北京师范大学 香港浸会大学 联合国际学院
BEIJING NORMAL UNIVERSITY · HONG KONG BAPTIST UNIVERSITY
UNITED INTERNATIONAL COLLEGE

Classification of Negative Emotions in Online Comments

[Group 25]

[Frank 谢欣言 1930026136]

[Tony 焦海旭 1930026057]

A report submitted to

Dr. Rui MENG & Dr. Zhe XUANYUAN

for

DS4023: Machine Learning (1002)

Data Science (DS)

Division of Science and Technology (DST)

[2022.5.23]

Contents

1. Introduction

1) Project description:

This project aims at applying machine learning algorithms (including but not limited to those covered in our lectures) to solve real-world tasks or conducting machine learning research.

2) Work flow:

We used data from the website **Kaggle** to conduct study on comments from social media. The advantage of this method is that it allows us to collect real samples of comments posted on various social media sites, which can have positive effects on the internet.

After analyzing and **visualizing** the data, we decided to remove the noise from the data by **preprocessing** it. This process involved removing any outliers and ensuring that the data are completely clean.

After that, we use 3 NLP methods to transform texts into embeddings: **TFIDF** based on term frequency, pretrained word embeddings such as **word2vec**, **fasttext**, and **glove**, and finally sentence embeddings **USE**.

We then implemented **logistic regression** and **BP Neural Network** by ourselves, and performed a variety of other techniques to compare the performance, such **LSTM**, **SVM**, and **ensemble learning** methods such as **XGBoost** for boosting and **extra tree** for bagging. We finally used the **AUC-ROC** score to compare the performance of these models.

Finally, we decided to analyze the data crawled from the website **"BiliBili Game"** using the best and most efficient model **XGBoost**, to find out if there are any real cases of users being harassed under the game **"King of Honors"**.

2. Our Topic

1) Topic:

The goal of this study is to develop a **Negative Comment Classification** system that is based on **Natural Language Processing** and **Emotion Recognition**.

2) **Background** and our **motivation**:

The rise of social networking sites has allowed people to share their opinions and ideas. However, there are also people who are known to spread toxic and hate speech. These people are referred to as “keyboard warriors”.

These toxic comments on social networking sites have become a serious issue due to the lack of proper supervision and discipline, and there is a challenge for the sites to control the spread of these messages. The task of identifying and preventing toxic comments on social networking sites is a **multi-class classification system.**

3) The **goal** of our topic:

In this study, we will build models to classify the posts that were obtained from various social networking sites into six categories: t**oxic, severe_toxic, obscene, threat, insult,** and **identity_hate**.

3. Survey on Existing Techniques about the Topic

1) While doing survey on the techniques on this topic, we found some models for **classification** such as **SVM**, **Naïve Bayes**, **Decision Tree** and **Logistic Regression.**
2) We also got familiar to some neural network models: **BP NN, Recurrent Neural Network (RNN)** and **Long Short-Term Memory (LSTM)**.
3) Besides, there are also some **Ensemble Learning** methods for this topic: **Extra Tree**, **Random Forest** for **bagging** and **AdaBoost, XGBoost** for **boosting**.
4) For **Natural Language Processing (NLP)**, if we want to process Chinese data

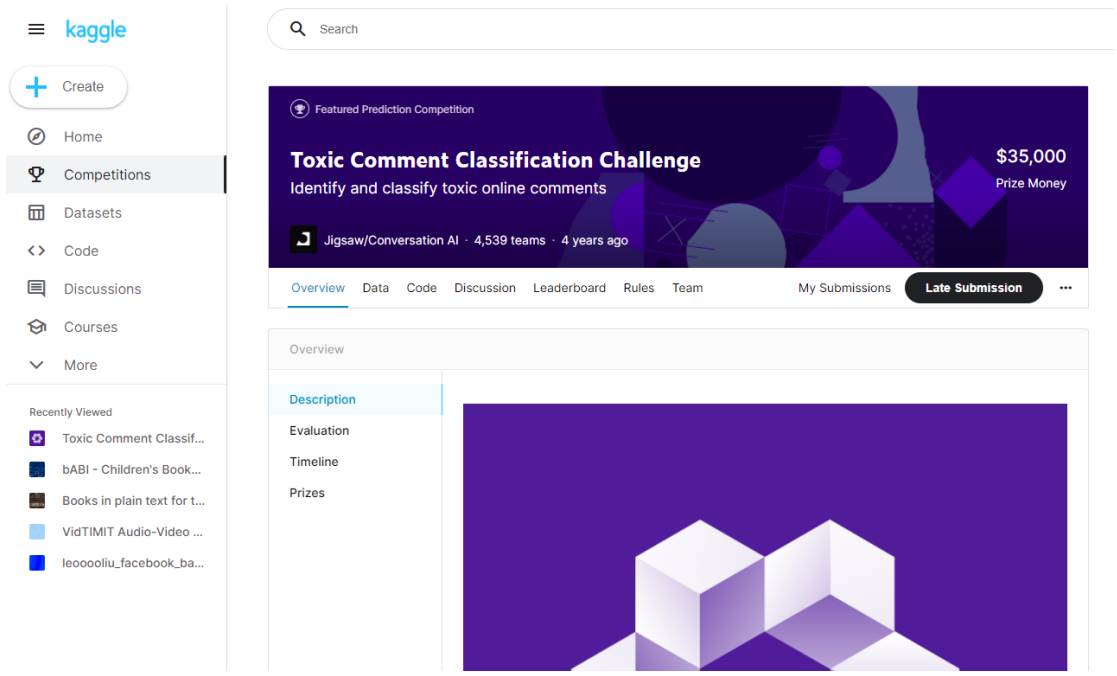
set, we can use **Snow NLP**.

For English texts, we can use **TFIDF** for term frequency, pretrained word embeddings such as **fasttext**, **glove** and **word2vec** for words, and **Universal Sentence Encoder (USE)** for sentences.

4. Collect, Read and Analyze Relevant Data

1) Data set description:

Our dataset was downloaded from https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge, and it has three files: **test.csv test_labels.csv, train.csv.**



The data structure is like following:

In [12]:
```
train=pd.read_csv(r"C:\Users\78531\Desktop\dataset\train.csv")
test=pd.read_csv(r"C:\Users\78531\Desktop\dataset\test.csv")

train.head(10)
```

Out[12]:

	id	comment_text	toxic	severe_toxic	obscene	threat	insult	identity_hate
0	0000997932d777bf	Explanation\nWhy the edits made under my usern...	0	0	0	0	0	0
1	000103f0d9cfb60f	D'aww! He matches this background colour I'm s...	0	0	0	0	0	0
2	000113f07ec002fd	Hey man, I'm really not trying to edit war. It...	0	0	0	0	0	0
3	0001b41b1c6bb37e	"\nMore\nI can't make any real suggestions on ...	0	0	0	0	0	0
4	0001d958c54c6e35	You, sir, are my hero. Any chance you remember...	0	0	0	0	0	0
5	00025465d4725e87	"\n\nCongratulations from me as well, use the ...	0	0	0	0	0	0
6	0002bcb3da6cb337	COCKSUCKER BEFORE YOU PISS AROUND ON MY WORK	1	1	1	0	1	0
7	00031b1e95af7921	Your vandalism to the Matt Shirvington article...	0	0	0	0	0	0
8	00037261f536c51d	Sorry if the word 'nonsense' was offensive to ...	0	0	0	0	0	0
9	00040093b2687caa	alignment on this subject and which are contra...	0	0	0	0	0	0

- The **first column** of the dataset contains the unique **id** of each row. This is an indication that the content text is included in the dataset.
- The **second column** is the **content text**, and we mainly focus on this to do our text mining research. And the other six columns are the six descriptive adjectives of the given content text, they are **toxic, severe_toxic, obscene, threat, insult, identity_hate.**
- The **second column** of the dataset is the **content text**. We mainly focus on this section to perform text mining research. The six columns are the adjectives that describe the content text: **toxic, insult, obscene, identity-hate, and threat**.
- **Remark**: The **test_labels.csv** is used to test the accuracy of each model running result, and interestingly, it includes many labels of **'-1'** (Basically, 1 stands for true, and 0 stands for false), this is to deter hand labeling since this dataset is served for an online competition. The test.csv in our dataset has the true label columns for each row, and thus, we intuitively used test.csv to achieve the process of predicting based our models and calculating the **AUC-ROC** result.

  The **test_labels.csv** file is used to test the accuracy of the models that predict on the dataset. It includes various labels that are designed to deter hand labeling since this is a public dataset that is used for an online competition. We intuitively used test.csv to perform the process of

calculating the AUC-ROC result.

2) **Data visualization:**

Data visualization is an important part of data mining. It can help us gain visual insights about the data set, such as some salient features or patterns in the data set, which can help us choose the appropriate machine learning algorithm to apply.

Firstly, we conducted the job of counting each descriptive adjective total number and drew some graphs to display and compare the figures:

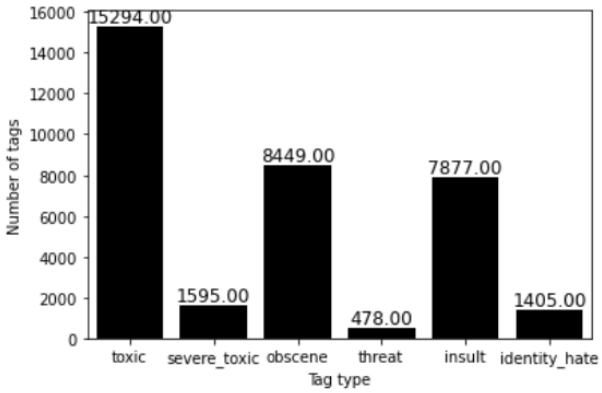
- We first drew a **histogram** showing the relative number of hate tags in the dataset and found a huge category **imbalance**. **"Toxic"** tags are the most, and **"threat"** tags are the least.

```
In [3]:
1 import pandas as pd

In [4]:
1 train=pd.read_csv("train.csv")
2 test=pd.read_csv("test.csv")

In [5]:
1 hate_tag_count=train.iloc[:,2:].sum()
2 hate_tag_count

Out[5]: toxic           15294
        severe_toxic     1595
        obscene          8449
        threat            478
        insult           7877
        identity_hate    1405
        dtype: int64
```

In [6]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

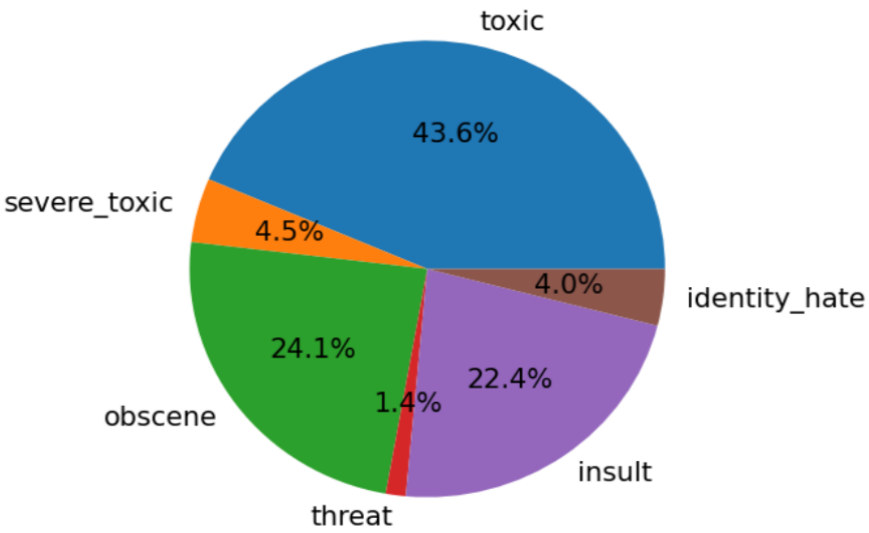
In [7]:
```
name_list=["toxic","severe_toxic","obscene","threat","insult","identity_hate"]
num_list = [15294,1595,8449,478,7877,1405]
width=0.3;#柱子的宽度
index=np.arange(len(name_list));
for a,b in zip(index,num_list):   #柱子上的数字显示
    plt.text(a,b,'%.2f'%b,ha='center',va='bottom',fontsize=12);
sns.barplot(x=name_list, y=num_list, color='black',orient='v')
plt.ylabel('Number of tags')
plt.xlabel('Tag type')
```

Out[7]: Text(0.5, 0, 'Tag type')



- And next we drew a **pie chart**, which could more precisely display the proportion of each adjective, and the differences are more intuitive in this way.

n [14]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.figure(dpi=150)
labels ="toxic","severe_toxic","obscene","threat","insult","identity_hate"
fraces = [15294,1595,8449,478,7877,1405]
plt.pie(x=fraces, labels= labels,autopct='%3.1f%%')
plt.show()
```



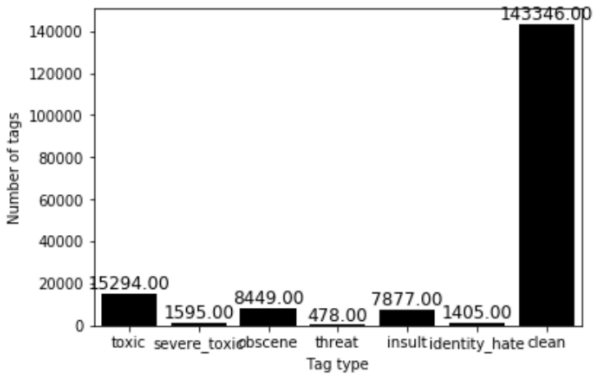
- Also, we plotted a **new histogram** after adding a column for **"clean"** tags, which was built under the case that if a content text gains all zero evaluation in six labels, we assumed it could be a **"clean"** tag. Interestingly, it was found that a significantly large number of comments were clean, while this phenomenon indicates that our dataset is quite **unbalanced.**

In [9]:
```
#Add an additional column for clean comments
train['clean']=(train.iloc[:,2:].sum(axis=1)==0).astype(int)
```

In [10]:
```
tag_count=train.iloc[:,2:].sum()
tag_count
```

Out[10]:
```
toxic            15294
severe_toxic      1595
obscene           8449
threat             478
insult            7877
identity_hate     1405
clean           143346
dtype: int64
```

In [9]:
```
#plot
name_list=["toxic","severe_toxic","obscene","threat","insult","identity_hate","clean"]
num_list = [15294, 1595, 8449, 478, 7877, 1405, 143346]
width=0.3;#柱子的宽度
index=np.arange(len(name_list));
for a,b in zip(index,num_list):   #柱子上的数字显示
    plt.text(a,b,'%.2f'%b,ha='center',va='bottom',fontsize=12);
sns.barplot(x=name_list, y=num_list, color='black',orient='v')
plt.ylabel('Number of tags')
plt.xlabel('Tag type')
plt.show()
```



3) Collect and crawl data by ourselves:

We also crawled a real data set from **"Bilibili Games"**, which are some newest comments on the game **"King of Honors"**, and we will use them for our test in the **"8. Application Using Crawled Data from Bilibili"** part.

程序运行时间:71.66904735565185秒

In [16]:
```
resultpd.head(50)
```

Out[16]:

	content	grade	publish_time	up_count	down_count	user_name	user_level
0	虽然经常都在玩王者可是这个游戏真的太糟糕了根本不让我跟\n如果不是充了钱我玩都不玩	2	2021-12-12 21:44:36	0	0	硬币够了再改名	4
1	无话可说	2	2021-12-12 21:11:28	0	0	幻梦无枉	3
2	(ˊ▽ˋ)	10	2021-12-12 21:09:55	0	0	一英小白	3
3	扎不多得勤!	2	2021-12-12 20:48:35	0	0	乌鱼子蜜拌饭	5
4	垃圾游戏，我被安排了，给个一星不过分吧	2	2021-12-12 20:32:44	0	0	hwbbsuunw	5
5	非常好!	10	2021-12-12 20:27:38	0	0	卖个萌给爷看	4
6	垃圾游戏，一星都不想给!	2	2021-12-12	1	1	火石崽	4

5. Data Preprocessing:

1) Remove Noise:

Because there are lots of characters or strings that are not in English and cannot be recognized by the models, so we regarded them as noises and

removed them.

In our data set before removing noise:

	002c9cccf2f1d05b	this inane conversation. (talk) Ireland whose official name is Ireland . 鈥? Preceding unsigned comment added by 109.76.191.188 http://www.constitution.ie/reports/Constitutionoflreland.pdf reading for you . Also , I see http://www.constitution.ie/reports/mbunreachtnaheireann.pdf , as you speak both . 鈥?Preceding unsigned comment added by 109.78.224.50 "

Our codes to remove noise:

```
links = '(http://.*?\s)|(http://.*)'
ip_addr = '\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}'
users = '\[\[User.*'
newline = '\\n'
print(train['comment_text'].str.contains(links).sum())
print(train['comment_text'].str.contains(ip_addr).sum())
print(train['comment_text'].str.contains(users).sum())
print(train['comment_text'].str.contains(newline).sum())
```

Noise

```
/usr/local/lib/python3.6/dist-packages/pandas/core/strings.py:2001: U
use str.extract.
  return func(self, *args, **kwargs)
```

```
2766
6313
196
59357
```

```
def clean(comment):
    import re
    comment=comment.lower()
    comment=re.sub(links,'',comment)
    comment=re.sub(ip_addr,'',comment)
    comment=re.sub(users,'',comment)
    comment=re.sub(newline,'',comment)
    return comment
```

```
train['comment_text']=train['comment_text'].map(lambda i:clean(i))
test['comment_text']=test['comment_text'].map(lambda i:clean(i))
```

2) Remove Stopwords:

The Stopwords are those words in the comment text, which don't add any

physical meaning to the comment, but are generally used as connectors in the natural language in order to make the text grammatically correct.

Such as: ***is, am, are, was, who, was, were***, etc. and some **punctuations**.

We removed the stop words from the comment text using the **NLTK** Stopwords corpus.

3) Stemming:

In the natural language, words have different derived forms (such as different tenses). However, as these words are only grammatically significant, they must be converted into the same form, as they mean the same.

Stemming involves converting a word from its normal form to its base stem:

词干提取算法	*Porter*
例子1： • *argue* • *argued* • *argues* • *arguing* • *argus*	结果 • *argu* • *argu* • *argu* • *argu* • *argu*
例子2: • *caring* • *was* • *this*	结果 • *care* • *wa* • *thi*

Here are our codes for removing Stopwords and Stemming:

```
import re
```

```
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
from sklearn.feature_extraction.text import CountVectorizer
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```
def clean(comment):
  comment = comment.lower()
  comment = re.sub('[^a-zA-Z]',' ', comment)
  comment = comment.strip()
  comment = comment.split()
  stop_words = set(stopwords.words('english'))
  stemmer = PorterStemmer()
  comment = [stemmer.stem(word) for word in comment if word not in stop_words and len(word)<30]
  comment = ' '.join(comment)
  return comment
```

remove Stopwords

Stemming

4) Natural Language Processing:

Because the features in our dataset are texts, we need to transform them into vectors which can be understood and calculated by computer.

In our project, we tried 3 methods for this procedure:

i) TFIDF:

TF-IDF is used to evaluate the importance of a word to a document set or one of the documents in a corpus.

- **TF: Term Frequency**.
  Which is the frequency term t shows in document d:

$$\mathbf{TF\ (t,\ d)}$$

- **IDF: Inverse Document Frequency.**

  Which is calculated by:

The number of documents containing term 't'

$$IDF(t) = \log \frac{1 + |D|}{1 + df(t)} + 1$$

Total number of documents

- **TF-IDF:**

$$TFIDF = TF(t,d) \times IDF(t)$$

The **TF** increases if the word is frequent in the document. However, if the word appears in most of the documents, then **IDF** decreases. So it considers both local and global impact of a word in a comment, and it penalizes those words that have a high document frequency and are not meaningful.

Here we used the **TfidfVectorizer** in **sklearn** package, which can transform **texts** into **matrices** based on term frequency:

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf=TfidfVectorizer(max_features=5000,strip_accents='unicode',stop_words='english',token_pattern=r'\w{2,}')

tfidf.fit(X_train)
X_train_feat=tfidf.fit_transform(X_train)
X_train_feat.shape

(80000, 5000)

X_val_feat=tfidf.transform(X_val)
X_val_feat.shape

(20000, 5000)
```

TFIDF also remove stopwords

TFIDF also do the nomalization

- **Tokenization** and **Normalization**:

  Before using TFIDF, we need to do the **tokenization** which can transform the whole sentences into separate tokens, and do the **normalization** after that.

ii) **Pretrained Word Embeddings (fasttext, Glove, word2vec):**

These embeddings are pretrained by Google or other organizations, which work well on transforming **words** into **embeddings** (vectors).

Because the **keras** package support well for **LSTM** and has a convenient method to use these embeddings, we only use them in **LSTM** in our project.

- **Tokenization**:

  To use these pretrained embeddings, we also need to do the **tokenization** to our texts first:

  The preprocessed data is still in natural language form, which is not

understandable by the machine. So, it needs to be broken down into tokens or words, and these tokens need to be encoded in the form of integer or floating-point numbers, to be used by machine learning algorithms for prediction.

In **LSTM**, we just need the tokens to train the model.

```
from keras.preprocessing.text import Tokenizer
```

```
max_features = 20000
tokenizer = Tokenizer(num_words=max_features)
tokenizer.fit_on_texts(list(list_sentences_train))
list_tokenized_train = tokenizer.texts_to_sequences(list_sentences_train)
list_tokenized_test = tokenizer.texts_to_sequences(list_sentences_test)
```

```
embedding_matrix = loadEmbeddingMatrix('word2vec')
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_la
4.0.0, use self instead).

Loaded 306943 word vectors.
total embedded: 0 common words
```

```
embedding_matrix = loadEmbeddingMatrix('glove')
```

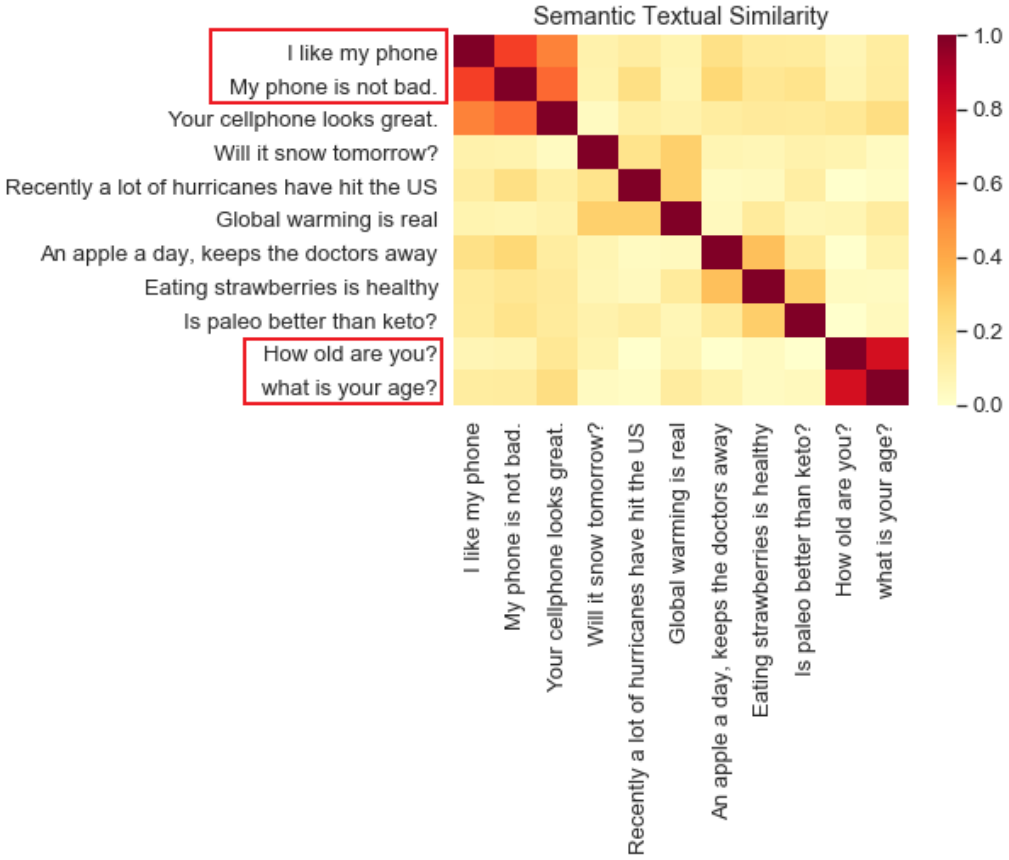
```
Loaded 1175734 word vectors.
total embedded: 79113 common words
```

```
embedding_matrix = loadEmbeddingMatrix('fasttext')
```

```
Loaded 110995 word vectors.
total embedded: 59312 common words
```

iii) USE (Universal Sentence Encoder):

USE is a sentence encoder, the advantage of it is that it can transform a whole sentence into embeddings instead of transform them word by word. It means that even if there are totally no same words in two sentences, USE can "understand" the sentences and find the similarity between them: For example, in **"how old are you"** and **"what is your age"**, although there is no same word, by plot the **cosine similarity** between their **USE embeddings**, we can find that they are really close:



Besides, another advantage of USE is that it has **512 dimensions** (the vectors it transforms to is 512d), which means it can be very detailed and accurate:

	0	1	2	3	4	5	6	7	8	9	...
0	-0.057829	-0.034936	0.027102	-0.030281	-0.036649	-0.028683	-0.028100	0.009702	-0.062288	-0.075045	...
1	-0.072239	-0.000845	-0.072422	0.077013	0.004003	-0.051323	-0.031766	0.034427	0.022856	0.031030	...
2	0.003207	-0.051132	0.010560	-0.016915	0.013706	0.037521	0.029077	0.022742	0.075265	0.024829	...
3	-0.037711	-0.079591	0.034021	-0.028984	-0.027576	-0.055188	0.073068	0.067444	-0.018094	-0.009031	...
4	-0.005501	-0.013888	-0.007009	-0.007767	-0.076502	0.020703	-0.013790	0.035132	0.054153	-0.011901	...
...	...	...	...	...	...	...	...	...	...	...	...
159566	0.001070	-0.055367	-0.034541	0.024528	0.065058	-0.057712	0.037719	-0.015136	0.028250	0.006690	...
159567	-0.042286	0.041825	-0.017267	-0.023695	-0.005338	0.058423	-0.079614	0.025481	0.000808	0.053419	...
159568	0.034068	-0.037034	-0.031748	-0.028014	0.030389	-0.017949	0.073730	0.013601	0.059243	-0.082300	...
159569	-0.044925	-0.000159	-0.020958	-0.017049	-0.048746	0.050928	-0.018042	0.093370	0.090315	-0.002193	...
159570	-0.012506	0.001415	0.022212	-0.031086	-0.051902	0.024268	0.006591	-0.007948	-0.084692	0.021051	...

159571 rows × 512 columns

In our project, we use "**tensorflow_hub**" package to load the USE model:

```
import tensorflow_hub as hub
model = hub.KerasLayer('universal-sentence-encoder_4')
def embed(input):    # function for embedding
    return model(input)
```

In **"USE_Embeddings.ipynb"**, you can find the detailed procedure of how we transformed our data into USE embeddings:

```
# transfrom sentences to embeddings: (training set)
embeddings_train = []
count = 0
for i in list_sentences_train:
    embedding = embed([i])
    embedding = np.reshape(np.array(embedding), (512,))
    embeddings_train.append(embedding)
    count += 1
    if count % 1000 == 0:
        print(count)
```

```
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
```

You can get more detailed information or download the model at: TensorFlow Hub (tfhub.dev).

6. Implementation of Models:

1) Logistic Regression (implemented by ourselves):

We implemented the Logistic Regression model by ourselves.

The procedure is to calculate the **cost function**, and then do the **gradient decent**:

```python
def sigmoid(z):
    temp = 1+np.exp(-z)
    result = 1/temp
    return result
```

```python
def cost(theta, X, y):
    hx = sigmoid(np.dot(X, theta))
    first = y * np.log(hx)
    second = (1 - y) * np.log(1 - hx)
    return -np.sum(first +second) / len(X)
```

```python
def gradient(theta, X, y):
    theta_x = np.dot(theta, X.transpose())
    h_x = sigmoid(theta_x)
    grad = (1/len(y))*np.dot(h_x-y.transpose(), X)
    return grad
```

```python
# add 1:
X_t.insert(0, 'Ones', 1)
# initialize theta (0 initialization):
theta = np.array(np.zeros(513))
X_t.shape, theta.shape, y_toxic.shape
```

```
((159571, 513), (513,), (159571,))
```

```python
#from sklearn.model_selection import t
#X_train, X_val, y_train, y_val = trai
```

```python
#cost(theta, X_t, y_toxic)
cost(theta, X_t, y_toxic)
```

```
0.6931471805599453
```

Because Logistic Regression only support **binary classification**, but our problem is a **muti-classification (6 labels)** problem, so we need to do the training and classification 6 times for each class. Each time the problem will be **1: "in that class" V.S. 0: "not in that class" / "in other classes":**

```python
# train for y_toxic:
result1 = opt.minimize(fun=cost, x0=theta, args = (X_t, y_toxic), method='TNC', jac = gradient)
result1
```

```
     fun: 0.12574192690669628
     jac: array([-7.02967719e-07,  2.16744348e-08, -3.80921940e-09, -1.85070905e-08,
       -4.79943605e-09, -7.91686408e-09, -3.49754246e-08, -4.56822846e-09,
        5.01661247e-09, -9.31292218e-09, -3.79710448e-08, -1.68361433e-08,
       -3.45578285e-08, -4.54785173e-09,  2.65888486e-08, -1.79226387e-08,
        1.62930858e-07, -6.86932082e-09, -1.74772668e-10,  5.85064243e-09,
        9.25086046e-09,  4.20960960e-08, -1.02213127e-08,  1.15326227e-08,
       -6.39568264e-09, -1.78548719e-08, -4.00729888e-09,  7.88242961e-09,
        2.37327960e-08,  7.70011305e-09, -4.55206358e-09,  2.13150572e-08,
        1.59102776e-09, -5.42986705e-10,  3.17620588e-08, -2.28566474e-08,
       -2.09645845e-08, -2.30463246e-08, -1.93828385e-08,  1.13992915e-08,
       -1.47583880e-08,  5.45515538e-09,  6.41861448e-08,  1.21861273e-08,
       -5.74679190e-08, -2.23209401e-08, -8.39360735e-09,  8.96548234e-08,
        2.27063243e-08, -1.24126480e-08,  1.02284267e-08, -2.82821353e-09,
       -1.66845123e-08, -8.98268531e-09,  4.97566489e-08,  1.26473458e-08,
        1.57978593e-08, -4.91435630e-08, -1.12155798e-08, -5.90563235e-08,
        2.89616029e-08, -2.84727886e-08,  2.49979339e-08,  1.43432034e-08,
        9.96342526e-08,  2.02389465e-09,  1.04237846e-08,  4.37014403e-08,
       -2.82268824e-08, -3.01156409e-08,  1.45632741e-09,  2.09912619e-08,
```

```python
# train for y_severe_toxic:
result2 = opt.minimize(fun=cost, x0=theta, args = (X_t, y_severe_toxic), method='TNC', jac = gradient)
result2
```

```python
# train for y_obscene:
result3 = opt.minimize(fun=cost, x0=theta, args = (X_t, y_obscene), method='TNC', jac = gradient)
result3
```

```python
# train for y_threat:
result4 = opt.minimize(fun=cost, x0=theta, args = (X_t, y_threat), method='TNC', jac = gradient)
result4
```

```
# train for y_insult:
result5 = opt.minimize(fun=cost, x0=theta, args = (X_t, y_insult), method='TNC', jac = gradient)
result5
```

```
# train for y_identity_hate:
result6 = opt.minimize(fun=cost, x0=theta, args = (X_t, y_identity_hate), method='TNC', jac = gradient)
result6
```

Finally, we calculated the mean **AUC-ROC score** for the model's prediction in these 6 times:

Using **USE**:

```
print(np.mean(scores_roc_aoc2))# Comput
```

```
AUC for the test set
#########################
Class: toxic
Test ROC AUC Score: 0.941926635863739
Class: severe_toxic
Test ROC AUC Score: 0.9795155645667679
Class: obscene
Test ROC AUC Score: 0.9511651273853227
Class: threat
Test ROC AUC Score: 0.9768433463742445
Class: insult
Test ROC AUC Score: 0.9139639809162889
Class: identity_hate
Test ROC AUC Score: 0.9763921024374702
0.9433011262573055
```

Using **TFIDF**:

```
print(np.mean(scores_roc_aoc))# Comput
```

```
Class: toxic
Test ROC AUC Score: 0.8417463006532745

Class: severe_toxic
Test ROC AUC Score: 0.6809342308240435

Class: obscene
Test ROC AUC Score: 0.823401212848411

Class: threat
Test ROC AUC Score: 0.6367507462186276

Class: insult
Test ROC AUC Score: 0.7690327793527002

Class: identity_hate
Test ROC AUC Score: 0.6410308739237967

0.7321493573034755
```

From the scores, we can find that **USE** works much better than **TFIDF** in Logistic Regression.

2) BP Neural Network (implemented by ourselves):

We also implemented the Backpropagation Neural Network by ourselves.

We set the number of layers to **3 (1 hidden layer)**, and all the activation functions to **Sigmoid function**:

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

```
def forward_propagate(X, theta1, theta2):
    m = X.shape[0]

    a1 = np.insert(X, 0, values=np.ones(m), axis=1)
    z2 = a1 * theta1.T
    a2 = np.insert(sigmoid(z2), 0, values=np.ones(m), axis=1) #insert new bias
    z3 = a2 * theta2.T
    h = sigmoid(z3)

    return a1, z2, a2, z3, h
```

And initialize all theta to **(-0.25, 0.25)**:

```
# the theta parameter vector
params = (np.random.random(size=hidden_size * (input_size + 1) + num_labels * (hidden_size + 1)) - 0.5) * 0.25
```

And use **"TNC"** method to train the model:

```
from scipy.optimize import minimize
fmin = minimize(fun=backprop, x0=params, args=(X, y),
                method='TNC', jac=True, options={'maxiter': 100})
fmin
```

```
     fun: 0.3578160532488686 7
     jac: array([-4.01852853e-04,  6.41655214e-06,  3.26822965e-06, ...,
        5.97044596e-04,  4.64399673e-04,  1.05502046e-03])
 message: 'Max. number of function evaluations reached'
    nfev: 101
     nit: 12
  status: 3
 success: False
       x: array([-0.38216075,  0.05357219,  0.27107219, ..., -0.65317092,
       -0.37540756, -1.25353737])
```

The AUC-ROC score is around **0.95**:

```
print('Test ROC AUC Score:', test_roc_auc)
```

```
Test ROC AUC Score: 0.9586175878003429
```

We have also tried **ReLU** activation functions:

```
def ReLU(x):
    return np.maximum(x, 0)
```

```
def ReLU_gradient(x):
    # ReLu 导数
    x[x <= 0] = 0
    x[x > 0] = 1

    return x
```

```
def forward_propagate(X, theta1, theta2):
    m = X.shape[0]

    a1 = np.insert(X, 0, values=np.ones(m), axis=1)
    z2 = a1 * theta1.T
    a2 = np.insert(ReLU(z2), 0, values=np.ones(m), axis=1) #insert new bias
    z3 = a2 * theta2.T
    h = ReLU(z3)

    return a1, z2, a2, z3, h
```

And used the **He initialization** for theta:

```
# He initialization:
params = np.random.random(size=hidden_size * (input_size + 1) + num_labels * (hidden_size + 1))  * np.sqrt(2 / input_size)
```

But the AUC-ROC score was only around **0.68:**

```
print('Train ROC AUC Score:', train_roc_auc)
```

```
Train ROC AUC Score: 0.6873656614730l8
```

So we finally chose to use **Sigmoid function** as activation function in BP NN.

3) LSTM (Long Short-Term Memory):

- **RNN (Recurrent Neural Networks)**:

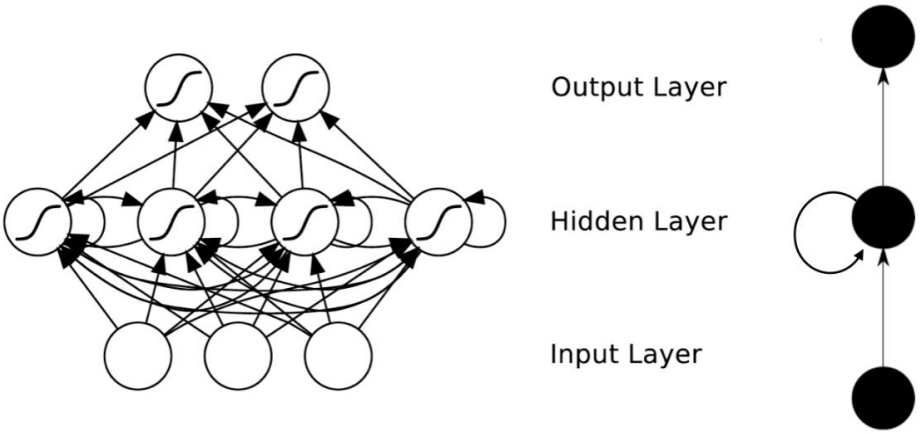
  Developed for the processing of **sequential data**. RNNs use their **hidden states** as memory to remember previously processed data. This makes RNN suitable for **long sequential data**, such as **NLP**.

  RNN Considers the influence of the previous input but not the other momentary inputs, for example, simple recognition of individual objects such as cats, dogs, handwritten numbers, etc. has good results. However, these algorithms do not perform as well for some time-dependent tasks, such as predicting the next moment of a video or the content of a document.

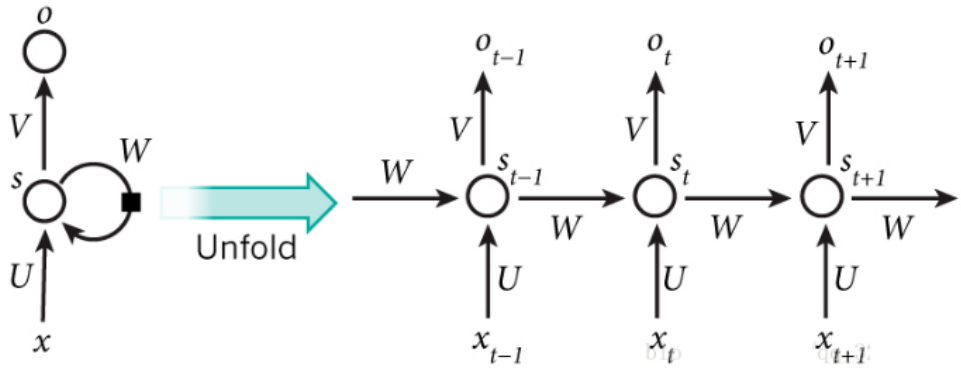
  The idea that **"human cognition is based on past experience and**

memory" is proposed. It differs from DNNs and CNNs in that it not only takes into account the previous moment's input, but also gives the network a 'memory' function for the previous content.

RNN is called a recurrent neural network because the current output of a sequence is also related to the previous output. This is expressed in the form that the network remembers the previous information and applies it to the computation of the current output, i.e., the nodes between the hidden layers are no longer unconnected but connected, and the input of the hidden layers includes not only the output of the input layer but also the output of the hidden layer at the previous moment.



We can unfold the **hidden layer** of RNN:



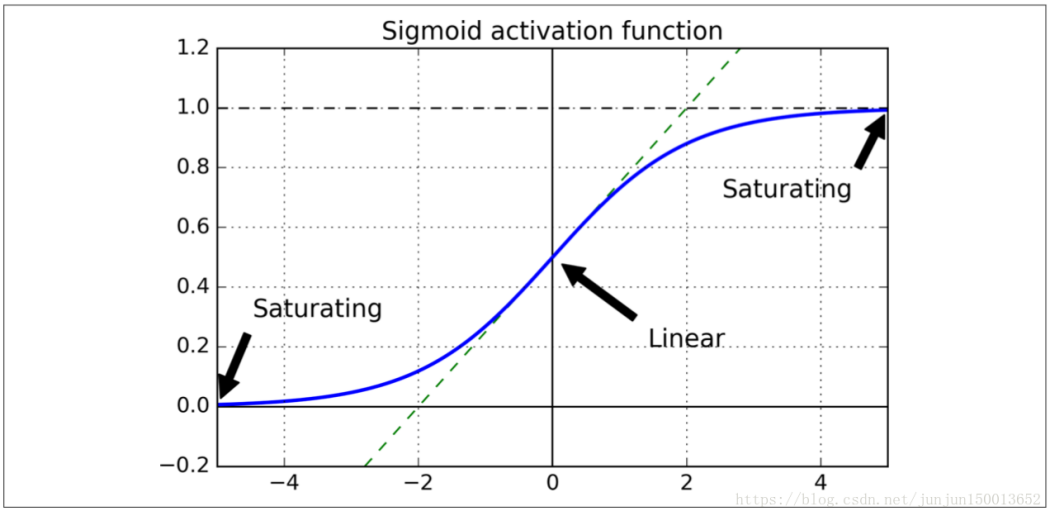
The calculation in hidden layer is:

$$\boldsymbol{h_t = Ux_t + Ws_{t-1}}$$

Where x_t is the **current input**, s_{t-1} is the summary of **past memory**. In that case, it can **"remember"** the previous information and applies it to the computation of the current layer.

- **Gradient explosion/disappearance:**

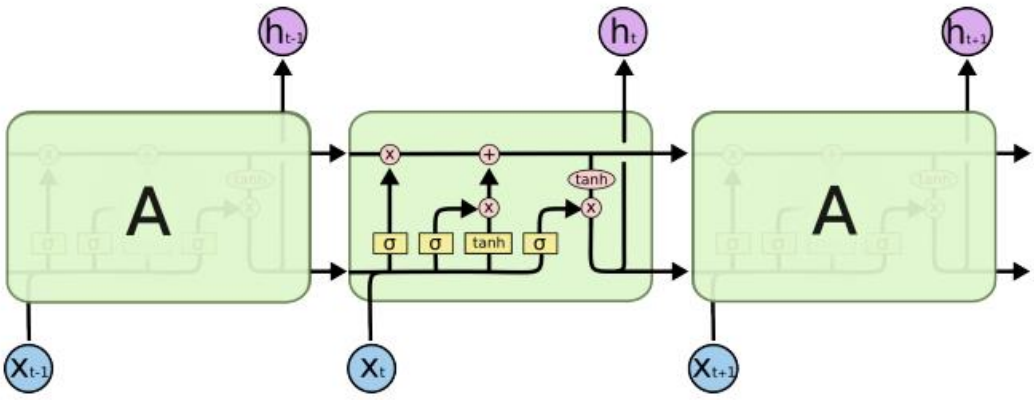
  However, there is a **problem** that in **RNN**, if the **gradient** between each step is too **large/small** (or using an improper activation function such as sigmoid), the gradient will become **too large or 0.**



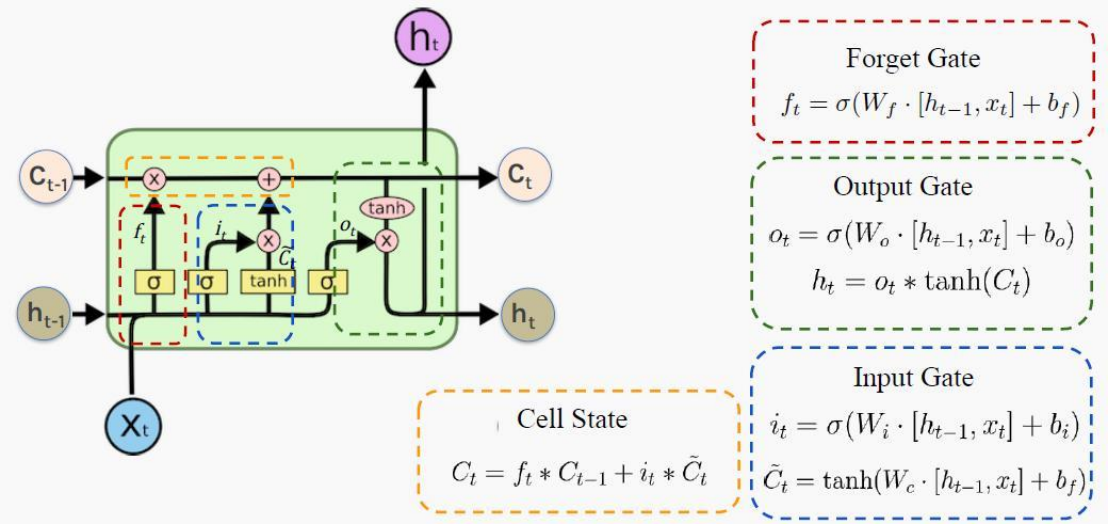
So we use **LSTM** to solve this problem.

- **LSTM (Long Short-Term Memory):**

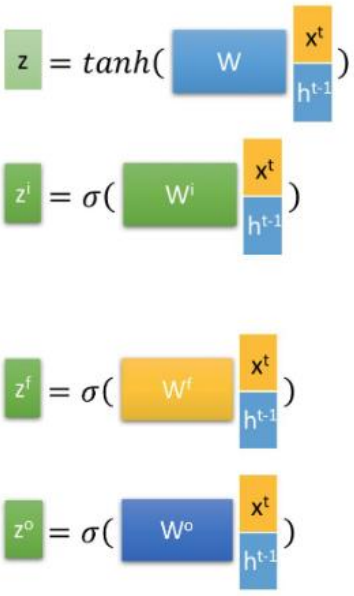
  LSTMs are special RNNs, they are designed to capture **long term dependencies**. The recurring units of the LSTM store **cell states (Ct)** along with the **hidden states**.



In LSTM, the previous **"memories"** are stored through special **gate** structures:



The **4 gates** are formed by **1 sigmoid function** and **3 *tanh* functions**:



To implement this model, we firstly imported the 3 pretrained **embeddings**:

```
def loadEmbeddingMatrix(typeToLoad):    #to load all 3 types of embedding matrix
        if(typeToLoad=="glove"):
            EMBEDDING_FILE=gl_path
            embed_size = 25
        elif(typeToLoad=="word2vec"):
            word2vecDict = word2vec.KeyedVectors.load_word2vec_format(wv_path, binary=True)
            embed_size = 300
        elif(typeToLoad=="fasttext"):
            EMBEDDING_FILE=ft_path
            embed_size = 300

        if(typeToLoad=="glove" or typeToLoad=="fasttext" ):
            embeddings_index = dict()
            f = open(EMBEDDING_FILE, encoding='gb18030', errors='ignore')
            for line in f:
                values = line.split()
                word = values[0]
                coefs = np.asarray(values[1:], dtype='float32')
                if len(coefs) != 25:
                  continue
                embeddings_index[word] = coefs

            f.close()
            print('Loaded %s word vectors.' % len(embeddings_index))
        else:
            embeddings_index = dict()
            for word in word2vecDict.wv.vocab:
                embeddings_index[word] = word2vecDict.word_vec(word)
            print('Loaded %s word vectors.' % len(embeddings_index))

        gc.collect()
        all_embs = np.stack(list(embeddings_index.values()))
        emb_mean,emb_std = all_embs.mean(), all_embs.std()

        nb_words = len(tokenizer.word_index)
        embedding_matrix = np.random.normal(emb_mean, emb_std, (nb_words, embed_size))
        gc.collect()

        embeddedCount = 0
        for word, i in tokenizer.word_index.items():
            i-=1
            embedding_vector = embeddings_index.get(word)
            if embedding_vector is not None:
                embedding_matrix[i] = embedding_vector
                embeddedCount+=1
        print('total embedded:',embeddedCount,'common words')

        del(embeddings_index)
        gc.collect()

        return embedding_matrix
```

And then, we import these **pretrained embeddings** (obtained by unsupervised training of a model on a large corpora):

Glove:

```
embedding_matrix = loadEmbeddingMatrix('glove')     #load 'glove' embedding matrix
```

```
Loaded 1175734 word vectors.
total embedded: 79113 common words
```

Word2vec:

```
embedding_matrix = loadEmbeddingMatrix('word2vec')
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_la
4.0.0, use self instead).

Loaded 306943 word vectors.
total embedded: 0 common words
```

Fasttext:

```
embedding_matrix = loadEmbeddingMatrix('fasttext')
```

```
Loaded 110995 word vectors.
total embedded: 59312 common words
```

Then, we use the LSTM model in **"keras"** package in Python:

```
from keras.layers import Dense, Input, LSTM, Embedding, Dropout, Activation
from keras.layers import Bidirectional, GlobalMaxPool1D,Bidirectional
from keras.models import Model
```

```
inp = Input(shape=(maxlen, ))#Define a keras Input tensor  #Start here, configure the model
```

```
x = Embedding(len(tokenizer.word_index), embedding_matrix.shape[1],weights=[embedding_matrix],trainable=False)(inp)    #Convert a positive int
x = Bidirectional(LSTM(60, return_sequences=True,name='lstm_layer',dropout=0.1,recurrent_dropout=0.1))(x)    #Realize the bidirectional struc
```

```
#config the output of the model:
x = GlobalMaxPool1D()(x)
x = Dropout(0.1)(x)
x = Dense(50, activation="relu")(x)
x = Dropout(0.1)(x)
x = Dense(6, activation="sigmoid")(x)
```

```
import keras.metrics as metrics    #Neural Network Evaluation Methods
```

```
model = Model(inputs=inp, outputs=x)    #Define the model here (have not entered the training set for training yet) (end configuring the model)
model.compile(loss='binary_crossentropy',
                optimizer='adam',
                metrics=[    #Use MSE and AUC as evaluation indicators
                        metrics.MeanSquaredError(),
                        metrics.AUC(),
                ])
```

Next, we use two techniques to config the model:

- **Embedding:**

  Convert a sparse positive integer (index value) to a dense vector of fixed size.

- **Bidirectional**:

Unidirectional LSTM does not take into account data further in sequence while predicting on current data. Bi-directional LSTM overcomes this shortage.

Finally, we got the **AUC-ROC scores**:

Glove:

```
preds_test = model.predict(X_test)    #predict using the test's x
```

```
print(roc_auc_score(y_test, preds_test))    #test accuracy for 'glove'
```

```
0.8731585814602864
```

Word2vec:

```
preds_test = model.predict(X_test)
```

```
print(roc_auc_score(y_test, preds_test))    #test accuracy for 'word2vec'
```

```
0.8564973871595932
```

Fasttext:

```
preds_test = model.predict(X_test)
```

```
print(roc_auc_score(y_test, preds_test))    #test accuracy for 'fasttext'
```

```
0.9597581026908394
```

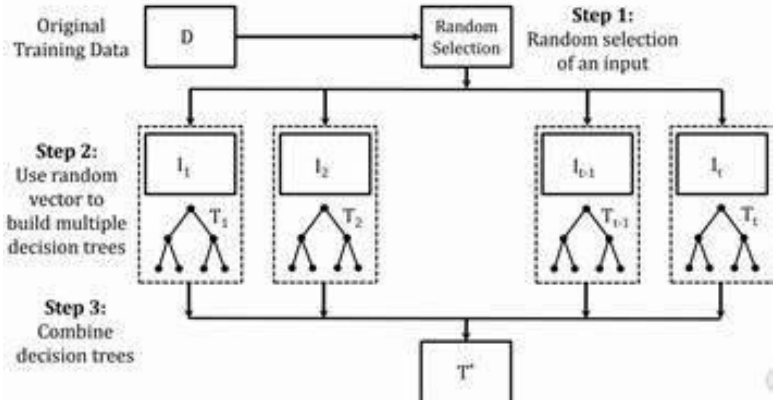
We can find that the best one is with "**fasttext**", which is around **0.95**.

4) **Ensemble Learning:**

We also used two models for **ensemble learning**, which can turn weak learners into a better classifier:

i) Bagging: Extra-Trees:

ET or **Extra-Trees (Extremely Randomized Trees)** algorithms are extension of Bagging and are very similar to random forest algorithms. They are composed of many decision Trees. ET randomness refers to characteristic randomness, parameter randomness, model randomness and split randomness.



Extra Tree is a **variant of Random Forest** with only the following differences:

- For the training set of each **decision tree**, **Random Forest** uses **random sampling bootstrap** to select the sampling set as the training set of each decision tree, while **Extra Tree** generally does not use random sampling, that is, each decision tree uses the **original training set**. Random Forest applies Bagging model, and extra-tree uses all samples, but **features are randomly selected**. Because splitting is Random, the results obtained by Random Forest are better than those obtained by Random Forest to some extent.
- After selecting partition features, the decision tree of **Random Forest** will select an **optimal partition point** of features based on principles such as **Gini coefficient**. This is the same as a traditional decision tree. However, **Extra Tree** is more radical and **randomly selects a feature** to divide the decision tree.

As can be seen from the second point, the scale of the generated decision tree is generally larger than that generated by RF because the partition point of features is randomly selected instead of the most advantageous bit. In other words, the variance of the model decreases further relative to RF, but the bias increases further relative to RF. In some cases, Extra Tree generalizes better than RF.

Random forest is to get the best bifurcation attribute in a random subset, while ET is completely random to get the bifurcation value, so as to realize the bifurcation of decision tree. **Extra Trees** will directly adopt all the samples and randomly select K features. For these K features, 1 split node is randomly selected for each feature to obtain K classification nodes. Then calculate the score of the K split nodes and select the node with the highest score as the split node.

In our project, we used the **ExtraTreesClassifier** in **sklearn** to implement it:

```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import ExtraTreesClassifier
```

```
losses = []
predictions = []
for class_name in class_names:
    train_target = train[class_name]
    classifier = ExtraTreesClassifier(n_estimators=30)
    score = np.mean(cross_val_score(classifier, comments_train,
                                    train_target, cv=2, scoring='roc_auc'
    print(class_name, score)
    classifier.fit(comments_train, train_target)
    predictions.append(classifier.predict_proba(comments_test)[:, 1])
```

```
toxic 0.9490797066944688
severe_toxic 0.9351889871049938
obscene 0.9719671393144169
threat 0.8609603094707186
insult 0.9545448797702478
identity_hate 0.8906286554177477
```

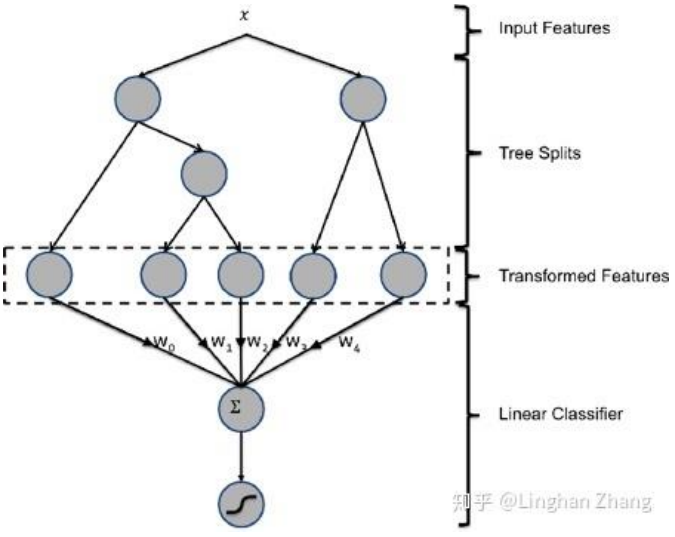
And finally, we can get the mean **AUC-ROC**, which is around **0.93**:

```
scores=[]
from sklearn.metrics import roc_auc_score
for i in range(6):
    scores.append(roc_auc_score(labels_consider[:,i],preds_consider[i,:]))
np.mean(scores)
```

```
0.9365694497823446
```

ii) Boosting: XGBoost:

XGBoost (eXtreme Gradient Boosting) is an algorithm based on **GBDT**. **GBDT** is an additive model based on Boosting inheritance idea, and the forward distribution algorithm is used for greedy learning in training. Each iteration learns a **CART** tree to fit the residual between the predicted result of the previous T-1 tree and the true value of the training sample.



There are some differences between **Ada-boost** and **XGBoost**.

- The first difference is that the **base learner** used by **XGBoost** can only be **decision tree**, whereas **Ada-Boost** can use a variety of **other weak learners**.
- The second difference is that the objective function of XGBoost has

been changed. The objective function consists of loss function and **regularization term**, which describes the complexity of the decision tree. The objective function of Ada-Boost has no regularization term.

There are **3 steps** to optimize the **XGBoost** target function.

- The first step: second order Taylor expansion, remove the constant term, optimize the loss function term;

$$l(y_i, x) \approx l(y_i, x_0) + l'(y_i, x_0)(x - x_0) + \frac{l''(y_i, x_0)}{2}(x - x_0)^2$$

- The second step: regularization term expansion, remove constant term, optimize regularization term;
- The third step: merge the coefficient of the first term and the coefficient of the second term to obtain the final objective function.

In our project, we used the **xgboost** in **sklearn** to implement it:

```
import xgboost as xgb
import gc
```

```
def XGBoost(train_X, train_y, test_X, test_y=None, feature_names=None):
    dic = {}
    num = 100
    list_dic = list(dic.items())

    xgtrain = xgb.DMatrix(train_X, label=train_y)
    xgtest = xgb.DMatrix(test_X, label=test_y)

    model = xgb.train(list_dic, xgtrain, num
                      , [ (xgtrain,'train'), (xgtest, 'test') ]
                      , early_stopping_rounds=10)

    return model
```

And finally, we can get the mean **AUC-ROC**, which is around **0.96**:

```
from sklearn.metrics import roc_auc_score
scores=[]
for i in range(6):
    scores.append(roc_auc_score(labels_consi
np.mean(scores)
```

```
0.9639784442207091
```

5) **SVM**:

We have also used the **SVM** model (**SVC** in **sklearn**) to see the AUC-ROC score:

```
from skmultilearn.problem_transform import ClassifierChain
from sklearn.svm import SVC
```

```
model_cc = ClassifierChain(classifier = SVC(), require_dense = [False, True])
model_cc.fit(X_train_feat, y_train)
```

```
ClassifierChain(classifier=SVC(), require_dense=[False, True])
```

```
preds_train = model_cc.predict(X_train_feat)
```

```
from sklearn.metrics import roc_auc_score, accuracy_score

roc_auc_score(labels_consider, preds_test.toarray())
```

```
0.7645674561502386
```

7. Results Analysis

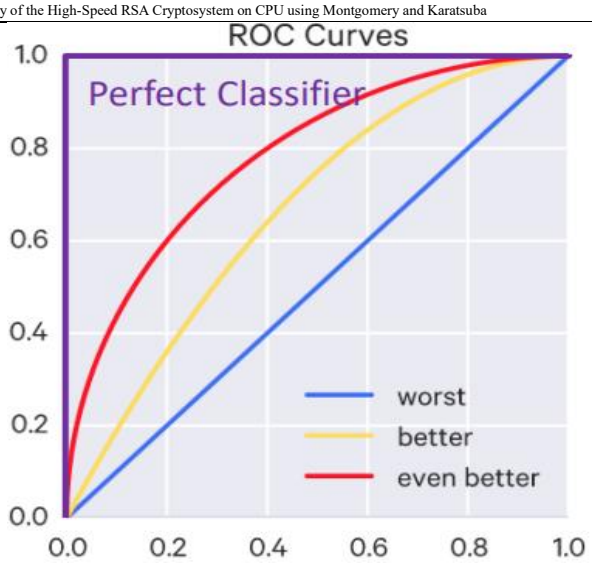
Because our results are not just simple **"1"** and **"0"**, but are probabilities:

```
preds_output.tail(10)
```

	0	1	2	3	4	5
90	0.056815	0.002836	0.020984	0.001009	0.024091	0.005074
91	0.059738	0.003667	0.024451	0.000888	0.025636	0.005367
92	0.103977	0.006066	0.031953	0.001133	0.035427	0.006543
93	0.086181	0.002883	0.028202	0.001251	0.033409	0.006100
94	0.032829	0.001784	0.012792	0.000670	0.012361	0.003595
95	0.055072	0.001483	0.011606	0.000712	0.012714	0.003028
96	0.028111	0.001634	0.010181	0.000774	0.011977	0.002618
97	0.061814	0.004361	0.022250	0.000921	0.022836	0.004960
98	0.026953	0.000888	0.011009	0.000417	0.009987	0.002093
99	0.149896	0.002675	0.028619	0.001337	0.019782	0.005681

It's not appropriate to transform them into **"1"** and **"0"**, so we decided to keep them as our results.

However, the simple "**accuracy**" works not well and not accurate enough for probabilities, so we decided to use **AUC-ROC score** to measure our models:



Finally, we calculated the AUC of all models, here we use a matrix to show AUC-ROC scores of the 6 models with 3 types of embeddings (because **word embeddings** are not easy to use for other models, only keras support it well, so we only use them for **LSTM**):

(Sorted by the highest score among 3 type of embeddings)

From the table, we can find that the best two models are **LSTM using Fasttext** and **XGBoost**.

Model/Embeddings	USE	TFIDF	Word Embeddings
XGBoost	**0.99**	**0.96**	----
LSTM (FastText)	----	----	**0.96**
Extra-Trees	**0.95**	**0.93**	----
BP NN	**0.95**	**0.77**	----
Logistic Regression	**0.94**	**0.73**	----
LSTM (Glove)	----	----	**0.88**
LSTM (Word2Vec)	----	----	**0.85**

From above, we can find that **XGBoost with USE** has the highest score, which is **0.99**, nearly perfect!

And the second ones are **XGBoost with TFIDF** and **LSTM with fastext**, which are both **0.96**.

However, although **USE** has a really high accuracy, it also has a serious problem, which is that **it took very long time to train** and get the embeddings of the texts, which can dramatically affect the **efficiency** of the model. Therefore, we did a comparison between the performance between **USE** and **TFIDF**:

- **USE**:

  Speed: **10s** for training 150,000 records (getting embeddings)

  Accuracy: **0.96** in XGBoost,

  0.73 in Logistic Regression

- **TFIDF**:

  Speed: **30min-1h** for training 150,000 records, need more time in training for Neural Networks or ensemble learning
  (Because it has **512 dimensions/features**)

  Accuracy: **0.99** in XGBoost,
  0.95 in Logistic Regression

From the comparison, we can summarize that **USE** definitely has a **high accuracy**, if we don't need to consider the time for training the model, we can choose to use that. However, in most time we need to **balance the speed and accuracy**, when **USE** takes too long for training, **XGBoost with TFIDF** already has an **accurate enough result (0.96)**, so we choose to use **XGBoost with TFIDF** as the final best model in our following study.

8. Application Using Crawled Data from Bilibili

In order to test our model is good or not, we planned to test it by recent comments coming from the Internet. After discussion, we decided to crawl data from the website: https://www.biligame.com/detail/?id=200.

Why we choosing this Game **"King of Honors"**? Because this game is indeed popular among young students in China, and also it is made by Tencent and thus, lots of bad comments may appear below its comments part. After we logged in website, we found it only rated at 2.7, and we endeavored to crawl data from this website.

And here is a part of our python crawling code:

```
app_id="200"    #王者荣耀的B站ID' 200 ', 每个游戏都有一个
page_total=50          #需要爬取的总页数

#准备容器
all_content=[]
all_grade=[]
all_publish_time=[]
all_up_count=[]
all_down_count=[]
all_user_name=[]
all_user_level=[]

print('************************************************')
print('Grapping comment data of app:{0} from BiliBili...'.format(app_id))
print('************************************************\n')

#step 2 爬取数据，循环爬取，每页获取10条评论数据
for pagenum in range(1,page_total+1):
    t1=tm.time()  #用于调试代码效率
    url="https://line1-h5-pc-api.biligame.com/game/comment/page?game_base_id={0}&rank_type=2&page_num={1}&page_size=10".format(app_id, pag
    comment_page = fetchURL(url)  #加载url
    parsered_comment=parserHtml(comment_page) #解析一页

    #从解析后的json对象中获取所需要的信息
    all_content.extend(parsered_comment[0])
    all_grade.extend(parsered_comment[1])
    all_publish_time.extend(parsered_comment[2])
    all_up_count.extend(parsered_comment[3])
    all_down_count.extend(parsered_comment[4])
    all_user_name.extend(parsered_comment[5])
    all_user_level.extend(parsered_comment[6])
    t2=tm.time()  #用于调试代码效率
    timing=t2-t1
    print('Page %d grapped, %5.2f seconds used' % (pagenum,timing))  #输出爬取进度
    if pagenum%10 == 0:
            print('Taking a break... To avoid being detected')
            tm.sleep(5)
```

```
    #从解析后的json对象中获取所需要的信息
    all_content.extend(parsered_comment[0])
    all_grade.extend(parsered_comment[1])
    all_publish_time.extend(parsered_comment[2])
    all_up_count.extend(parsered_comment[3])
    all_down_count.extend(parsered_comment[4])
    all_user_name.extend(parsered_comment[5])
    all_user_level.extend(parsered_comment[6])
    t2=tm.time()  #用于调试代码效率
    timing=t2-t1
    print('Page %d grapped, %5.2f seconds used' % (pagenum,timing))  #输出爬取进度
    if pagenum%10 == 0:
            print('Taking a break... To avoid being detected')
            tm.sleep(5)

#step 3 爬取完成，整理并导出数据
result={"content" : all_content,
    "grade" : all_grade,
    "publish_time" : all_publish_time,
    "up_count" : all_up_count,
    "down_count" : all_down_count,
    "user_name" : all_user_name,
    "user_level" : all_user_level,
    }

resultpd=pd.DataFrame(result)
print('\n************************************************')
print('Comment grapping finished. %d comments grapped in total' % (len(resultpd['content'])))
resultpd.to_excel('Bili_comment_appid{}.xlsx'.format(app_id))
print('Written to 【Bili_comment_appid{}.xlsx】'.format(app_id))
print('************************************************')
T2 = tm.time()
print('程序运行时间:%s秒' % ((T2 - T1)))
```

****

And the results:

```
Taking a break... To avoid being detected
Page 41 grapped,  0.43 seconds used
Page 42 grapped,  0.32 seconds used
Page 43 grapped,  0.34 seconds used
Page 44 grapped,  0.28 seconds used
Page 45 grapped,  0.32 seconds used
Page 46 grapped,  0.43 seconds used
Page 47 grapped,  0.32 seconds used
Page 48 grapped,  0.85 seconds used
Page 49 grapped,  1.74 seconds used
Page 50 grapped,  0.81 seconds used
Taking a break... To avoid being detected

**************************************************
Comment grapping finished. 500 comments grapped in total
Written to 【Bili_comment_appid200.xlsx】
**************************************************
程序运行时间:71.66904735565186秒
```

程序运行时间:71.66904735565186秒

In [16]:
```
resultpd.head(50)
```

Out[16]:

	content	grade	publish_time	up_count	down_count	user_name	user_level
0	虽然经常都在玩王者可是这个游戏真的太糟糕了根本不让我赢\n如果不是充了钱我玩都不玩	2	2021-12-12 21:44:36	0	0	硬币够了再改名	4
1	无话可说	2	2021-12-12 21:11:28	0	0	幻梦无枉	3
2	(･`▽･)	10	2021-12-12 21:09:55	0	0	一笑小白	3
3	*扎不多德勒！	2	2021-12-12 20:48:35	0	0	乌鱼子酱拌饭	5
4	垃圾游戏，我被安排了，给个一星不过分吧	2	2021-12-12 20:32:44	0	0	hwbbsuunw	5
5	非常好！	10	2021-12-12 20:27:38	0	0	卖个萌给爷看	4
6	垃圾游戏，一星都不想给！	2	2021-12-12	1	1	火石岩	4

And we put this output dataframe into excel, and translating the Chinese comments to English by excel command:

C2 =FILTERXML(WEBSERVICE("http://fanyi.youdao.com/translate?&i="&B2&"&doctype=xml&version="),"//translation")

	A	B	C	D	E
1		content		grade	publish_time
2	0	虽然经常都在玩王者可是这个游戏真的太糟糕了根本不让我赢如果不是充了钱我玩都不玩	Although often playing king but the	2	2021-12-12
3	1	无话可说	Have nothing to say	2	2021-12-12
4	2	(･`▽･)	(･`▽･)	10	2021-12-12
5	3	*扎不多德勒！	Mr Dodd not le!	2	2021-12-12
6	4	垃圾游戏，我被安排了，给个一星不过分吧	The garbage game, I was arranged to	2	2021-12-12
7	5	非常好！	Very good!	10	2021-12-12
8	6	垃圾游戏，一星都不想给！	The garbage game, don't want to give	2	2021-12-12
9	7	我去(⇀‸↼") 三次五连就出了荣耀水晶了，我近期是不是能吃就多吃点了，不然没机会了？？！	I went to (⇀ ‸ ↼ ") three times fiv	10	2021-12-12
10	8	嫦娥的翻牌都得翻全了才能到下一关，该不会只有我是这么倒霉的吧，我不信！！	The goddess of the moon's flip all t	8	2021-12-12
11	9	快过年了，是不是该得给我们瑶瑶公主来件新衣呢，不要多华丽的，类似嫦娥的那套也行。	The Chinese New Year is coming soon,	10	2021-12-12
12	10	傻卵游戏，傻卵玩家，什么**游戏，狗都不玩，**的一群**玩家	Silly eggs games, silly eggs players	2	2021-12-12
13	11	那个请问瑶和云中君的情侣皮什么时候考虑一下，我很想要，换蔡文姬和澜的也行，最近磕他们的糖磕上瘾了。	The couple is jun yao and the cloud	10	2021-12-12
14	12	垃圾游戏	The garbage game	2	2021-12-12
15	13	求你们不会玩嫦娥的玩家不要再玩嫦娥了，嫦娥这胜率现在都惨不忍睹了....	Beg you not to play the goddess of t	8	2021-12-12
16	14	凭实力得到你们家客服的关爱礼包，一天连跪十几局，我也是蛮厉害的，下次不要再送了啊	With strength to get your home care	10	2021-12-12
17	15	已经不知道多少了，毁去了多少孩子的童年，只是沉迷在虚拟的实力之中，可游戏再怎么强，他们现实中的实力也	Already don't know how many, how muc	2	2021-12-12
18	16	报复社会	Revenge society	10	2021-12-12
19	17	看到一条评论差点把我笑断气了，内容是：滴个声音好听的女孩子，段位别低于钻石，不然就玩不了排位，然后	See a comment almost put me smile di	10	2021-12-12

Sheet1

Finally, we put the English comment into model of XGBoost, which is one of the best evaluation models after we testing all of our models, and get the result:

```
In [127]: 1 pd.DataFrame(preds_consider)
```

Out[127]:

	0	1	2	3	4	5
0	0.058796	0.004888	0.021741	0.000921	0.024539	0.005419
1	0.742393	0.035874	0.940924	0.000454	0.309874	0.006232
2	0.121828	0.007203	0.037679	0.001133	0.035876	0.006386
3	0.059197	0.001628	0.016351	0.000846	0.013170	0.003497
4	0.103977	0.006066	0.031953	0.001133	0.035427	0.006543
...	...	...	...	...	...	...
95	0.055072	0.001483	0.011606	0.000712	0.012714	0.003028
96	0.028111	0.001634	0.010181	0.000774	0.011977	0.002618
97	0.061814	0.004361	0.022250	0.000921	0.022836	0.004960
98	0.026953	0.000888	0.011009	0.000417	0.009987	0.002093
99	0.149896	0.002675	0.028619	0.001337	0.019782	0.005681

100 rows × 6 columns

And you can see that the first row is the six descriptive adjectives, and we take 100 from the 500 comments to go through the **XGBoost with TFIDF** model.

Each element from this dataframe is a number standing for a **possibility** whether is belong to the relevant label or not.

Finally, we use that predicted result to compare with our crawled data with labels (we identified them by ourselves), and get the accuracy:

```
from sklearn.metrics import roc_auc_score
scores=[]
for i in range(6):
    scores.append(roc_auc_score(labels_consider[:,i],preds_consider[:,i]))
np.mean(scores)
```

```
0.9845326560839275
```

From the **AUC value**, we can know that our final model is very accurate to predict the negative comments in the real Internet virtual world.

9. Conclusion and Reflection

To achieve our goal, which is to implement a system that can process a comment which is in the form of natural language on the Internet, classify its main negative features in the 6 labels and recognize its inner emotion (whether it's a negative and hateful comment or not), we **surveyed on the existed techniques** in this field, **preprocessed** the data, implemented **Logistic Regression**, **BP NN** by ourselves, and used **Extra-Trees**, **XGBoost, SVM** and **LSTM** to **with 3 different NLP methods: TFIDF, pretrained word embeddings and USE,** experimented on our data set, and finally compared and **chose the best two models** to do the prediction on our **crawled data set** from Bilibili.

In this process we have come up with some **novel ideas** that are different from the **normal emotion recognition** to implement this system:

1) **Pretrained word vectors for Internet phrases:**

   Because our data set are comments from the Internet, there are lots of Internet phrases that are different from our daily language, we need to specify them to make a better prediction. The raw texts for training **fasttext**, **Glove** and **word2vec** are from **twitter**, **google** and other Internet sources, which can have a good performance on our dataset.

2) **Comparison between USE and TFIDF:**

   We didn't adopted **USE** because of its high accuracy, but **balanced the speed and accuracy** with TFIDF, and then chose the model with the **highest efficiency (accurate enough and fast)**.

3) **Six detailed labels to specify the emotion:**

   In our results, we have six labels, which are **toxic, severe_toxic, obscene, threat, insult, identity_hate** to analyze the detailed emotion in the comment, but not only the simple **"positive-1"** and **"negative-0"**. They can help us to understand and find the deep meaning of the bad languages and to deal with them better.

In further use, we can use those labels to classify the category of negative comments, and choose different methods to regulate the network environment, or even make a **statistical analysis** on the categories of **Internet violence**. If we **add more labels**, we can let the user to report or machine to identify those bad comments, then **remove them** or **block these accounts** as soon as possible.

In the future researches and development, we will work hard on those aspects to improve our system and try to use our data mining techniques to maintain a healthy and harmonious network environment.

10. Reference

Sundermeyer, M., Schlüter, R., & Ney, H. (2012). *LSTM neural networks for language modeling*. In Thirteenth annual conference of the international speech communication association.

Merity, S., Keskar, N. S., & Socher, R. (2017). *Regularizing and optimizing LSTM*

language models. arXiv preprint arXiv:1708.02182.

Zheng, H., Yuan, J., & Chen, L. (2017). *Short-term load forecasting using EMD-LSTM neural networks with a Xgboost algorithm for feature importance evaluation.* Energies, 10(8), 1168.

Zhou, Y., Li, T., Shi, J., & Qian, Z. (2019). *A CEEMDAN and XGBOOST-based approach to forecast crude oil prices*. Complexity, 2019.

Borzekowski, D. L., & Rickert, V. I. (2001). *Adolescent cybersurfing for health information: a new resource that crosses barriers*. Archives of pediatrics & adolescent medicine, 155(7), 813-817.

Kharwar, A. R., & Thakor, D. V. (2022). *An Ensemble Approach for Feature Selection and Classification in Intrusion Detection Using Extra-Tree Algorithm*. International Journal of Information Security and Privacy (IJISP), 16(1), 1-21.

Pregibon, D. (1981). *Logistic regression diagnostics*. The annals of statistics, 9(4), 705-724.

Midi, H., Sarkar, S. K., & Rana, S. (2010). *Collinearity diagnostics of binary logistic regression model*. Journal of Interdisciplinary Mathematics, 13(3), 253-267.

Li, H., Liu, H., & Zhang, Z. (2020). *Online persuasion of review emotional intensity: A text mining analysis of restaurant reviews*. International Journal of Hospitality Management, 89, 102558.