected 2199 different news as the whole data set. Each news text has a label $\in \{0, 1\}$ whose value stands for the property of the news. Here 1 means positive, and 0 means negative.

We separate them into two parts, namely the training set and the test set. Currently we take the last 400 news as the test set and the previous 1799 as the training set.

1.2. Task Description

Machine Learning is used to complete the emotion prediction of news data. Given the news text, then corresponding emotion categories will be output.

1.3. Evaluation Metrics

F1 score. The rules are as follows.

	Retrieved	**Not Retrieved**
Relevant	True Positives (TP)	False Negatives (FN)
NonRelevant	False Positives (FP)	True Negatives (TN)

$$Precision = \frac{TP}{TP + FP} * 100\%$$

$$Recall = \frac{TP}{TP + FN} * 100\%$$

F1 score is the harmonic-mean of Precision and Recall:

$$F1 = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}}$$

2. Algorithm Design

2.1. Data Processing

2.1.1 Language Type Determination

Use the *langdetect* library to detect the type of text language, and store the language type and the number of occurrences of such kind of news in a dictionary. The code looks like this:

```
from langdetect import detect
import json
language_dic={}
i=1
for data in train_data:
    language=detect(data['content'])
    print(i,end='')
    print(language)
    i+=1
    if(language_dic.get(language)):
            language_dic[language]+=1
    else:
            language_dic[language]=1
print(language_dic)
```

The result obtained is shown in Figure 1:

```
{'es': 54, 'pt': 556, 'en': 1405, 'de': 162, 'ru': 21, 'af': 1, 'zh-cn': 1}
```

Figure 1. language type of train set

Then use the same method to detect the language types of the test set, and the results are shown in Figure 2:

```
{'pt': 222, 'en': 605, 'de': 45, 'es': 31, 'ru': 4}
```

Figure 2. language type of test set

es, pt, en, de, ru, af, zh-ch are the abbreviations of *langdetect* for different language types, which represent Spanish, Portuguese, English, German, Russian, Afrikaans, and Chinese respectively. It can be seen that about 3/4 of the languages types are English, 1/4 of them are Portuguese, and other languages account for a smaller proportion.

2.1.2 Text Translation

Obviously, it is hard for the training set in one language to assist the test set in another language, and it may even cause interference, so it is necessary to translate all the news texts in the training set into the same language. Given that there are more language processing libraries in English, and the original language of the training and test sets is mostly English, the *translate* library can be called to translate all the news texts into English. The specific code implementation is shown in the program implementation section.

2.1.3 Text Division

The next step is to do a series of data cleaning. First of all, we need to decompose a news text into individual words, i.e. to convert a string of text into a list composed of words, here we use the **word_tokenize()** function in *NLTK* library to divide the text into words, the code is as follows:

```
words_list=nltk.word_tokenize(content)
```

2.1.4 Lexical annotation

Next, the lexical annotation of each word is done to obtain more accurate results in the later word normalization. The *NLTK* library's **pos_tag()** function is used to tag the list after word separation, resulting in a list consisting of a binary (string word, string tag), as shown in the following code. In addition, the *NLTK* library uses different conventions to tag words, and these words form the lexical tagging set, such as "JJ" for adjectives, "RB" for adverbs, etc.

```
pos_tag_list=nltk.pos_tag(words_list)
```

2.1.5 Word type normalization

In English, a word is often a variant of another word, for example, looked is the past tense of look, finds is the triple singular form of find, and so on. These words will be considered as different words in the program by string comparison, but in fact they express the same meaning. So, by normalizing the different forms of a word, the interference from root affixes and some deformations can be cleaned out.

The process of lexical normalization consists of two main types:

1. stem extraction: remove the affixes that do not influence the morphology and get the stem of the word

2. lexical reduction: capturing the normalized word form based on the root

here are several stem extraction methods provided in *NLTK* library, such as **Porter Stemmer** and **Lancaster Stemmer**. In addition, a powerful reduction module using *WordNetizer* class to obtain word roots is also provided in *NLTK* library.

The **lemmatize()** method in the *WordNetLemmatizer* class is used here to obtain the basic form of the word. The result is compared with the *WordNet* corpus and affixes are removed recursively until a match is found in the lexical network, eventually playing the basic form of the input word, or if no match is found, the input word is returned directly without any changes.

It is worth noting that the **lemmatize()** method does not necessarily restore words as one would expect without qualifications. For example, when we type "went", we might expect it to be reduced to "go". However, sometimes in reality it is a person's name, so it is not expected to be reduced to "go". In order to achieve better accuracy, we need to specify the lemma of the word, so when calling **lemmatize()**, we pass the lemma from **2.1.5** into the optional **pos** parameter, and the corresponding code is as follows:

```
word_net_obj=WordNetLemmatizer()
normalize_words_list=[]
for word_tag in pos_tag_list:
    if word_tag[1]=='JJ':
        normalize_words_list.append(
            word_net_obj.lemmatize(
            word_tag[0], 'a'))
    elif (word_tag[1]=='VB' or word_tag
        [1]=='VBD' or word_tag[1]=='VBG'
        or word_tag[1]=='VBN'):
        normalize_words_list.append(
            word_net_obj.lemmatize(
            word_tag[0], 'v'))
    elif (word_tag[1]=='NN' or word_tag
        [1]=='NNS' or word_tag[1]=='NNP'
        or word_tag[1]=='VBN'):
        normalize_words_list.append(
            word_net_obj.lemmatize(
            word_tag[0], 'n'))
    elif word_tag[1]=='RB':
        normalize_words_list.append(
```

```
            word_net_obj.lemmatize(
            word_tag[0], 'r'))
    else:
        normalize_words_list.append(
            word_net_obj.lemmatize(
            word_tag[0]))
```

2.1.6 Stop Words Filtering

Stop words are any word in a stop list which are filtered out before or after processing of natural language data in order to save space and improve retrieval efficiency during information retrieval. In machine learning, doing so can also reduce the interference in prediction results caused by words that appear in large numbers in all texts.

A standard list of stop words is available in the *NLTK* library, and words are filtered by comparing and deleting words that are in the list, as shown in the following code:

```
remind_word_list=[]
for word in normalize_words_list:
    if word not in stop_word_list:
            remind_word_list.append(
                word)
```

2.2. Model Algorithm

A Naive Bayes classifier was used to train the model, and the specific model algorithm is as follows.

1. Featuring: Let $x_\alpha = j$ be the j_{th} occurrence of the α_{th} word in the dictionary in news x. By counting each word in the text, news x can be transformed into a word vector of the form $x = \{x_1, x_2, x_3 \cdots x_d\}$, where d is the number of words in the dictionary.

2. Assuming that the probability of occurrence of each word in the text is independent, write the probability of occurrence of the word vector $x = \{x_1, x_2, x_3 \cdots x_d\}$ in a news item of length m and label c using a polynomial distribution:

$$P(x_j|y=c) = \frac{m!}{x_1! \cdot x_2! \cdot x_3! \cdots x_d} \prod_{\alpha=1}^{d} (\theta_{\alpha c})^{x_\alpha}$$

where $\theta_{\alpha c}$ is the probability of occurrence of x_α in the news with label c. $\frac{m!}{x_1! \cdot x_2! \cdot x_3! \cdots x_d}$ and $\sum_{\alpha=1}^{d} \theta_{\alpha c} = 1$ are permutations of the text composed by c.

3. The expression for $\theta_{\alpha c}$ is given next:

$$\theta_{\alpha c} = \frac{\sum_{i=1}^{n} I(y_i = c) x_{i\alpha} + 1}{\sum_{i=1}^{n} I(y_i = c) m_i + I \cdot d}$$

where $\sum_{i=1}^{n} I(y_i = c) m_i$ is the total number of words in all news with label = c and $\sum_{i=1}^{n} I(y_i = c) x_{i\alpha}$ is the total number of occurrences of $x_{i\alpha}$ in all news with label = c.

It should also be noticed that the train set is finite, which can bring limitations to the results. For example, the word x_i does not appear in all the news with label = c, so $\theta_{ic} = 0$. Consider a news X containing the word x_i, and substituting θ_{ic} into the equation of $P(x|m, y = c)$ yields a probability of 0 for this news label=c. This is obviously unreasonable, because even articles with the same opinion may have completely different wording. Therefore, the smoothing parameter I is introduced, assuming that the probability of occurrence of each word is the same in news with any label before training with the train set, so I is added to the numerator and $I \cdot d$ is added to the denominator, and the size of I depends on the degree of smoothing.

4. prediction: the expression of the prediction function is:

$$h(x) = argmax_y P(x|m, y = c) P(y)$$

substitute θ into $P(x|m, y = c)$, get the prediction function:

$$h(x) = argmax_c \frac{m!}{x_1! \cdot x_2! \cdot x_3! \cdots x_d} \prod_{\alpha=1}^{d} (\theta_{\alpha c})^{x_\alpha} P(y = c)$$

Substituting into the $P(y = c) = \frac{\sum_{i=1}^{n} I(y_i = c)}{n}$ and removing the term unrelated to y, the final prediction function is obtained as:

$$h(x) = argmax_c \frac{\sum_{i=1}^{n} I(y_i = c)}{n} \prod_{\alpha=1}^{d} (\theta_{\alpha c})^{x_\alpha}$$

2.3. Model Evaluation

Naive Bayes is a simple classifier, whose input is a vector while the output is the classification of each label in the data set. The purpose of this topic is to divide news into positive and negative types, which is a binary classification problem. It can fit the naive Bayes model well by converting news text into word vector.

Indeed, as mentioned in **2.2**, naive Bayes is based on the assumption that the probability distribution of each dimension of the vector is independent. Such assumption is obviously untenable. Because of the logic between words of the texts and the existence of a large number of synonyms, the probability distribution of a word will be affected by other words. Such bold assumptions will have some negative effects on the accuracy of classification. Nevertheless, in practical application, naive Bayes classifier can produce relatively good results. Therefore, the naive Bayes training model is still selected for this topic.

Finally, as mentioned in **2.2**, due to the characteristics of texts, it is necessary to introduce smoothing parameters, and their specific values need to be determined through debugging. In the experiment, we use the Laplace Smoothing with parameter *k* between 1 to 10.

2.4. Model Merging

We write the Naive Bayes Model on our own. Then since the *sklearn* library has its own function of text vectorization, after data cleaning, the list composed of words should be re-combined into a cleaned text and sent into the vectorization function, so as to integrate data cleaning and model training.

3. Environment and Platform

3.1. Hardware

Figure 3. macOS 11.4

3.2. python

Figure 4. python 3.8.3

4. Program Implementation

4.1. language_detect.py

The language detection part has been shown in **Algorithm Design** and will not be described here.

4.2. translate.py

The implementation of the translation part calls *pygtrans* library. Its advantage is that the word limit of each call is very large, so there is no need to segment the text, and it can be continuously called without interruption. The core code is as follows.

```python
with open("train_translate.json", 'a') as  f:
    # f.write('[')

    client = Translate()
    translate_train_data=[] # list
    i=1363
    for data in train_data[1364:]:
        translate_data_dic={}   # dic
        translated_text=client.translate(data['content'], target='en')
        while translated_text == Null:
            translated_text=client.translate(data['content'], target='en')
        translate_data_dic['content']=translated_text.translatedText
        translate_data_dic['label']=data['label']
        json.dump(translate_data_dic, f)
        i+=1
        print(i)
    f.write(']')


with open("test_translate.json", 'w') as  f:
    f.write('[')
    translate_test_data=[]  # list
    i=0
    for data in test_data:
        translate_test_dic={}   # dic
        translated_text=client.translate(data['content'], target='en')
        while translated_text == Null:
            translated_text=client.translate(data['content'], target='en')
        translate_test_dic['content']=translated_text.translatedText
        translate_test_dic['label']=data['label']
        json.dump(translate_test_dic, f)
        i+=1
        print(i)
    f.write(']')
```

Figure 5. translate.py

4.3. naive_bayes.py

This is another core part of the project. We write the classifier on our own, and the main code is as follows.

```python
    def fit(self, feature, label, laplace=1):
        '''
        train the model and save probabilities in self.label_prob and self.condition_prob represently
        :param feature: ndarray contains all features in training set
        :param label:ndarray contains all labels in training set
        :return: void
        '''

        #******** Begin ********#
        trainNum=len(label)
        featureNum=len(feature[0])

        labelDic={}
        for l in label:
            labelDic[l]=labelDic.get(l, 0)+1

        for lTemp,nTemp in labelDic.items():
            self.label_prob[lTemp]=(nTemp+1)/(trainNum+len(labelDic))

        labelFeatureDic={}
        for i in range(len(label)):
            if labelFeatureDic.get(label[i]):
                labelFeatureDic[label[i]].append(feature[i])
            else:
                labelFeatureDic[label[i]]=[feature[i]]

        #feature num for each colum
        featureTypeNum=[]
        featureTypeDic={}
        for i in range(featureNum):
            typeDic={}
            for fea in feature:
                typeDic[fea[i]]=1
            featureTypeNum.append(len(typeDic))
            featureTypeDic[i]=typeDic

        for lTemp,feasTemp in labelFeatureDic.items():
            lTempNum=len(feasTemp)
            helpDic1={}
            for i in range(featureNum):
                helpDic2={}
                for feas in feasTemp:
                    helpDic2[feas[i]]=helpDic2.get(feas[i], 0)+1
                # add feature types that haven't appeared
                for featType in featureTypeDic[i].keys():
                    if not helpDic2.get(featType):
                        helpDic2[featType]=0
                for feaTemp, nTemp in helpDic2.items():
                    helpDic2[feaTemp]=(nTemp+laplace)/(laplace * lTempNum+featureTypeNum[i])
                helpDic1[i]=helpDic2
            self.condition_prob[lTemp]=helpDic1
```

Figure 6. naive$_b ayes.py$

4.4. predict.py

The data cleaning part has been shown in **Algorithm Design** and will not be described here.

When it comes to the model training part, we use **PCA** to reduce the dimension to 100. The core code is as follows.

```
def news_predict(train_sample, train_label, test_sample):
    '''
    training model and predict then return the result
    :param train_sample: news context in original training set <ndarray>
    :param train_label: the corresponding labels in training set <ndarray>
    :param test_sample: news context in original testing set <ndarray>
    :return predict result <ndarray>
    '''
    # instantiate the vectorizer
    vec = CountVectorizer()
    # vectorize the news in training set
    X_train_count_vectorizer = vec.fit_transform(train_sample).toarray()
    # vectorize the news in test set
    X_test_count_vectorizer = vec.transform(test_sample).toarray()

    # instantiate the tf-idf object
    #tfidf = TfidfTransformer()
    # transfer the frequency vectors in training set by tf-idf
    #X_train = tfidf.fit_transform(X_train_count_vectorizer)
    # transfer the frequency vectors in test set by tf-idf
    #X_test = tfidf.transform(X_test_count_vectorizer)

    all_data = np.vstack((X_train_count_vectorizer, X_test_count_vectorizer))

    pca = PCA(n_components = 100)
    principalComponents = pca.fit_transform(all_data)

    X_train_count_vectorizer = principalComponents[:-test_num]
    X_test_count_vectorizer = principalComponents[-test_num:]

    clf = NaiveBayesClassifier()
    #clf = BernoulliNB()
    #clf = SVC()
    clf.fit(X_train_count_vectorizer, train_label, 2)   # the 3rd arg is k for Laplacian Smoothing
    #clf.fit(X_train_count_vectorizer, train_label)
    result = clf.predict(X_test_count_vectorizer)

    return result
```

Figure 7. predict.py

5. Results Analysis

Running *predict.py*, we can get a result that:

$$TP = 368, FP = 2, FN = 30$$

$$F1\_score = 0.9583333333333334$$

Good prediction results are obtained through data cleaning and model training based on Naive Bayes. The workload and improvement mainly lies in the following two aspects.

5.1. text data cleaning

Because the initial texts contain multiple languages, mixed parts of speech of words, and a large number of moot words, text cleaning is very important. The F1-score without text cleaning is about 0.6. After the introduction of text cleaning, the prediction effect of the model is greatly improved.

5.2. train model for prediction

First of all, it is very important to select an appropriate model. Considering that this is a binary classification problem and features can be transformed into vectors, Naive Bayes classifier is selected. The results prove that it is good.

Secondly, the importance of parameter tuning is self-evident in the process of model training. The smoothing factor has a great influence on the experimental results.

6. Reflections

Currently the model is not perfect. To make an intuitive comparison, we substitute the written *Naive Bayes* classifier with the **BernoulliNB** and **support vector classifier (SVC)**. We call them separately from *sklearn*.**naive_bayes** and *sklearn*.**svm** in line 93-97 in *predict.py*. We set the smoothing parameter *alpha = 0.03* and record the running time. The outcomes are as follows.

	TP	FP	FN	F1-score	Time(ms)
ours	368	2	30	0.958333	326.9
Bnb	358	12	23	0.953395	4.6
SVC	368	2	26	0.963350	61.2

As we can see from the table, our NB classifier has an acceptable ability of predicting positive cases, but the number of **False Negatives (FN)** shows that it needs better calibration for predicting negative news. The reasons might be that when deleting stop words, the program will sometimes filter those negative words such as *not*, *no*, which will probably affect the meanings of some sentences that convey negative intentions in some news. Also, the various styles of writing of different news may affect the prediction to some extent.

In addition, the running time of our classifier is relatively long, so its working efficiency needs to be increased.

To sum up, In the future, we'd definitely like to promote the algorithm for clearer summarizing and improve the model for more accurate classification and less time consumed.

7. External Libraries

1. *langdetect*
   (a) **detect()**: to detect number of text language types.
2. *pygtrans*
   (a) **Translate()**: to translate news from different languages into English.
3. *NLTK*
   (a) **word_tokenize()**: to performs word segmentation on the text
   (b) **pos_tag()**: to get the part of speech tagging of each part.
   (c) **WordNetLemmatizer.lemmatize()**: to get the basic form of each word.
   (d) **corpus.stopwords**: to detect stopwords(which are noises when analyzing) in filtering step.
4. *sklearn*

(a) **feature_extraction.text.CountVectorizer**(): to vectorize the news in the data set

(b) **decomposition.PCA**: to reduce the dimension of data and improve efficiency

(c) **naive_bayes**: to compare with our Naive Bayes model

(d) **svm**: to compare with our Naive Bayes model