Assignment 1: movie reviews sentiment analysis

Yumin, a1754926
Supervisor: LingQiao Liu

1. Text preprocessing

1.1 load the data

I download this Stanford movie data in my local location "/Users/cao.yumin/Desktop/UA第二学期/NLP/imdb_master.csv", and third package – pandas was used for load the data. However, the data cannot be read with **'UTF-8'** for Latin and other special words/ characters, so I use **'ISO-8859-1'** to encode the data. After reading, I removed the data with label 'unsup'.

1.2 train test split

After loading, the data was split into x_train, y_train, x_test, y_test.

1.3 Lower

Lower function was used to reduce words dimension, because words with upper and lower case have the same meaning and have no effect to the sentiment analysis.

1.4 HTML

Special html characters like <br \> are remained in the texts, so I use beautifulsoup package to handle this html problem.

1.5 URL

Along with html, url link is another problem when we craw online data. I use RE(regularisation) package to remove the url link.

1.6 Accented characters

When we have a view of the data, we can find accented characters like @, ©, ℗, ® and so on, I used unicodedata package to remove those accented characters.

1.7 Stopwords

For stop words, I import the stopwords package from nltk, however, **this stop words list cannot be directly used**. Our further processing need to remain negation words **'not'** and **'no'**, so I removed these words from the original stop words list.

What's more, some high frequency but meaningless words (or to say has no effect to the sentiment analysis) like **'film', 'movie', 'actor', 'actress'** in the movie area should also be removed from the text, so I add these words into the stop words list.

1.8 Extra white space

RE function was used for remove the extra white space.

1.9 First punctuation: remove ' _ '

I plan to use a simple RE function '[^\w\s_]' to remove punctuation, however this function would not remove words like '_he', '_good_' and so on. Considering that we need to add words like 'NOT_like' in the negation step, and the negation step should be processed before the second punctuation step (because **negation step need to know where to stop** adding 'NOT_'). so I remove the '_' before negation.

1.10 Negation

We need to handle 4 negations, among 'n't, not, never, no', **'n't' is special,** because this should be handle before token. **I wrote a dictionary with words structure like 'doesn't'**, such as he'll –> he will, can't –> can not, shouldn't –> should not, she's –> she is.
I notice that some reviews have **wrong spelling words, like 'does'nt'**, which is absolutely wrong but this kind of words **would affect the result largely**, so I also add these wrong spelling in my dictionary, for example: {"weren't" : "were not", "were'nt" : "were not"},
After replacing the above special words from the text, I use the strategy that **adding 'NOT_'** to every words after negation words and before punctuation. Which would then be like: 'I does not NOT_like NOT_movie'.

1.11 Second punctuation

After negation step, then we can remove the punctuation. Here I also directly remove the symbol like ':)', because I check the No.2628 text of our data, in that case **it has ':)' but the label is 'neg'**, so I can't determine this kind of symbols have a strong relation with the sentiment. Therefore, I just removed them.

1.12 Tokenization

I used **stemming and lemmatisation** in this step to reduce the word dimension, which generate stem_x_data and lemma_x_data separately. The stemming comes from PorterStemmer and the lemmatisation comes from spacy package.

1.13 Numbers

Some words like '2achieve', '007' could be removed, I use RE function to clean those numbers.

2. Naive Bayes and F1 score

Here I would introduce the **baseline** of Naive Bayes classifier, the <unk> NBC, k–smooth NBC, and N–gram NBC would be introduced in the evaluation.
Nowadays, our sentiment models are largely depend on statistics, knowing some incomplete observation of training data with its label, we want to figure out which label our test data belong to, and we could handle this issue with Naive Bayes formula:

$$\underbrace{P(h|D)}_{posterior} = \frac{\overbrace{P(D|h)}^{likelihood}\ \overbrace{P(h)}^{prior}}{\underbrace{P(D)}_{evidence}}$$

Our goal is to maximize the likelihood:

$$h_{MAP} \equiv \underset{h\in H}{\operatorname{argmax}} P(h|D)$$

$$= \underset{h\in H}{\operatorname{argmax}} \frac{P(D|h)P(h)}{P(D)}$$

$$= \underset{h\in H}{\operatorname{argmax}} P(D|h)P(h)$$

P(h) is the **prior distribution**, and **we assume that our training data is big enough to make the P(+) and P(–) be similar to the real word distribution**. Here we could get the prior values through counting the number of positive and negative reviews, divide by the total amount of reviews.

$$P(+) = \frac{Count(positive)}{Count(reviews)}$$

Then we need to figure out P(D|h). We use w to represent word and c to represent label. For unigram, we have

$$P(w_1 w_2 \dots w_n | c) = p(w_1 | c) p(w_2 | c) \dots p(w_n | c)$$

For bigram, we have

$$P(w_1 w_2 \dots w_n | c) = p(w_1 | c) p(w_2 | w_1, c) \dots p(w_n | w_{n-1}, c)$$

Then what we need to do is **counting the word frequency under different labels,** and the total word numbers of each label.

For unigram, under each label we have

$$p(w_i) = \frac{Count(w_i)}{N}$$

For bigram, under each label we have

$$p(w_i | w_{i-1}) = \frac{Count(w_i, w_{i-1})}{Count(w_{i-1})}$$

After adding 1–smooth, we have

Unigrams:

$$P(w) = \frac{freq(w)+1}{N+V}$$

Bigrams:

$$P(w_2 | w_1) = \frac{freq(w_1, w_2)+1}{\sum_w freq(w_1, w)+V}$$

Following these formulas, we could **get the** $P(w_1 w_2 \dots w_n | c)$ **easily**, the specific example is in the Lecture3 pp.14 and Lecture3 pp.47, Dr.Liu has given us a very specific instance. Notice that for **bigrams**, we could **add <cls> at the front of text and <end> at the end of text**, this tricks would make the calculation easier.

$$P(<cls> w_1 w_2 \dots w_n <end> | c) = p(w_1 | <cls>, c) p(w_2 | w_1, c) \dots p(w_n | w_{n-1}, c) p(<end> | w_n, c)$$

For F1 score, it is quite easy, we could **set TP=0, FP=0, FN=0, TN=0**, and we compare the predicted result with the truth label. For the corresponded location, if the golden label is 'pos', and the predict value is 'pos', then the TP would add 1, elif the predicted value is 'neg', then the FN would add 1. If the golden label is 'neg' and the predict value is 'neg', the TN would add 1, elif the predicted value is 'pos', the FP would add 1.

Then the **precision would be TP/(TP+FP), and the recall would be TP/(TP+FN)**.

Then **F1** would be $\frac{2precision\,recall}{precision + recall}$

3. Evaluation and Analysis

3.1 Pipeline

I build a pipeline to simplify the steps.

3.2 Analysis: Spelling mistakes

I use **textblob package** for correcting spelling mistakes, I put it before adding the 'NOT_' and stemming, however, this package depends on the original text, and give a probability of a possible right spelling, this structure **makes the computer quite slow**, so I have to remove this function. However, I still want to mention here that spelling mistakes would possibly be a way to increase final result.

3.3 Evaluation: Stem and Lemmatisation

At first, I plan to use **byte-pair encoding**, however, this methods requires to train the model with vocabularies and its frequency, for our sentiment analysis issue, What is the proper size of corpus for training when we get from original data? If we use a small size corpus, we **might lost comparative words and get very limited increase.** If we use a big size corpus, I could already image **the computer would be very slow**. So, instead of using BPE, I compared the performance of stemming and lemmatisation.

Chart.1

	Stem	Lemma
F1	69.5%	69.9%

According to the Chart.1, we can find that two tokenise methods have similar performance, this is because **the idea of lemmatisation is to make tokens more easier to interpret**, such as 'apples', the stemming would return 'appl', while the lemma would return 'apple', but these two results would have same contribution to the sentiment.

3.4 Handle the unknown words

Methods 1:

Usually, people would set a threshold in the train data, and the word **whose frequency is lower than the threshold would be replaced by the <unk>**, as we can see in Fig.1. The idea of this method is that **if one word only appears once** across the texts, then it would probably is the **proper noun**, or some words that **contribute less to the sentiment**. What's more, when we express our emotions, we tend to use emotion words or emojis, **some of these emotion words and emojis are quite usually to be used**, and this would make these words have **high frequency among the texts**. So, if we replace the low frequency words with <unk>, then our model would becomes **robustness** to the test data, we could just replace the unknown words in the test data with <unk>.

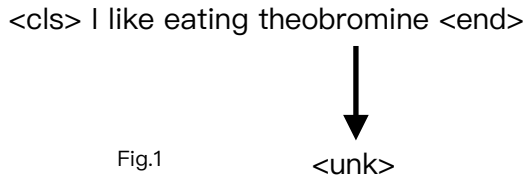

Fig.1

Methods 2:

Methods 2 is the methods I used in the **baseline** of Naive Bayes classifier. If we meet some unknown words in the test data, then we could just ignore it, because we make judgments based on known words, for example, the w_2 is unknown, then we could just calculate the

$P(w_1 w_3 \dots w_n | c)$.

Chart.2 training data is stemming data

	Method 1 : replace unknown words with <unk>	Method 2 : ignore the Unknown words
F1	69.2%	69.5%

As we can see from the Fig.2, two methods have similar performance, which certificate that just **ignoring the unknown words also works in the Naive Bayes Classification** problem.

Methods 3:

Besides the above two methods, I also **tried the TF-IDF method**, my idea is this: **Does methods 1 really works? Because some words with low frequency may also be important to the sentiment**, like word 'preposterous' (absurd), if it only appear once, could we just mark it to <unk>? It is obviously not good.

I want to use **TF-IDF method to remove the less important words across the texts, and replace these words into <unk>**.

However, this calculation method **exceeds my computer power**. Because usually the TF-IDF method is used in the **sentence-level**, for this assignment, after using the TF-IDF, the transformed array size is 25000, and the word size |V| is 122243, for every word I need to check the 25000 rows, and that seems hard.

3.5 Evaluation: k-smooth

As we have been told in the lecture, if we don't use k-smooth, the multiplication factor may be zero, and thus cause problem. I tried this myself using the class NosmoothNBClassifier. **It did return the math domain error**.

I set the k in the range of [0.01,0.1,0.5,1,3,5,7,9,15,20,30,50,100,500,1000,3000,500000] in this k-smooth experiment, and we can find that in Fig.2, after a short increase, the F1 score drops quickly as the k get larger. From the formula we could know that, as the K

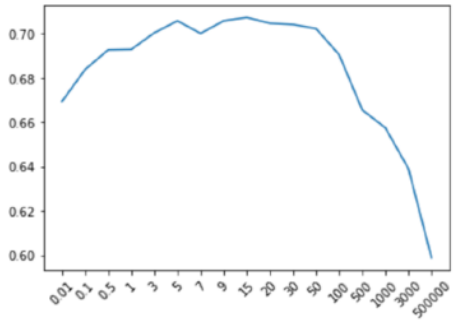

Fig.2

3.6 Evaluation: threshold of <unk>

The threshold is to determine whether a word should be marked as unknown word. I set the threshold in the range of [1,2,3,5,10,20,50,100,500,1000,3000,5000], while the k-smooth set to 1. According to the Fig.3 we can find that when the threshold is 500, the F1 score get to the peak, and when threshold get to 1000, the line decrease sharply.

I print the vocabulary set, and I found that **when threshold=500, the model remains some important words, like 'good', 'best', 'not_bad', 'bad', when threshold=1000 those key word still exists.** But **after 1000**, all of these words are set to **<unk>,** which cause the F1 score decrease.

Imaging that every train text is nothing but <unk>, and every test text is also <unk>, then the distribution would follow the prior distribution which is 0.5, that explain why the curve decrease.

Besides, we can see that through setting threshold, the model **could reduce the effect comes from noise words**, and there are still lots of **useless words** that contribute less to the final result even after we clean data.

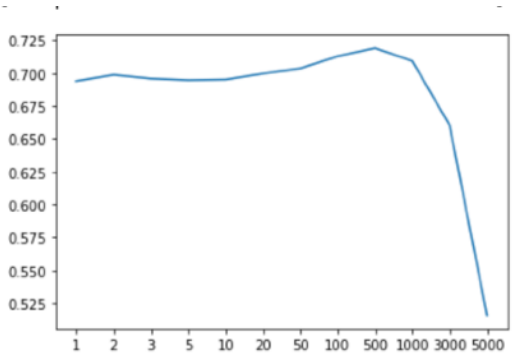

Fig.3

3.7 Evaluation: word frequency distribution & deep analysis of k-smooth and threshold

For a better view of the data, I print the **distribution of word frequency**. Combining this bar figure with the above k-smooth curve, and the threshold curve, we can conclude that:

- There are plenty of **low frequency words**, among those words, **some are useful(because k-smooth curve arise when increasing those words' weight), but most of then are useless.**
- K-smooth is **not that the bigger the better**. There are also some **high frequency words which plays role in the classification,** such as 'good', 'best', 'not_bad', 'bad' . Adding k would make those vital high-frequency words less important.
- We need to find a **balance** between threshold and k-smooth.

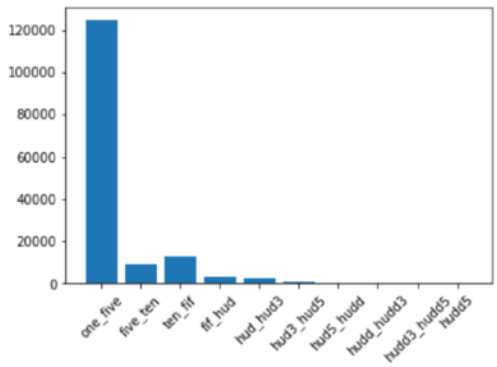

Fig.4

3.8 optimize

According to the above analysis, I **narrow the values** of the two lists and want to find a batter solution.

ksmooth_list = [1,3,5,9,15,30,50]

thres_list = [50,100,500,1000]

And the best result is F1_score = 73.2%, while k-smooth = 50, thres_list = 50.

3.9 Evaluation: Bigram Naive Bayes Classifier

For bigram NBC, we need to count $p(w_i|w_{i-1}) = \frac{Count(w_i, w_{i-1})}{Count(w_{i-1})}$ in positive and negative class.

For example, given 'I like this movie', we need to know $p(w_{this}|w_{like}, +)$ and $p(w_{this}|w_{like}, -)$

Chart.3

	Unigram, ignore unknown words	Bigram, ignore unknown words
F1	69.5%	76.6%

According to the Chart.3 result we can know that Bigram does improve the performance, this is partly because **bigram catches more information from the context, which helps increase the result.**

However, **the number of gram is also a trade-off issue**. Assuming that we need to know all of the history information where N=len(text), then the robustness would be bad, we need to have the test text looks very similar to the train text. That is almost impossible.