ERG2050 Final Project: Binary Sentiment Analysis

Passionate ERG Lovers

Hailu, CHEN 119010013

Hanfei, ZHU 119020585

Jingyuan, YANG 119010380

Mengjie, CHEN 11901020

Qingxuan, CHEN 119010024

Shuyu, HUANG 119010111

Tianyuan, XIE 119010352

Yiling, KUANG 119010139

Yixun, CHEN 119010038

Yongjia, HENG 119010100

The Chinese University of Hong Kong, Shenzhen

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

Content

I. Motivation

This project requires us to use provided 25k train sets, which are split into positive review and negative review, to determine whether each comment in test set is positive or negative comments.

The problem can be composed of 3 subproblems: selecting appropriate feature functions to represent input x, building effective and efficient models (classifier), and using the classifies to predict positive or negative sentiment. By analyzing these subproblems, our group conclude several properties that the feature functions, models should have in advance:

Feature function:

It should be able to convert each comment (original text) into a feature vector.

It should be able to handle unseen words, misspelled words in the test data.

It cannot use too many features so that NumPy cannot handle.

The algorithm for deriving the feature function does not require GPU or TPU.

Model:

It should be able to output binary value, which represent positive and negative sentiment, or it should be able to classify data into 2 categories.

It should be able to handle numerous data, since some comments are long.

Based on these motivations, we select 4 feature functions, which are TF-IDF, word2vector, glove, and Bert. We also select 8 models, which are dictionary-based sentiment analysis, logistic regression, kNN, SVM, NaiveBayes, LSTM, biLSTM, and Neural Network. For feature functions, all 4 features can convert each comment (original text) into a feature vector. Moreover, word2vector, glove, and Bert are designed for natural language processing.

These 3 feature functions can convert each word into a vector so that a whole comment is represented by a matrix with each row as a word vector.

For models, they are all able to make binary decisions. Even when part of the output value is continuous value, it is possible to use sigmoid function to convert it into a number between 0 and 1. They can handle large dataset to compute predictions. The innovation in our group is that we try different combinations between all feature functions and models. First, we derive the feature vector for each comment, then we apply it to model that can handle these feature vector in order to derive a training model.

II. Preprocessing: get the vector for each comment

Model overview:

4. **TF-IDF, hash**
5. **Word2vec, glove**
6. **Bert**

Each part:

1. Principle

2. Code (2.1 package 2.2 process 2.3 parameter fitting)

Tf-idf

1. Principle

TF-IDF is a measure that represent the words into vectors. It is done simply by multiplying term frequency and the inverse documents frequency. It works by increasing proportionally to the number of times a word appears in a document but is offset by the number of documents that contain the word. Because if certain words appear in too many documents, then it becomes less important.

tf(*t*,*d*) is term frequency, i.e. the count of a term in a document.

$$tf(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

idf(t,D) is inverse document frequency, obtained by the following formula

$$idf(t,D) = log \frac{N}{|\{d \in D: t \in d\}| + 1}$$

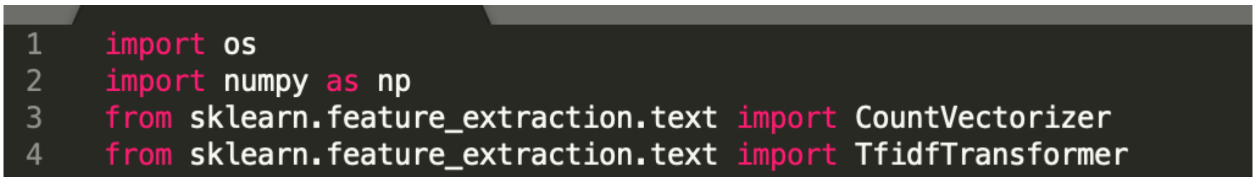
where

N is the total number of documents in the corpus. and the denominator is the number of documents where the word appears. To avoid the case that the denominator goes to 0, we simply plus 1.

2. Code

2.1 Package

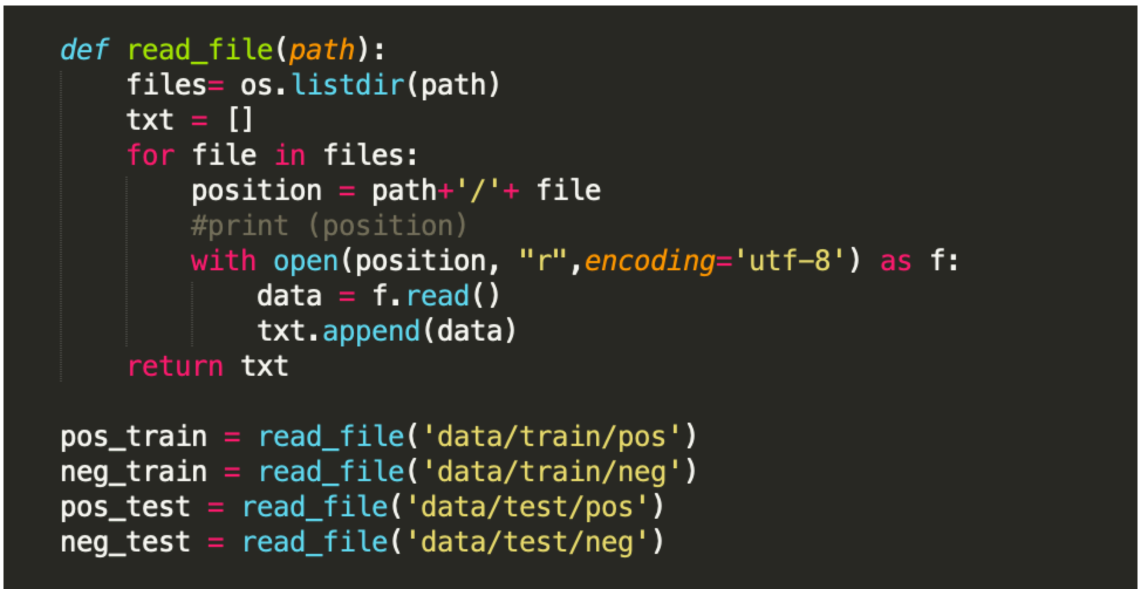
CountVectorizer" and "TfidfTransformer" from sklearn.fearure_extraction.text, this is two modules from sklearn to extract word vectors



```python
import os
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
```

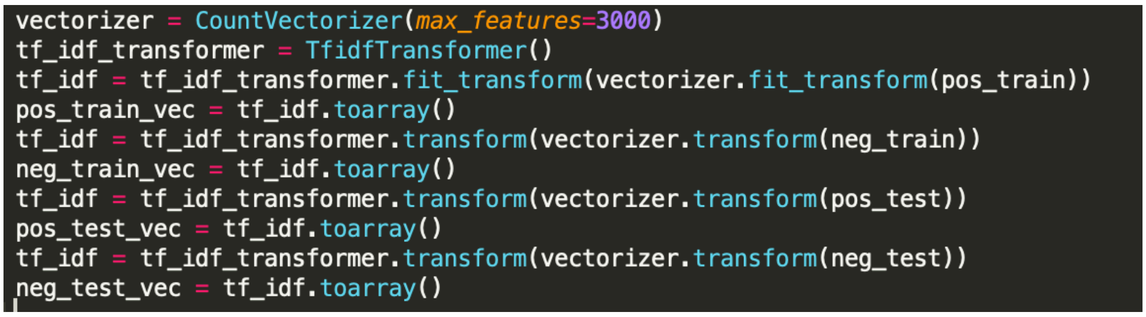
2.2 Process

2.2.1 Read data



```python
def read_file(path):
    files= os.listdir(path)
    txt = []
    for file in files:
        position = path+'/'+ file
        #print (position)
        with open(position, "r",encoding='utf-8') as f:
            data = f.read()
            txt.append(data)
    return txt

pos_train = read_file('data/train/pos')
neg_train = read_file('data/train/neg')
pos_test = read_file('data/test/pos')
neg_test = read_file('data/test/neg')
```

2.2.2 **use the package to generate tfidf vector**



```python
vectorizer = CountVectorizer(max_features=3000)
tf_idf_transformer = TfidfTransformer()
tf_idf = tf_idf_transformer.fit_transform(vectorizer.fit_transform(pos_train))
pos_train_vec = tf_idf.toarray()
tf_idf = tf_idf_transformer.transform(vectorizer.transform(neg_train))
neg_train_vec = tf_idf.toarray()
tf_idf = tf_idf_transformer.transform(vectorizer.transform(pos_test))
pos_test_vec = tf_idf.toarray()
tf_idf = tf_idf_transformer.transform(vectorizer.transform(neg_test))
neg_test_vec = tf_idf.toarray()
```

2.3 parameter

The parameter "max_features" of CountVectorize can be changed. Based on the result of combining tf-idf and naïve bayes, we selected the best number 3000 after trial and error. The process of selecting parameters would be represented on the last part of naïve bayes model.

Hash

1. Principle:

It belongs to **bag of words** models. The bag-of-words model assumes that we do not consider the contextual relationship between words in the text, but only consider the weights of all words. The weight is related to the frequency of words in the text.

It is similar to TF-IDF model, but it performs better when dealing with large number of words after tokenization. It uses hash trick to lower the dimension.

In Hash Trick, we will define the size of the hash table corresponding to a feature hash. The dimension of this hash table will be much smaller than the feature dimension of our vocabulary, so it can be regarded as dimensionality reduction. The specific method is to correspond to any feature name, we will use the Hash function to find the location of the corresponding hash table, and then accumulate the statistical value of the word frequency corresponding to the feature name to the location of the hash table. If expressed in mathematical language, if the hash function h makes the i-th feature hash to position j, that is, h(i)=j, then the word frequency value $\phi(i)$ of the i-th original feature will be added to the hashed on the word frequency value ϕ¯ of the j-th feature, that is:

$$\bar{\phi}(j) = \sum_{i \in \mathcal{J}; h(i)=j} \phi(i)$$

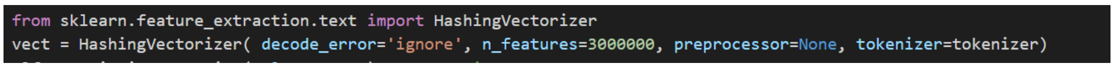
Here since there were 50000 comments, we tried hashing vectorizer to generate the vectors for a file.

2. Code:

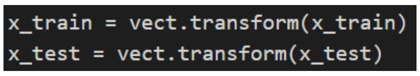
2.1 Package:

pandas: use data frame to store the data
nltk: from Nltk.corpus import stopwords
re: use re.sub and re.findall in tokenizer function to help tokenization
sklearn.feature_extraction.text: import HashingVectorizer

2.2 Process:



```
from sklearn.feature_extraction.text import HashingVectorizer
vect = HashingVectorizer( decode_error='ignore', n_features=3000000, preprocessor=None, tokenizer=tokenizer)
```



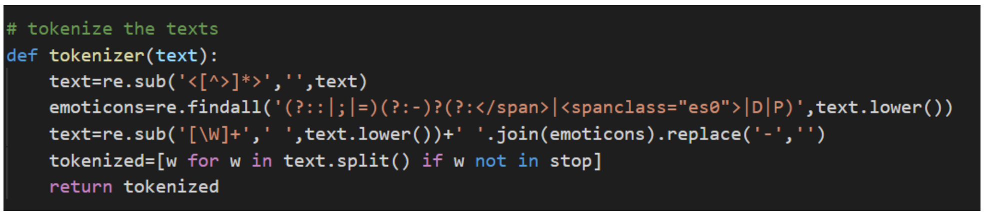
```
x_train = vect.transform(x_train)
x_test = vect.transform(x_test)
```

1. We read the data from the files and restore them in a csv file.
2. We read the data from csv file and separate them into texts and labels.
3. We import the package from sklearn and initiate the hashingvectorizer object.
4. We use the package to generate vectors for our texts.

2.3 Parameter:

n_features: it can be left at its default value (approximately one million), we set a number (3000000). We chose this number because it can give us relatively high accuracy and the run time is not too long.

Tokenizer: remove stopwords, punctuation……



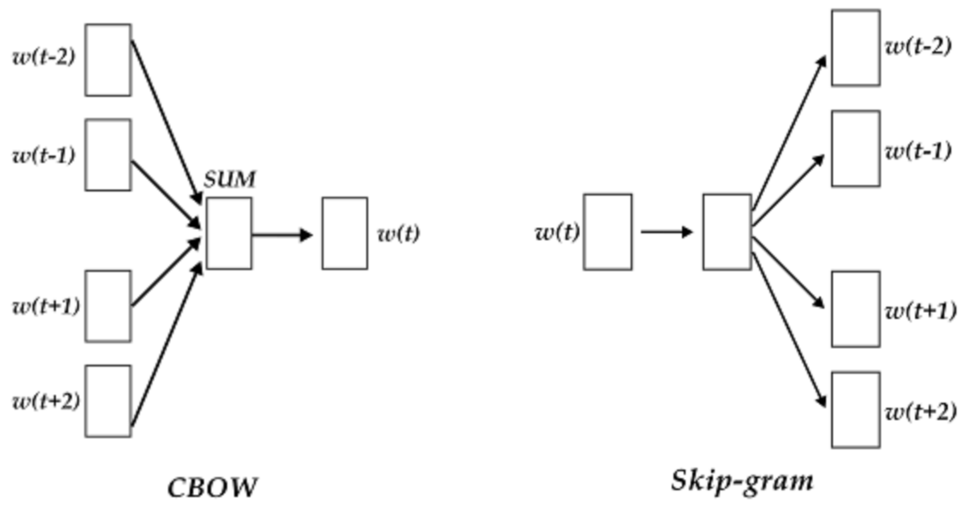
```
# tokenize the texts
def tokenizer(text):
    text=re.sub('<[^>]*>','',text)
    emoticons=re.findall('(?::|;|=)(?:-)?(?:</span>|<spanclass="es0">|D|P)',text.lower())
    text=re.sub('[\W]+',' ',text.lower())+' '.join(emoticons).replace('-','')
    tokenized=[w for w in text.split() if w not in stop]
    return tokenized
```

Word to Vector (Word2Vec)

1. Principles

Word2Vec is a method that represents a word by a vector that allows us to measure the relations between words. Traditionally, One-Hot encoding is used to encode the features of a word. The method has some shortcoming, for example, the feature matrix is sparse, and it assumes the independence between words. Word2Vec solves the problem using a neural network.

There are two ways to accomplish the word to vector transformation. The first is Continuous Bag-Of-Words Model (CBOW) which inputs the 2t words nearby and output the target word.



We want to learn the transformation matrices to maximize the log likelihood function of ω that $\sum log\ p(\omega|Context(\omega))$. The method is preferrable when training texts with smaller scales.

The second way is skip-gram method, which is also used in the Word2Vec function in the Gensim module of Python. The skip-gram method requires an input word, and then it predicts the neighboring words. First, we need to define the scale of window W, which means there will be 2W words – W words before the input word and W words after it – in the window. Based on these vectors, the neural network will output a probability distribution that a word in the dictionary is the output word. Then, we should define the dimension (i.e., number of features) of the word vectors. Combining the vectors and back propagation, through gradient descent, the neural network gives us the embedded matrix. These vectors of words allow us to calculate the similarities and predict the contexts.

2. Code

2.1 Packages

Re: This module provides regular expression matching operations. With the embedded integration Re module, we can call directly to implement regular matching. In our program, re is mainly used in the tokenization of the sentences.

Genism: Contains functions to transform documents (strings) into vectors and calculate similarities between documents.

Genism.models: Contains algorithms for extracting document representations from their raw bag-of-word counts.

From sklearn import svm: The module sklearn contains many machine-learning algorithms, including SVM.

2.2 Process

To get a matrix of word vectors, we first need to wipe the stop words and punctuations in the texts. We got a list of stop words in Python in-built library, and then tokenize the sentences after removing the stop words and punctuations.

```
# Collect stopwords
stop=[]
```

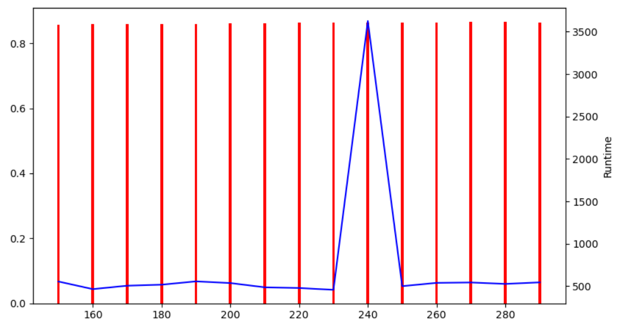
```
with open('stopwords.txt','r') as f:
    for line in f:
        stop.append(line)

# Transfer word to word vector
def tokenizer(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text.lower())
    text = re.sub('[\W]+', ' ', text.lower()) +\
        ' '.join(emoticons).replace('-', '')
    tokenized = [w for w in text.split() if w not in stop]
    return tokenized
```

We defined a class csvStream which creates an iterable object of the texts. Using the object, we can easily apply the function of Word2Vec. We stored the matrix of word vector in a file named "outpath".

```
outpath = 'w2v_trainResult'
train_model = gensim.models.Word2Vec(lineIterator, vector_size=vec_size, window=5, min_
count=mincount, workers=4, epochs=10) #
train_model.save(outpath)
```

2.3 Selecting Parameters

Referring to the general solution, we determined the size of window to be 5. To decide the dimension of word vector, we need to balance between the accuracy and running time. We picked the multiplication of 10, from 150 to 300. We found little difference between the runtimes and accuracies under each parameter. Therefore, we chose 250 as the vector size.



Glove

1. Principle:

Co-occurrence matrix: co-occurrence probabilities matrix can combine the overall statistics of corpus and the local contextual feature (sliding window).

Xij = the number of Wordj appears in the context of wordi in Corpus

Xi=∑kXik=the total number of words which appear in the context of wordi
Pij=P(j/i)=Xij/Xi=the probability of wordj appearing in the context of wordi
Ratio=Pik/Pjk. Ration can reflect the relevance of words as follow:

	Wordj, wordk correlate	Wordj, wordk are incorrelate
Wordi, wordk correlate	Ration≈1	Ratio is large
Wordi, wordk is uncorrelate	Ration is small	Ration≈1

In Glove model, we assume that the word vectors of wordi, wordj, wordk are wi, wj and wk. And the basic of the model is F(wi, wj, wk)=Pik/Pjk. The equality means the function of word vector contains the information of co-occurrence. So the aim of Glove model is to find F.

2. Code

2.1 Packages:

Os: os.listdir(path): return the list of the files or folders contained in the specific folder
os.path.join(path ,file): combine all files in the path

Corpora: corpora.Dictionary (document, prune_at): document should be a document matrix; prune_at is used to control the total dimension of matrix.

Pyprind: pyprind.ProgBar: return a visible progress bar

Punctuation: contain all punctuation

Numpy: np.zeros(shape, dtype=float, order='C'): Returns a new array filled with 0 given the shape and type.

GloVe: GloVe(n, max_iter, learning_rate): n: the length of matrix, max_iter: maximum number of iterations, learning_rate: the rate of learning
GloVe.fit(matrix): return the glove matrix

Nltk: Nltk is a natural semantic processing library in Python

Stopwords: stopwords.words('english'): return stopwords in English

Word_tokenize (sentence): return the tokenized copy of text

Pos_tag: pos_tag(tokens): tag the given list of tokens

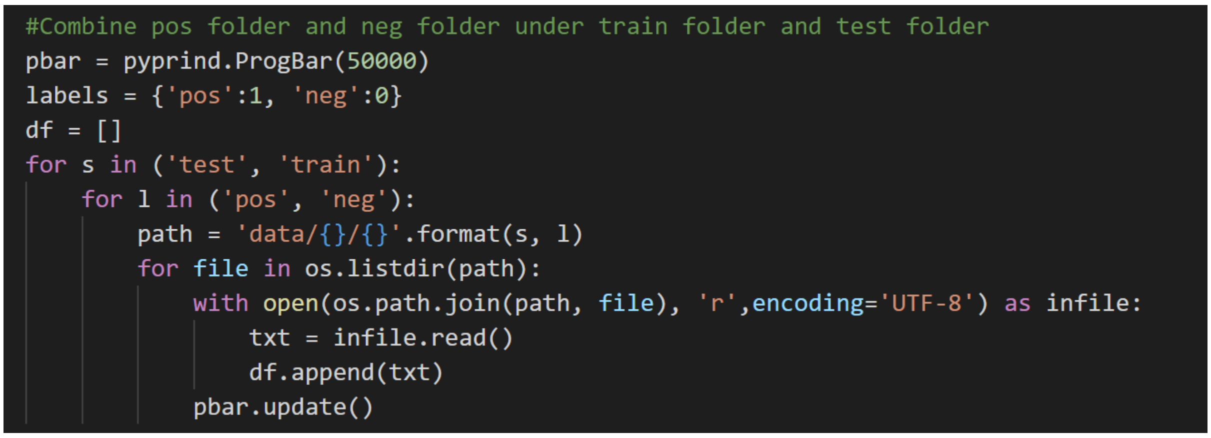
WordNetLemmatizer: WordNetLemmatizer.lemmatize(word, pos): return the lemma of the input according to the pos.

Re: re.sub(pattern, repl, string, count=0, flags=0): substitute regulation expression string

Genism: is a simple and efficient Python library for natural language processing. The input of genism is original and unstructured digital text. The semantic structure of the document is automatically found by calculating the statistical co-occurrence patterns in the training corpus.
gensim.models.KeyedVectors.load_word2vec_format(text): save text as the form of word2vec

2.2 Process

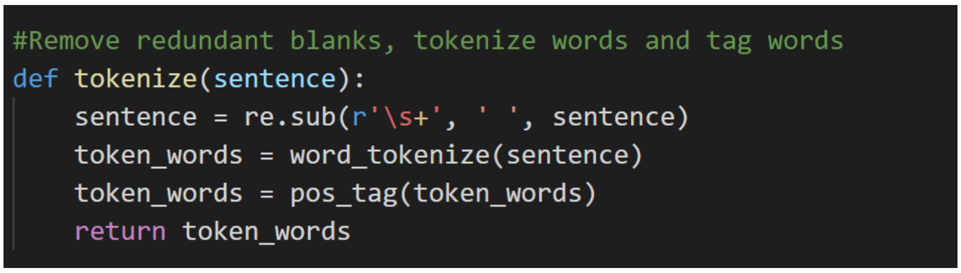
(1)Text collection: combine all text under pos folders and neg folders together.



```python
#Combine pos folder and neg folder under train folder and test folder
pbar = pyprind.ProgBar(50000)
labels = {'pos':1, 'neg':0}
df = []
for s in ('test', 'train'):
    for l in ('pos', 'neg'):
        path = 'data/{}/{}'.format(s, l)
        for file in os.listdir(path):
            with open(os.path.join(path, file), 'r',encoding='UTF-8') as infile:
                txt = infile.read()
                df.append(txt)
            pbar.update()
```

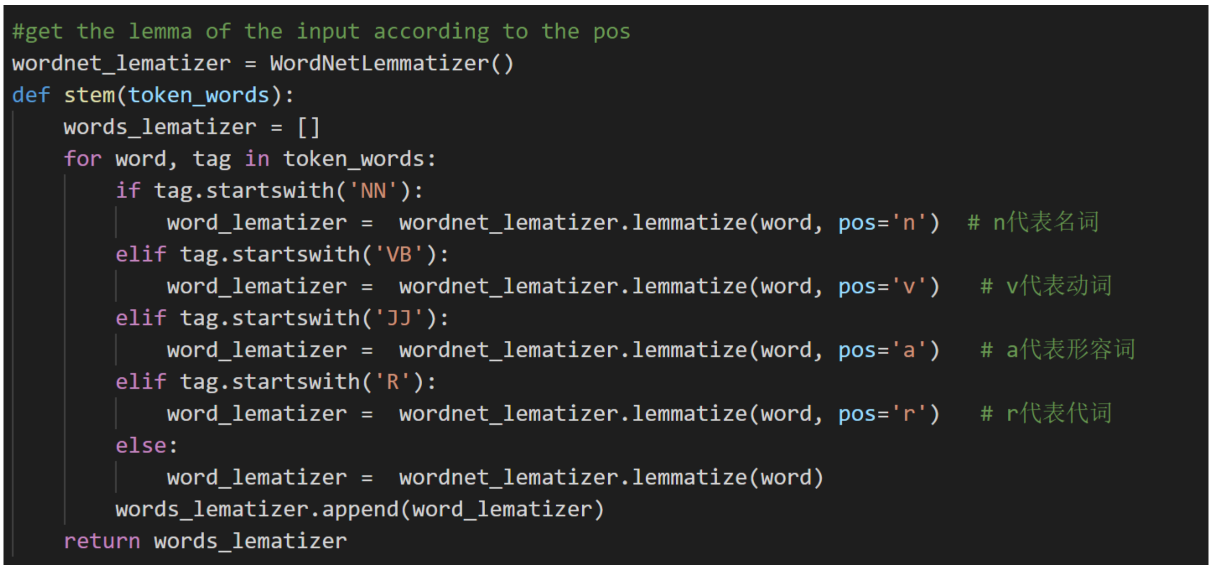
(2) Preprocess data:

(a) Remove redundant blanks, tokenize words and tag words



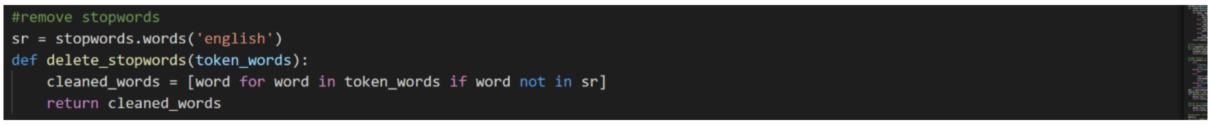
```python
#Remove redundant blanks, tokenize words and tag words
def tokenize(sentence):
    sentence = re.sub(r'\s+', ' ', sentence)
    token_words = word_tokenize(sentence)
    token_words = pos_tag(token_words)
    return token_words
```

(b) Get the lemma of the input according to the tagged pos.

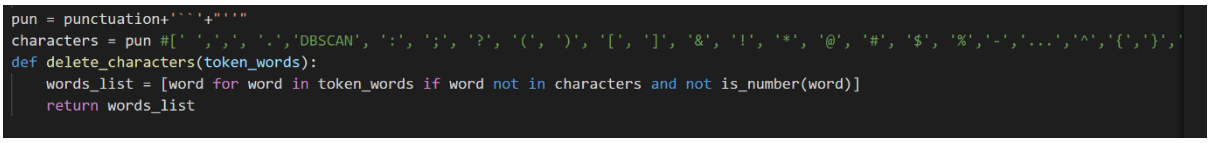


```python
#get the lemma of the input according to the pos
wordnet_lematizer = WordNetLemmatizer()
def stem(token_words):
    words_lematizer = []
    for word, tag in token_words:
        if tag.startswith('NN'):
            word_lematizer =  wordnet_lematizer.lemmatize(word, pos='n')  # n代表名词
        elif tag.startswith('VB'):
            word_lematizer =  wordnet_lematizer.lemmatize(word, pos='v')   # v代表动词
        elif tag.startswith('JJ'):
            word_lematizer =  wordnet_lematizer.lemmatize(word, pos='a')   # a代表形容词
        elif tag.startswith('R'):
            word_lematizer =  wordnet_lematizer.lemmatize(word, pos='r')   # r代表代词
        else:
            word_lematizer =  wordnet_lematizer.lemmatize(word)
        words_lematizer.append(word_lematizer)
    return words_lematizer
```

(c) Remove stopwords (most common functional words), digits and punctuation.

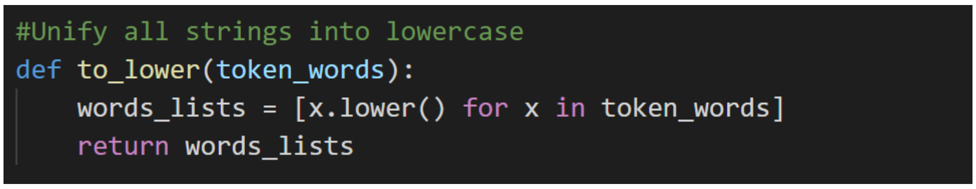


```python
#remove stopwords
sr = stopwords.words('english')
def delete_stopwords(token_words):
    cleaned_words = [word for word in token_words if word not in sr]
    return cleaned_words
```



```python
pun = punctuation+'```'+"'''"
characters = pun #[' ',',', '.','DBSCAN', ':', ';', '?', '(', ')', '[', ']', '&', '!', '*', '@', '#', '$', '%','-','...','^','{','}','
def delete_characters(token_words):
    words_list = [word for word in token_words if word not in characters and not is_number(word)]
    return words_list
```

(d) Make all words lowercase.

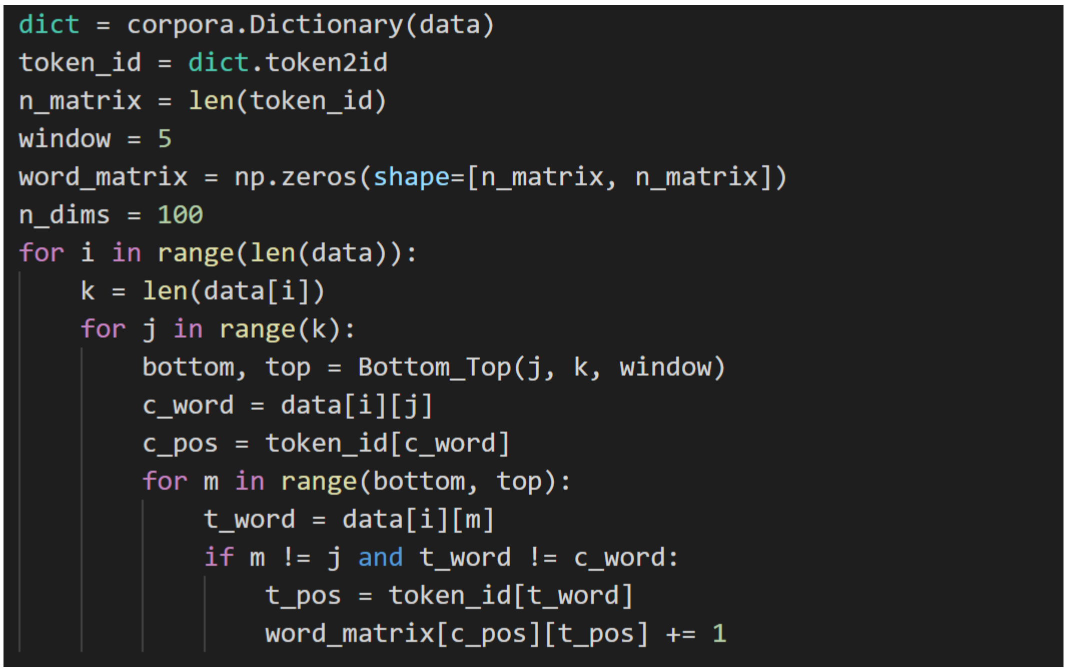


```
#Unify all strings into lowercase
def to_lower(token_words):
    words_lists = [x.lower() for x in token_words]
    return words_lists
```

(3) Glove models:

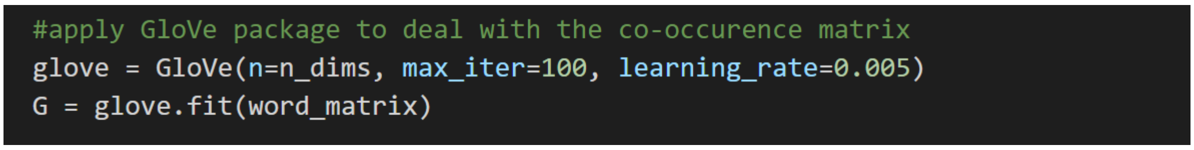
(a) Form co-occurrence matrix: co-occurrence probabilities matrix can combine the overall statistics of corpus and the local contextual feature (sliding window).

Xi,j = the number of $word_i$ and $word_j$ appear in the same window (the length of window is set). Co-occurrence matrix is obtained by traversing the whole corpus.



```
dict = corpora.Dictionary(data)
token_id = dict.token2id
n_matrix = len(token_id)
window = 5
word_matrix = np.zeros(shape=[n_matrix, n_matrix])
n_dims = 100
for i in range(len(data)):
    k = len(data[i])
    for j in range(k):
        bottom, top = Bottom_Top(j, k, window)
        c_word = data[i][j]
        c_pos = token_id[c_word]
        for m in range(bottom, top):
            t_word = data[i][m]
            if m != j and t_word != c_word:
                t_pos = token_id[t_word]
                word_matrix[c_pos][t_pos] += 1
```

(b) Use GloVe package to input co-occurrence matrix and fit each word's word vector.

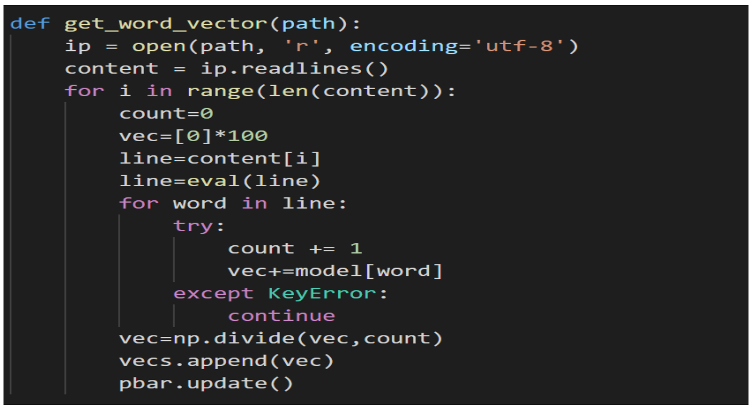


```
#apply GloVe package to deal with the co-occurence matrix
glove = GloVe(n=n_dims, max_iter=100, learning_rate=0.005)
G = glove.fit(word_matrix)
```

(c) Save GloVe word vector into text.

(4) Transform texts into feature vectors

Add up each word vector and use the average to represent the text's feature vector.



```
def get_word_vector(path):
    ip = open(path, 'r', encoding='utf-8')
    content = ip.readlines()
    for i in range(len(content)):
        count=0
        vec=[0]*100
        line=content[i]
        line=eval(line)
        for word in line:
            try:
                count += 1
                vec+=model[word]
            except KeyError:
                continue
        vec=np.divide(vec,count)
        vecs.append(vec)
        pbar.update()
```

- **Problems & Solutions:**

In the process of applying GloVe package, there should form a 250934*250934 matrix and in this case, the window will automatically stop the process. So we try to solve the problem by cut down words. And to guarantee the accuracy, we choose to keep adjectives and adverbs.

- **Improvement (cut down words):**

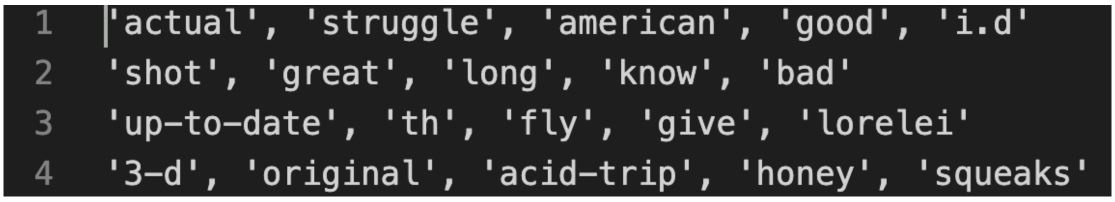
Data preprocessing:
Glove is a feature function which convert each word into a vector with dimension 1*100, which is store as a NumPy array vector. By using the glove vector as word representation, each comment can be represented by a matrix with 100 columns and word length as its row number. However, NumPy is unable to derive a matrix whose dimension is over 150000 and, training efficiency in this process is not guaranteed. Therefore, it is necessary to choose and select more important words that appear in the comments.
Usually, it is natural for people to use adjective to express their sentiment. **Our group select 5 adjectives in each comment in order to reduce running time and let the training process** become more efficient. Since the stop-words, punctuations and symbols have been removed by previous data cleaning (It is an object orientated programming.) This process includes tokenization and POS tagging.

- Sentence tokenization: Our group uses nltk python packages to tokenize the comments. Then our group uses a list in order to store all the words in one comment. In our code, the command is

  ```
  text_list = nltk.word_tokenize(text)
  ```
- POS tagging: POS tagging is also called part-of speech-tagging. It gives back the part of speech of each word in individual sentence.
  For example, text = nltk.word_tokenize("And now for something completely different")
  nltk.pos_tag(text)
  ```
   [('And', 'CC'), ('now', 'RB'), ('for', 'IN'), ('something', 'NN'),
   ('completely', 'RB'), ('different', 'JJ')]
  ```
  After tagging each comment, we select words with "JJ" because it represents adjective. Then we the outcome of POS tagging into NumPy array and select only the first column, which includes all the adjective word. For previous example, the word that will be selected is 'different'. In the end, we store the first 5 adjective in a list and write it into a txt file for further use. The reason why we use a txt file to store is it takes some time to

run this program and inherit a txt file will be much more efficient.

The final file we write is like:



```
'actual', 'struggle', 'american', 'good', 'i.d'
'shot', 'great', 'long', 'know', 'bad'
'up-to-date', 'th', 'fly', 'give', 'lorelei'
'3-d', 'original', 'acid-trip', 'honey', 'squeaks'
```

BERT

1. Principle

Masked Language Model $BERT_{BASE}$: L=12, H=768, A=12. (length of each vector is 768)

In BERT, 15%WordPiece Token will be randomly masked. When training the model, each sentence will be learnt multiple times. Tokens are not always masked. There are probabilities.

80%: my dog is hairy -> my dog is [mask]

10%: my dog is hairy -> my dog is apple

10%: my dog is hairy -> my dog is hairy

We split each word in the sentence, and create a token dictionary, find the corresponding number in the exist tokenizer dictionary.

2. Code

2.1 Package:

BertEmbedding

class BertEmbedding(ctx=mx.cpu(), dtype='float32', model='bert_12_768_12', dataset_name='book_corpus_wiki_en_uncased', params_path=None, max_seq_length=25, batch_size=256)

Encoding from BERT model.

Parameters

(ctx : Context. running BertEmbedding on which gpu device id. Dtype: str data type to use for the model. Model: str, default bert_12_768_12. pre-trained BERT model dataset_name: str, default book_corpus_wiki_en_uncased. pre-trained model dataset params_path: str, default None path to a parameters file to load instead of the pretrained model. Max_seq_length : int, defaultmax length of each sequence batch_size : int, default 256 batch size

os

OS routines for NT or Posix depending on what system we're on.

Programs that import and use 'os' stand a better chance of being portable between different platforms. Of course, they must then only use functions that are defined by all platforms (e.g., unlink and opendir), and leave all pathname manipulation to os.path (e.g., split and join).

2.2 Process

• For each sentence in the train and test file, we use BertEmbedding to get the word vector.

• Similar to SVM, we adjust the parameters to fit the situation of sentences in the train file.

• Finally we find a line that could divide those views in the test file into two classes.

III. The model of classification

Model overview:

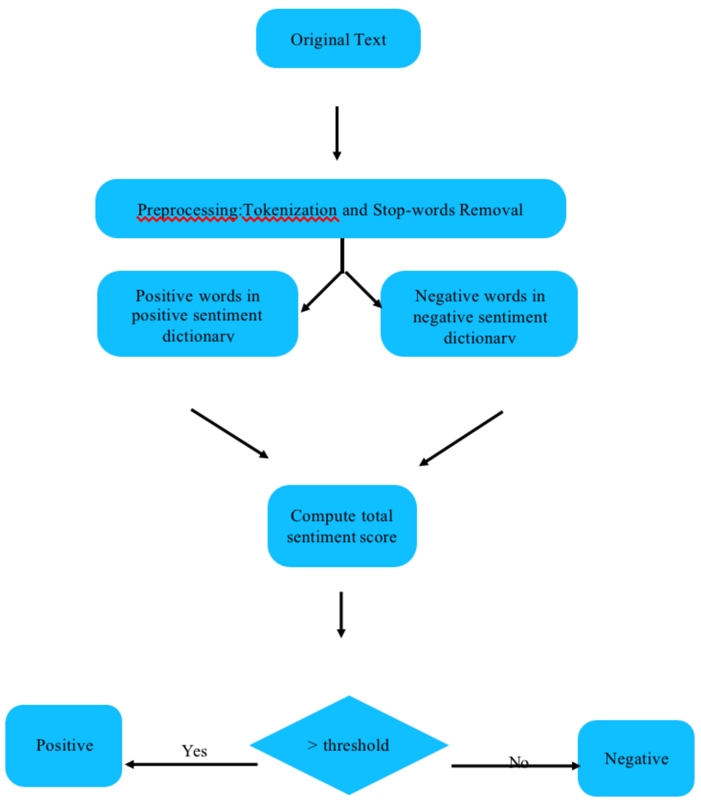
1. **Dictionary based word-Level Sentiment analysis**
2. **KNN, Logistic**
3. **SVM, Naive Bayes**
4. **Neural Network (NN, LSTM, biLSTM)**

Dictionary based word-Level Sentiment analysis

1. Principle

In the word-based approach the criterion for selection a tweet to automatic classification is the presence of words that express sentiment such as good, bad, excellent or terrible. From these words it is possible to infer the sentiment present in the text. These words are used to determine the sentiment (positive and negative) according to the application. This is the simplest form of sentiment analysis and it is assumed that the word contains an opinion on one main object expressed by the author of the document. In this approach people assign scores directly to words. We call these *prior polarities*, i.e., polarities of the words independent of their context (and thus their meaning).

Algorithm:


Original Text
Preprocessing:Tokenization and Stop-words Removal
Positive words in positive sentiment dictionary
Negative words in negative sentiment dictionary
Compute total sentiment score
> threshold
Yes
No
Positive
Negative

2. Code

2.1 package

Os, NLTK, re, collections

re: Python package provides regular expression matching operations. Both patterns and strings to be searched can be Unicode strings (str) as well as 8-bit strings (bytes). Unicode strings and 8-bit strings cannot be mixed, in other words, it is impossible to match a Unicode string with a byte pattern or vice-versa; similarly, when asking for a substitution, the replacement string must be of the same type as both the pattern and the search string.

collections: Collections is a specialized container datatypes providing alternatives to Python's general purpose built-in containers, dict, list, set, and tuple. It is a useful data frame to store the occurrence of certain strings.

nltk: nltk is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries, and an active discussion forum.Emotion dictionary for sentiment analysis.

2.2 Process

2.2.1 Preprocessing

This process includes sentence tokenization and stop words removal. Python NLTK packages provide built in method for users to perform sentence tokenization, stop words removal, and punctuation removal.

1) For sentence tokenization, nltk.word_tokenize(sentence) can split a string into different pieces and include them in a list. For example, the output of nltk.word_tokenize(At eight o'clock on Thursday morning, Arthur didn't feel very good.) is ['At', 'eight', "o'clock", 'on', 'Thursday', 'morning','Arthur', 'did', "n't", 'feel', 'very', 'good', '.']. Using this method, the comments about movies can be tokenized.

2) For stop words removal, we use built in stop-words dictionary. First we use set() function to create a set of all the words in each comment and create another set of stop-words of the built-in stop-words dictionary. Then, it is desire to compare these 2 sets in order to remove all the overlapping stop words in the comments.

3) For punctuation removal, we create a list which includes all the possible punctuation in English, including all the possible symbols that are irrelevant to sentiment analysis. For instance, $, %, @, etc. For python, the operation 'not in' require the computer to select these strings that are not in the punctuation list. In this way, we will be able to remove all the punctuations and irrelevant symbols out of the comments.

2.2.2 Training process

The classifier is the 2 dictionaries of portage sentiment words and negative sentiments words. However, the sentiment dictionary is downloaded from the Internet. Previous studies have already collected frequently used polar sentiment words and select all the synonym of these word in the English words. In all, there's are 4783 negative sentiment words and 2006 positive sentiment words. Previous studies have collected the popularized version of English that people used on social media, such as "lol", "hhhhh" and other kinds of words. The only step in the training process is to load those 2 sentiment dictionaries.

2.2.3 Testing process

The classifier from the training process and the preprocessed test data are recognized as inputs in the testing process. Each appearance in the positive sentiment dictionary is assigned

1 score. Similarly, each appearance in the negative sentiment dictionary is assigned -1 score. After looping through all the words in each comment, we calculate the overall sentiment score of each comment. If the overall sentiment score is above 0, then we classify this comment as positive, otherwise, we classify it as negative. If the overall sentiment score is exactly 0, which means that no sentiment words appear in this comment, we randomly select a label for this comment.

2.3 Result & Analysis

The accuracy of the word-level sentiment analysis is not satisfying the overall accuracy of all the comments in the test data is 52%. We analyze several for this unpleasant result:

The word-level sentiment analysis regards each word individually, rather than looking at the context of each word. This implicit assumption, in fact violates the natural property of language using.

Moreover, there are several emphasizing words, transition word that convey several messages about the sentiment of the people online. For example, the word 'however' clearly indicates that the sentences coming after will be more important. For instance, I was happy and willing to see this movie at the very beginning, however, I hate the characters in the movie. The word level sentiment analysis cannot classify this comment with robust prediction.

The dictionary may not be enough for computer to analyze all the comments. Several words may work together to express polar sentiment. Individual word sentiment analysis may not be able to handle cases like these.

2. KNN, Logistic

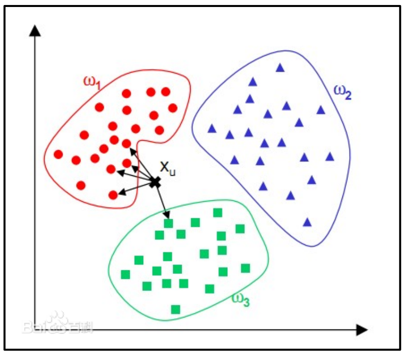
KNN

1. Principle

1.1 The Algorithm

For the new test instance Xu, the following steps should be taken to determine its label.

• Determine parameter K
• Calculate the distance between the test instance Xu and all the training instances
• Sort the distances and determine K nearest neighbors
• Gather the labels of the K nearest neighbors (e.g., 4*w1,1*w3)
• Use simple majority voting or weighted voting



1.2 Properties

• A "lazy" classifier (cluster).

• No learning in the training stage
• Feature selection and distance measure are crucial

2. Code
2.1 Packages:
module gensim
This package contains functionality to transform documents (strings) into vectors and calculate similarities between documents.
module sklearn.neighbors
sklearn.neighbors module implements the k-nearest neighbors algorithm.
2.2 process
• Split the data into training data, test data (by the given file: train&test)
• Record instances as feature vectors
• For each data d in the test file
• Find k training instances (in the train file) that are closest to d

Euclidean distance

$$dist(d_i, d_j) = \sqrt{\sum_k (a_{i,k} - a_{j,k})^2}$$

2.3 Model
KNeighborsClassifier (n_neighbors=5, weights=’uniform’, algorithm=’auto’, leaf_size=30, p=2, metric=’minkowski’, metric_params=None, n_jobs=1, **kwargs)
Parameters:
n_neighbors: because we want to classify the data into two classes (positive and negative), we choose n_neighbor=2.
Weights:
‘uniform’: uniform weights. All points in each neighborhood are weighted equally.
Accuracy = 55.0266%, time=135.08s
‘distance’: weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
Accuracy = 68.03%, time=297.24s
So we choose weights= “distance”

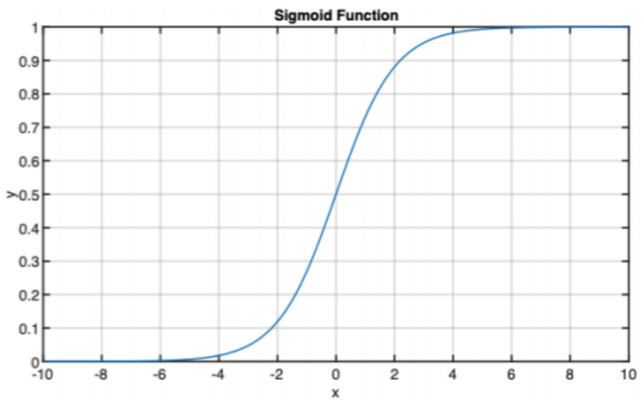
Logistic

1. Principle
Task: Given an x (continuous), decide its label y (binary)
Data:
x: the thing to be labeled/regressed (a vector)
y: the continuous value in the real axis (class: 0 or 1)
Regression process:

Logistic Regression

$$p(\boldsymbol{y} = \mathbf{1}|\boldsymbol{x}) = \sigma(\boldsymbol{w}^T\boldsymbol{x})$$
$$p(\boldsymbol{y} = \mathbf{0}|\boldsymbol{x}) = 1 - \sigma(\boldsymbol{w}^T\boldsymbol{x})$$

- σ: sigmoid function (aka. logistic sigmoid function)
- σ: $\sigma(t) = \frac{1}{1+\exp(-t)}$
- "logistic": σ "squashes" $\boldsymbol{w}^T\boldsymbol{x}$ to $(0, 1)$



Loss function: (maximum likelihood function)

$$L(\mathbf{y}|\mathbf{X}, \beta) = L(y_1, y_2, \dots, y_n|\mathbf{X}, \beta) = \prod_{i=1}^{n} P(y_i|\mathbf{x}_i) = \prod_{i=1}^{n} p_i^{y_i}(1 - p_i)^{1-y_i}$$

Usually, we consider minimizing the negative log-likelihood

$$J(w) = -\sum_{n=1}^{N}(y_n \log p(y = 1|\boldsymbol{x_n}) + (1 - y_n) \log p(y = 0|\boldsymbol{x_n}))$$

Then the algorithm uses different optimization methods, like Newton's method to get the maximum-likelihood estimate β^ML.

Training stage: given a set of n data points in S = {(y1, x1),(y2, x2), ...,(yn, xn)}, we fit the model parameters β and obtain the maximum-likelihood estimate β^ML.

Regularization: Previously, we minimize the cost term $w = argminw\, J(w)$

Now, we minimize the sum of the cost term and the regularization term

$w = argminw\, J\, w + \Omega(w)$

L2 Regularization: $\|\boldsymbol{w}\|_2^2 = w_0^2 + w_2^2 + \cdots + w_k^2$

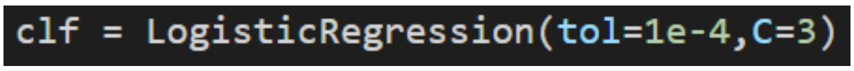
L1 Regularization: $\|\boldsymbol{w}\|_1 = |w_0| + |w_1| + \cdots + |w_k|$

By minimizing it, we force the parameters to small values to make the model simpler.

Test phase: given a novel input x∗, we desire an accurate prediction on the probability of having "0" or "1" based on the trained logistic regression model.

2. **Code**

We use the package of logistic regression from sklearn. And we adjust two parameters.



```
clf = LogisticRegression(tol=1e-4,C=3)
```

C: float, optional (default=1.0)
Inverse of regularization strength; must be a positive float. Smaller values specify stronger regularization.
tol: The error range of the iterative termination criterion.

3. SVM, Naive Bayes

SVM

1. Principle

A support vector machine (SVM) is a supervised machine learning model that uses classification algorithms for two-group classification problems. After giving an SVM model sets of labeled training data for each category, they are able to categorize new text.

It is given a training data-set of n points and each data is represented as a vector of the form (x1,y1), … , (xn,yn). The xi are features of the data, and each data can have more than one feature. And yi are either 1 or −1, each indicating the class to which the point xi belongs. And those n points can be divided into two groups by a hyper plane. Any hyperplane can be written as the set of points x satisfying $w^T x - b = 0$. There can be many hyper planes that can divided those points, but there is only one maximum spacing hyper plane. To find the maximum spacing hyper plane, the points closest to the line from both the classes are needed to be found, which are called support vectors. Then the distance between the line and the support vectors, which is called the margin needs to be maximized. The hyper plane for which the margin is maximum is the maximum spacing hyper plane according to the SVM algorithm.

The maximum spacing hyper plane can help prediction. With the xi of points that need to be predicted, the distance between the line and the points can be computed. And the predicting labels can be determined by the distance.

2. Code

Python has a specific package to train a SVM model for classification, and also predict the labels for the test data, which is called Sklearn.svm. To obtain the SVM model and use it to predict the emotional orientation of film reviews, a svm model need to be built first by using svm.SVC(). Then, the package can automatically get a smv model by fit(). Finally, the model can be used to predict the labels of the testing data by predict().

The input of the fit() should be should be the vector representation and labels of the training data. The vector representation of the training data is a sparse matrix of shape (n_samples, n_features), where n_samples is the number of samples and n_features is the number of features.

4.Neural Network (NN, LSTM, biLSTM)

1. data processing

1.1 Load data

First, we load the data. Since the data set is not an organized csv file, but each record is a txt file, we traverse the corresponding folder to read in the data and get the corresponding label.

1.2 Preliminary data processing

(A) token

For each data record obtained, we cannot directly judge the original data. We first perform word segmentation and divide each record into tokens, which are the units of model processing.

(b) stop words

Human language contains many functional words. Compared with other words, function words have no practical meaning. The most common functional words are qualifiers ("the", "a", "an", "that", and "those"). These words help describe nouns and express concepts in the text. These functional words are extremely common. Recording the number of these words in each document requires a lot of disk space. At the same time, due to their universality and function, these words rarely express information about the degree of document relevance alone. Therefore, these functional words are basically not helpful for judging the emotion of the text.

Another name for these function words is stop words. Stop words mainly include English characters, numbers, mathematical characters, punctuation marks, and single Chinese characters that are frequently used. They are called stop words because if they are encountered during text processing, the processing is stopped immediately, and they are thrown away. In this project, we also discard stop words.

(c) steam

There are many forms of the same word in English, such as the singular and plural of nouns, the present and past tense of verbs, etc., but these forms have little effect on the meaning of the vocabulary and the emotion of the text, so when dealing with English Consider the problem of stemming. So, we continue to stem the text to make the meaning of each comment easier to capture.

1.3 Construct a word dictionary

Next, we need to build our dictionary based on these corpora. The construction of the dictionary is to correspond to a number for each token to represent the token.

After counting the processed text tokens, we will find that there are many words with a frequency of only one time. Such words will increase the capacity of our dictionary and will also bring some noise to the text processing. After removing these words, on the one hand, our dictionary capacity will be greatly reduced, and model training will be accelerated. On the other hand, some noise will be reduced. Therefore, we only keep the words that appear more than 1 in the corpus in the process of constructing the dictionary.

It is worth noting that <pad> and <unk> are two initialized tokens, <pad> is used for sentence filling, and <unk> is used to replace words that have not appeared in the corpus.

With the dictionary, we can construct word-to-token mapping and token-to-word mapping,

1.4 Convert text

Based on the mapping table, we can convert the original text, that is, convert the text into a machine-recognizable code.

We construct a function that can receive a complete token type sentence and convert it into tokens according to the mapping table. In this function, we first need to get the <unk> code for use in sentence conversion later. Next, the sentence is mapped, and if there is a word that has not been seen, it is replaced with the <unk> token.

1.5 pad

After the input sequence is obtained in the previous step, since the original text does not

necessarily have the same length, in order to ensure that the sentence has the same length, the sentence length needs to be processed. For different models, the length requirements may be different. We set the desired length as N, truncate sentences with more than N words.
For sentences with less than N words, use <pad> to fill in at the end of the sentence.

1.6 Embedding Layer

Embedding layer, its main function is to map the id of each word to a vector of fixed dimensions, each time the same id is mapped to the same data, and the training is completed in continuous iterations. What the Embedding layer gets is the word vector of each word. As long as we input the corresponding id of the word, we can get its corresponding word vector. Since the direct use of one-hot encoding will make the overall network dimension very large and difficult to converge, we use embedding_layer to obtain the embedding of the vocabulary and use the low-latitude dense embedding vector as the input of the subsequent model.

2. Models

Neural Network

The DNN model processes sentences in a very simple way. For each word in the sentence, the word embedding is first obtained, and then a flatten layer is connected to splice these word embeddings, and finally the sentence vector is obtained. Then the sentence vector is passed through multiple layers of nerves to obtain the final output.

1. **Model**

In the model, we define the weights of the fully connected layer and the output layer and calculate the results. The fully connected layer uses ReLu as the activation function, binary_crossentropy is defined in loss, and calculation accuracy is defined in evaluation. Since our pos and neg samples are 1:1, so the predicted probability exceeds 0.5, we think it is pos, otherwise it is neg.

Neural networks are a set of algorithms, modeled loosely after the human brain, that are designed to recognize patterns. They interpret sensory data through a kind of machine perception, labeling or clustering raw input. The patterns they recognize are numerical, contained in vectors, into which all real-world data, be it images, sound, text or time series, must be translated.

For a nonlinear function $f^{(h)}(.)$, and for j^{th} node on specific h hidden layer:

$h_j = f^{(h)}(\sum_{i=1}^{k} w_{ij}x_i + b_j), \quad y = \sum_{j=1}^{\tilde{k}} \tilde{w}_j h_j$. Popular choice for $f^{(h)}(.)$ are tanh, the weight vector can either be updated in one go (*batch* update)

$$\vec{w}_{t+1} := \vec{w}_t - \eta \frac{\partial E(\vec{w})}{\partial \vec{w}}\bigg|_{\vec{w}_t} = \vec{w}_t - \eta \sum_i \frac{\partial E_i(\vec{w})}{\partial \vec{w}}\bigg|_{\vec{w}_t},$$

or it can be updated *sequentially* using one training example at a time:

$$\vec{w}_{t+1} := \vec{w}_t - \eta \frac{\partial E_i(\vec{w})}{\partial \vec{w}}\bigg|_{\vec{w}_t}.$$

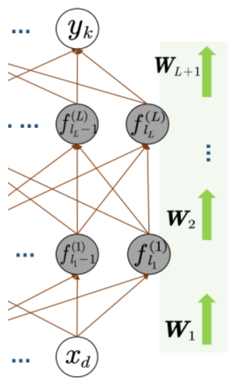


Figure: Block diagram of deep neural network architecture.

sigmoid, and relu functions. This neural network uses back propagation to solve weights for each node.

$$\frac{\partial E(\vec{w})}{\partial \vec{w}} = \sum_i \frac{\partial E_i(\vec{w})}{\partial \vec{w}},$$

2.

$$\begin{aligned}\frac{\partial E_i(\vec{w})}{\partial w_{j\to k}} &= \frac{\partial}{\partial w_{j\to k}}(h_{\vec{w}}(\vec{x_i}) - y_i)^2 \\ &= 2(h_{\vec{w}}(\vec{x_i}) - y_i)\frac{\partial h_{\vec{w}}(\vec{x_i})}{\partial w_{j\to k}} \\ &= 2(h_{\vec{w}}(\vec{x_i}) - y_i)\frac{\partial h_{\vec{w}}(\vec{x_i})}{\partial s_k}\frac{\partial s_k}{\partial w_{j\to k}} \\ &= 2(h_{\vec{w}}(\vec{x_i}) - y_i)\frac{\partial h_{\vec{w}}(\vec{x_i})}{\partial s_k} z_j.\end{aligned}$$

Experiment:

Parameter fit:

There are several parameters need to fit in order to get the highest accuracy.

- output_dim: dimension of the dense embedding. This parameter controls the dense

  embedding dimension, the only method that can be tried to get the best parameter is through trials. In this process, we try several numbers, since it a classic deep neural network, layers for dense will be much higher than LSTM and biLSTM. The attempted layers can be$2^{10}, 2^9, 2^8$.... We tried $2^{10}, 2^9, 2^8, 2^7, 2^6, 2^5, 2^4, \ldots 2^0$ for 5 different lays. Finally, we find 1024, 1024, 512,128,1 is the best combination.

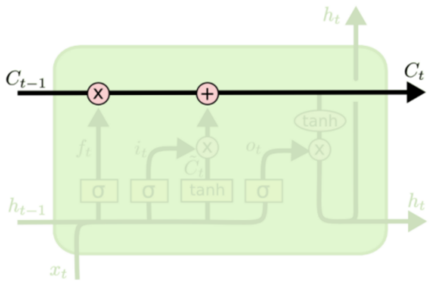
activation: active function specifies the activation function that will be used in LSTM. Common choices are tanh, relu, and sigmoid. Since there are actually 5 layors, we wrote a prthon program to loop through all 3^5 combinations, for example, sigmoid + relu + tanh + relu + relu, in order to get the best activation function. At last, the highest accuracy occurs when combination relu + relu + relu + relu + sigmoid appears.

LSTM

Compared with the simple NN model, LSTM allows us to operate on the vector sequence, which is calculated on the time series, and each calculation considers not only the previous state but also the current input. Since the text data itself is organized using certain grammars, it is more reasonable to process in time sequence.

We use LSTM to process the input embedding sequence, and then use the multi-layer neural network to classify the final output to get the final output.

1. Principles:

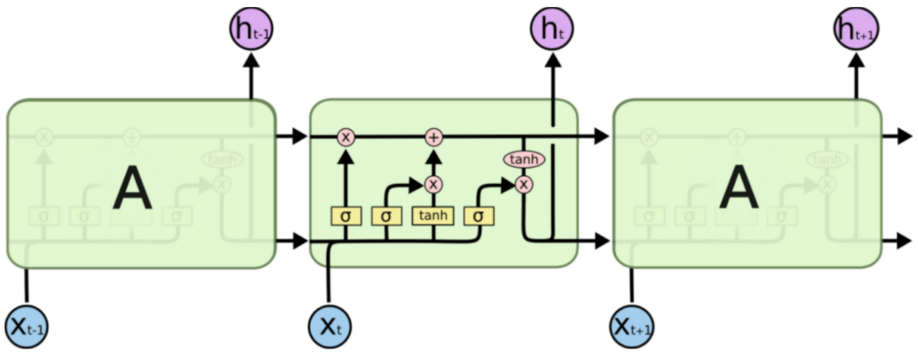


Long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture used in the field of deep learning. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell.

Core ideas behind the LSTM:

Cell State:

The cell state transformation line is at the top of an LSTM cell. There is only minor linear interactions in order to inherit old information. It is consistent with the natural human language because the context acts as a whole to express sentiment.



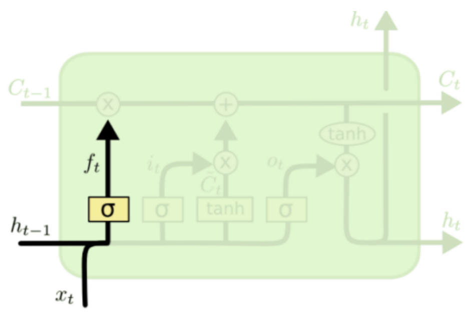
The repeating module in an LSTM contains four interacting layers.

Sigmoid Function:

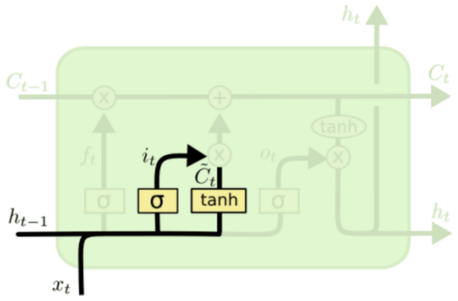
$\sigma(x) = \frac{1}{1+e^{-x}}$, it acts as a function to transform a function into a numerical value between 0 and 1. In actual language processing, it decides which proportion of data should be inherited and how much portion of data should be forgotten.

Step by step LSTM information transformation line:

1.



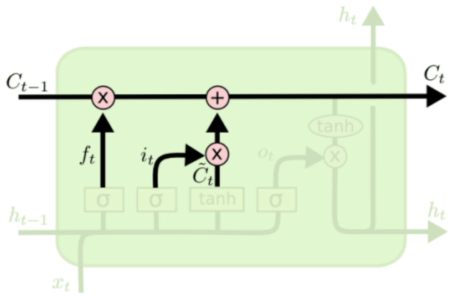
$f_t = \sigma(W_t[h_{t-1}, x_t + b_f])$. In this step, h_{t-1} is some information computed in the previous cell and, x_t is the input value specified in the present cell. $W_t[h_{t-1}, x_t + b_f]$ is function in terms of h_{t-1} and x_t to compute a combination of these two input values. After using sigmoid function, it decides the portion that how much information should be kept in the following steps.



2.
The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values need to be updated. Next, a tanh layer creates a vector of new candidate values, C_t, that could be added to the state.

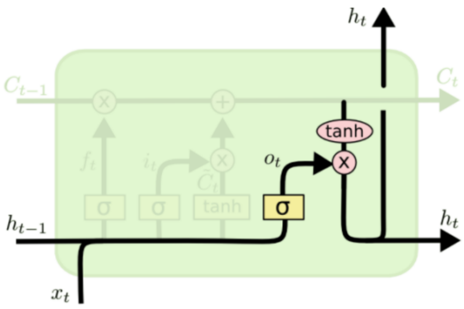
$i_t = \sigma(W_i[h_{t-1,x_t} + b_i]),\ \tilde{C}_t =$

$tanh(W_C[h_{t-1}, x_T] + b_C)$. Using these equations, $\tilde{C}_{t-1}$ can be replaced by new candidate $\tilde{C}_t$.



3.
This step updates the old cell state, C_{t-1}into the new cell state, C_t. The previous steps already decided what to do, we just need to actually do it. Multiplying the old state $f_t$by forgetting the things that has been decided to forget earlier. Then we add $i_t\tilde{C}_t$. This is the new candidate values, scaled by how much people decided to update each state value. It depends on the real contents.

Function: $C_t = f_tC_{t-1} + i_t\tilde{C}_t$.



4.
Finally, the output is decided. This output will be based on the cell state but will be a filtered version. First, a sigmoid layer is ruined to decides what parts of the cell state will be the output. The cell gate value is turned through tanh function to convert it into a numerical between 0 and 1. Then multiply it by the output of the sigmoid gate. The function in this transition line is $o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) and h_t = o_t tanh(C_t)$.

In real application, the activation function in the transformation always depends on the context. In our project, we define activation function to to relu function and sigmoid function.
Algorithm:
Python has an open-source library Tensorflow, which can train these models automatically.

2. Code

2.1 Packages

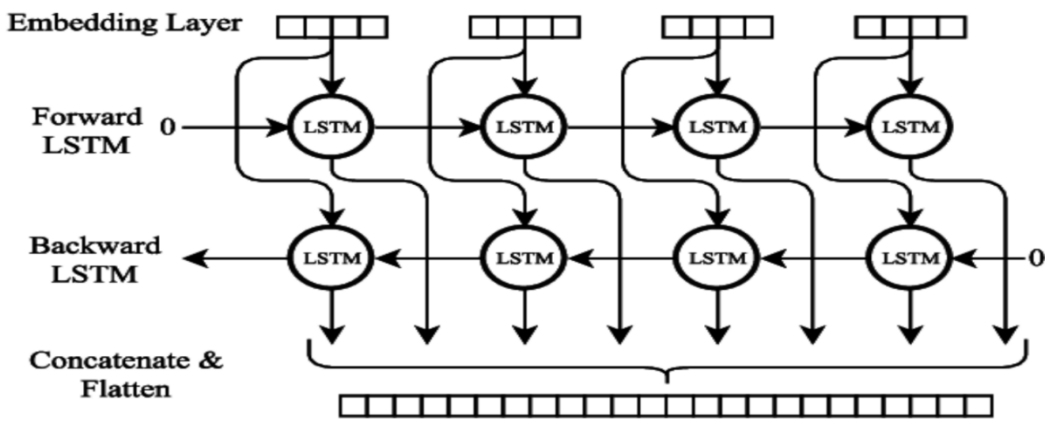
Keras from Tensorflow: Keras is an open-source, Ml library that is written in Python. Keras offers something unique in machine learning: a single API that works across several ML frameworks to make that work easier. We recommend using Keras for most, if not all, of your machine learning projects.
2.2 Parameter fit:
There are several parameters need to fit in order to get the highest accuracy.

- output_dim: dimension of the dense embedding. This parameter controls the dense embedding dimension, the only method that can be tried to get the best parameter is through trials. When output_dim = 20, 30, 40, 50 ,60 ,70…100, the highest accuracy happens when output_dim = 60. Then, we test all the integers between 55-68. Finally, when output_dim = 64, the accuracy reaches highest.
- batch_size: this is a batch size which controls the number of dimensions that will flow in the LSTM layers. The usual set up is dimension >300 considering the the size of all the data. When output_dim = 300, 350… 600, the top 2 accuracy happens when batch_size = 450 and 500. Then, we test all the integers between 450-500. Finally, when batch_size = 512, the accuracy reaches highest.
- activation: active function specifies the activation function that will be used in LSTM. Common choices are tanh, relu, and sigmoid. Since there are actually 4 layers, we wrote a prthon program to loop through all 81 combinations, for example, sigmoid + relu + tanh + relu, in order to get the best activation function. At last, the highest accuracy occurs when combination relu + relu + relu + sigmoid appears.

biLSTM

LSTM can only process the sequence from left to right, which makes the embedding of each position only obtain the previous information. The meaning of the vocabulary in each position should be affected by the left and right sides together. One-way information will obviously affect the effect of the model, so we further use biLSTM to process the input Embedding sequence. Similarly, after obtaining the output, a multi-layer neural network is used for classification processing.

1.Model



A Bidirectional LSTM, or biLSTM, is a sequence processing model that consists of two LSTMs: one taking the input in a forward direction, and the other in a backwards direction. BiLSTMs effectively increase the amount of information available to the network, improving the context available to the algorithm (e.g., knowing what words immediately follow and precede a word in a sentence).

One sentence pass BiLSTM at a time. For each input sentence of n words, we define an n-dimensional vector x whose elements are the indices in V corresponding to words appearing in the sentence, preserving the order. The input x is passed to an Embedding Layer that returns the sequence S = $\{w_j \ | \text{j} = x_1, x_2, x_3, \ldots x_n\}$ where w_j is the j^{th} row of a dense matrix W ∈ R | $V_{1\times d}$, where d ∈ N is a hyperparameter. The vector w_j represents a low-dimensional vector representation, or word embedding, whereas W is the corresponding embedding matrix. The sequence of word embeddings S is then passed as input to two LSTM layers that process it in opposing directions (forwards and backwards). Figure 2 shows the LSTM layers in their ”unrolled” form as they read the input. Each LSTM layer contains k LSTM memory cells. The output from each of the LSTM layers is H = $\{h_t \in R_k$ | t = 1, 2, ..., n$\}$. Next, we concatenate and flatten $H_{forward}$ and$H_{backward}$, obtaining a vector p ∈ R_{2kn}.

2. Experiment:

Parameter fit(调参过程):

There are several parameters need to fit in order to get the highest accuracy.

- output_dim: dimension of the dense embedding. This parameter controls the dense

  embedding dimension, the only method that can be tried to get the best parameter is through trials. When output_dim = 20, 30, 40, 50 ,60 ,70…100, the highest accuracy happens when output_dim = 60. Then, we test all the integers between 55-68. Finally, when output_dim = 64, the accuracy reaches highest.

- batch_size: this is a batch size which controls the number of dimensions that will

  flow in the LSTM layers. The usual set up is dimension >300 considering the the size of all the data. When output_dim = 300, 350… 600, the top 2 accuracy happens when batch_size = 450 and 500. Then, we test all the integers between 450-500. Finally, when batch_size = 512, the accuracy reaches highest.

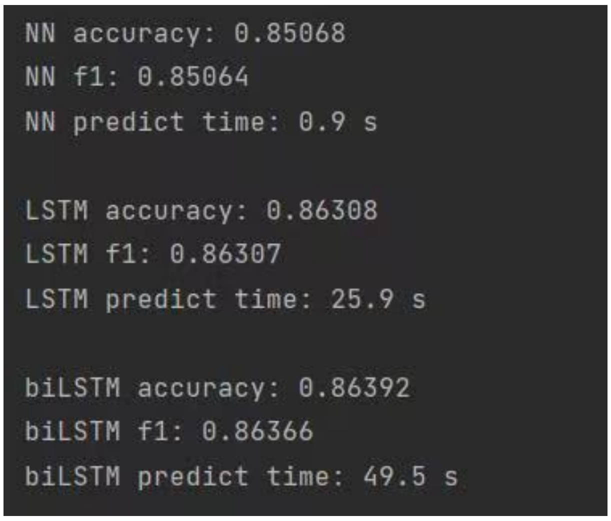
- activation: active function specifies the activation function that will be used in

  LSTM. Common choices are tanh, relu, and sigmoid. Since there are actually 4 layors, we wrote a prthon program to loop through all 81 combinations, for example, sigmoid + relu + tanh + relu, in order to get the best activation function. At last, the highest accuracy occurs when combination relu + relu + relu + sigmoid appears.

3. results

The results are shown in the figure. Among the three models, NN has the lowest accuracy of 0.85068, and biLSTM has the highest accuracy rate of 0.86392

Models	Accuracy	F1	Time(s)
NN	0.85068	0.85064	0.9
LSTM	0.86308	0.86307	25.9
biLSTM	0.86392	0.86366	49.5



```
NN accuracy: 0.85068
NN f1: 0.85064
NN predict time: 0.9 s

LSTM accuracy: 0.86308
LSTM f1: 0.86307
LSTM predict time: 25.9 s

biLSTM accuracy: 0.86392
biLSTM f1: 0.86366
biLSTM predict time: 49.5 s
```

IV. Model combination and implementation

Bert+KNN
Model: KNeighborsClassifier (n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=1, **kwargs)
Accuracy: 59.2%, time: 167s
Hash + Logistics:
(1) Model: LogisticRegression(penalty='l2', dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='liblinear', max_iter=100, multi_class='ovr', verbose=0, warm_start=False, n_jobs=1)
(2) Parameters:
Penalty: l1 is only suitable for liblinear. And l2 is suitable for all.
l1+liblinear: accuracy= 88.1%, f1 = 0.8822, time: 114.3399s
Solver: liblinear+l2: accuracy= 88.2%, f1 = 0.8815, time: 105.932
lbfgs+l2: accuracy= 88.2%, f1 = 0.8815, time:147.53
sag+l2: accuracy= 88.2%, f1 = 0.8815, time:106.04
newton-cg+l2 = 88.2%, f1 = 0.8815, time:121.97
Because the total time is similar, we choose lbfgs+l2 (default)
C (Regularization parameter): we choose C from 1 to 10 and test the accuracy. The

conclusion is that when C=2, the accuracy is highest.
(3) Input: x is a matrix, and each row represents the feature vector of a text. Y is a column vector, and each row represents the label (1 or 0) of a text.
(4) Result: the accuracy is 88.2% and the total time is 105.932s.

Hash + KNN:
(1) Model: KNeighborsClassifier (n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=1, **kwargs)
(2) Parameters:
n_neighbors: because we want to classify the data into two classes (positive and negative), we choose n_neighbor=2.
Weights: 'uniform': uniform weights. All points in each neighborhood are weighted equally.
Accuracy = 62.6%, f1= 0.5196, time=133.828s
'distance': weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
Accuracy = 63.7 %, f1 = 0.629, time= 141.414s
So we choose weights= "distance"
p: integer, optional (default = 2) Power parameter for the Minkowski metric.
p = 1, manhattan_distance (l1) (given weights= "distance")
accuracy = 57.4%, f1 = 0.481, time=257.325s
p = 2, euclidean_distance (l2) (given weights= "distance")
accuracy = 63.7 %, f1 = 0.629, time= 141.414s
Based on accuracy and efficiency, we choose p=2 (default).
(3) Input: x is a matrix, and each row represents the feature vector of a text. Y is a column vector, and each row represents the label (1 or 0) of a text.
(4) Result: the accuracy is 63.7% and the total time is 141.414s.

Hash + SVM:
(1) **Model: svm.SVM()**
(3) Input: x is a matrix, and each row represents the feature vector of a text. Y is a column vector, and each row represents the label (1 or 0) of a text.
(4) Result: the accuracy is 88.4%, f1_scoreaccuracy: 0.884 and the total time is 1764 s.

Glove + Logistics:
(1) Model: LogisticRegression(penalty='l2', dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='liblinear', max_iter=100, multi_class='ovr', verbose=0, warm_start=False, n_jobs=1)
(2) Parameters:
Penalty: l1 is only suitable for liblinear. And l2 (default) is suitable for all.
l1+liblinear: accuracy=62.9075%
Solver: liblinear+l2: accuracy=62.7715%
Lbfgs+l2: accuracy=62.9115%

Sag+l2: accuracy=62.9115%

Newton-cg+l2: accuracy= 62.9075%

Because the total time is similar, we choose lbfgs+l2 (default)

C (Regularization parameter): we choose C from 1 to 10 and test the accuracy. The conclusion is that when C=3, the accuracy is highest.

(3) Input: x is a matrix, and each row represents the feature vector of a text. Y is a column vector, and each row represents the label (1 or 0) of a text.

(4) Result: the accuracy is 62.9115%, f1: 0.6402 and the total time is 23.22s.

Glove + KNN:

(1) Model: KNeighborsClassifier (n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=1, **kwargs)

(2) Parameters:

n_neighbors: because we want to classify the data into two classes (positive and negative), we choose n_neighbor=2.

Weights: 'uniform': uniform weights. All points in each neighborhood are weighted equally.

Accuracy = 55.0266%, time=135.08s

'distance': weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

Accuracy = 55.2586%, time=135.04s

So we choose weights= "distance"

p: integer, optional (default = 2) Power parameter for the Minkowski metric.

p = 1, manhattan_distance (l1) (given weights= "distance")

accuracy = 54.9346%, time=149.90s

p = 2, euclidean_distance (l2) (given weights= "distance")

accuracy = 55.2586%, time=135.04s

Based on accuracy and efficiency, we choose p=2 (default).

(3) Input: x is a matrix, and each row represents the feature vector of a text. Y is a column vector, and each row represents the label (1 or 0) of a text.

(4) Result: the accuracy is 55.2586%, f1: 0.5435 and the total time is 135.04s.

Word2vec + Logistics

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

(2) Parameters:

Penalty: l1 is only suitable for liblinear. And l2 is suitable for all.

l1+liblinear: accuracy=85.93%, time=105.09s

Solver: liblinear+l2: accuracy=85.94%, time=68.16s

Lbfgs+l2: accuracy=85.976%, time=61.09s

Sag+l2: accuracy=85.95%, time=65.84s

Newton-cg+l2 = 85.95%, time=67.24s

Because the total time is similar, we choose lbfgs+l2 (default)

C (Regularization parameter): we choose C from 1 to 10 and test the accuracy. The conclusion is that when C=3, the accuracy is highest.

(4) Input: x is a matrix, and each row represents the feature vector of a text. Y is a column vector, and each row represents the label (1 or 0) of a text.

(3) Result: the accuracy is 85.976%, f1:0.8603 and the total time is 61.09s.

Word2vec + KNN

(1) Model: KNeighborsClassifier (n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=1, **kwargs)

(2) Parameters:

n_neighbors: because we want to classify the data into two classes (positive and negative), we choose n_neighbor=2.

Weights: 'uniform': uniform weights. All points in each neighborhood are weighted equally.

Accuracy = 66.56%, time=307.56s

'distance': weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

Accuracy = 68.03%, time=297.24s

So, we choose weights= "distance"

p: integer, optional (default = 2) Power parameter for the Minkowski metric.

p = 1, manhattan_distance (l1) (given weights= "distance")

accuracy = 67.84%, time=331.53s

p = 2, euclidean_distance (l2) (given weights= "distance")

accuracy = 68.03%, time=297.24s

Based on accuracy and efficiency, we choose p=2 (default).

(3) Input: x is a matrix, and each row represents the feature vector of a text. Y is a column vector, and each row represents the label (1 or 0) of a text.

(4) Result: the accuracy is 68.03%, f1: 0.6675 and the total time is 297.24s.

Word2vec + SVM

(1) Model:

SVM (self,data,labels,vec=None,vec_size = 250)

(2) Parameters:

Data: The list tokenized texts of movie comments in each document.

Labels: The 0/1 numeric labels we accessed in the Word2Vec function.

Vec: A zero matrix of size (50000, self.vec_size).

Vec_size: The size of the Word2Vec vectors.

In the gensim.models.Word2Vec() function, we also heve several parameters:

Vec_size: The size of the Word2Vec vectors consistent with the previous definition.

Window: The size of window, e.g., the number of adjacent words that forms vectors for computing the word vectors.
Workers: The number of parallelisms used to control training.
Min_count: Words with a frequency less than min_count is discarded.
Vec_size = 100, accuracy = 71%
Vec_size = 200, accuracy = 74.09%
Vec_size = 250, accuracy = 75.48%

(3) Inputs

The SVM class will inherit the word vector generated by Word2Vec and generate a hyperplane boundary of positive/negative sentiment use the vectors. The svm.SVC() function we imported from module sklearn.svm could return the result of SVM prediction of sentiments.

(4) Result

The accuracy of Word2Vec with SVM is 75.48%

TF-IDF + Naïve_Bayes

Based on the feature vectors previously generated by tfidf, we can use the naïve bayes model to fit the data. The default of "max_feature" is none. If we use the default set, there'll be no result.

Parameter fitting:

But when we set the max_feature to 10000, the result becomes

The accuracy of GaussianNB is 0.6226
The accuracy of MultinomialNB is 0.82428
The accuracy of BernoulliNB is 0.81884

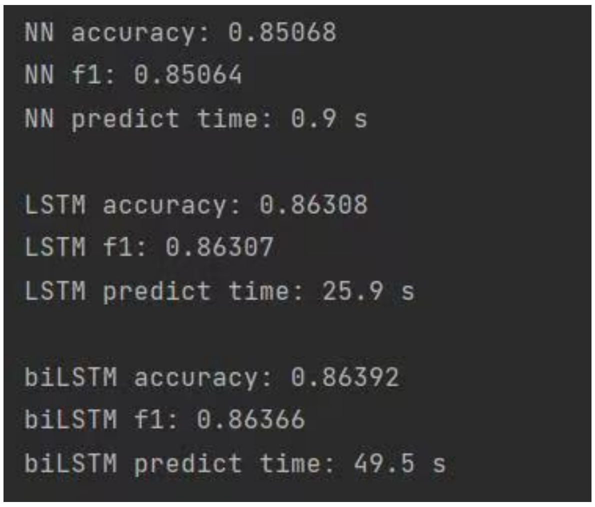
then set the parameters to 1000, the result becomes

The accuracy of GaussianNB is 0.76896
The accuracy of MultinomialNB is 0.80752
The accuracy of BernoulliNB is 0.79792

So we can see that the parameters can't be too small, after many trial and error, we got the best parameter: 3000, and the result is

The accuracy of GaussianNB is 0.74312, f1_score: 0.715
The accuracy of MultinomialNB is 0.829， f1_score: 0.827
The accuracy of BernoulliNB is 0.82316, f1_score: 0.819

NN, LSTM, biLSTM



```
NN accuracy: 0.85068
NN f1: 0.85064
NN predict time: 0.9 s

LSTM accuracy: 0.86308
LSTM f1: 0.86307
LSTM predict time: 25.9 s

biLSTM accuracy: 0.86392
biLSTM f1: 0.86366
biLSTM predict time: 49.5 s
```

V. Conclusion

From all the models above: the combination of hash and SVM has the highest accuracy. So the final accuracy of our project is 88.4%.

1. Data Processing

In our project, our group apply TF-IDF, word2vec, Glove and Bert to process data. For this project, the above methods have some advantages and disadvantages as follow:

TF-IDF: the key to TF-IDF is to filter out less important words and to retain more important words as features. It focuses on the role of words in the text classification. As our results shown, TF-IDF performs well in the text sentiment analysis. Disadvantages: ignore the associations between text contexts.

Hash Technique: hash technique is useful when dealing with large amounts of words after tokenization. Disadvantages: similar to TF-IDF, hash technique is belong to bag of words models so it also assumes that all words are independent. It ignores the contextual relationship between words in the text.

Word2vec & Glove: both of them can represent each word with a vector based on co-occurrence matrix. But they also have some differences.
Word2vec: the key of word2vec is to train each local content window separately. It considers the influence of the context of words.
Disadvantages: word2vec make less use of the information of global co-occurrence matrix. In addition, word2vec can deal with polysemous words well.
Glove: d is an improvement based on word2vec. Glove make good use of the information contained in the co-occurrence matrix. The key to glove is to form a matrix in the shape of (number of words, number of words) firstly and do dimension

reduction for the matrix. So, in our project, the code must build a matrix in the form of (250934*250934) firstly. But due to the big size of matrix, the requirement for computer's running memory is high and normal computers can handle the task. So, our group do some reduction of words for the original word bags. But if we cut down words over, the accuracy will be affected by the reduction of words.

Bert: advantages: 1. bert can capture bidirectional context information 2. Bert is long term dependency. Disadvantages: for the project, word processing with bert converges slowly. It takes more than 40 hours to process the given 50000 texts totally.

2. Model

For the project, our group use logistic regression, Naïve Bayes, KNN, SVM and neural network. For this project, the above methods have some advantages and disadvantages as follow:

Word-Level Sentiment Analysis (NOT machine learning):
Disadvantages: (1) the word-level sentiment analysis ignores the context of words and the relationship between each word. (2) the weight of each word is equal, and it cannot extract more important words.

Logistic regression: logistic regression plays a significant role in dealing with binary classification. And as the result shown, logistic regression does well in both accuracy and efficiency.

Naïve Bayes: the target of naïve bayes is to find p (label/word features) and it fits the target of our project well. Disadvantages: naïve bayes makes assumptions that all words are independent, and all features are equally important. The first assumption will make some influences on the accuracy. And because IF-IDF help extract more features, the second assumption' impacts are reduced to some extent.

KNN: when the size of features is large, KNN needs a large amount of computation so its efficiency is low.

SVM: in our project, the key of SVM is to build a hyperplane and solve binary classification problems. For our project, the accuracy of SVM is relatively high but the efficiency of SVM is relatively low.

Neural network: we make use of NN, LSTM and biLSTM to achieve classification. Neural network is a kind of advanced algorithm for text sentiment analysis. And as our results shown, the accuracy of biLSTM is highest.

VI. Distribution