Toxic Content Classification in Django

Peihong Yu Ziran Gong Haoran Peng
Stevens Institute of Technology
pyu7@stevens.edu zgong5@stevens.edu hpeng3@stevens.edu

***Abstract*—In this project, we aim at finding out inappropriate comments from Quora website by building a binary classification model and apply the model to our website where you enter some questions and then the application will provide you with output to classify whether your words are TOXIC or NOT. We use word embedding method to map each text into corresponding data. Then we tried three different models and one combination method to train the model. The approaches we adopt to solve the problem are 'GRU', 'LSTM' and 'Attention' . We used the Django to build the AI application which including friendly interaction and beautiful interface. Finally, in the evaluation part, our accuracy reaches 0.70583. And the application can provide stable problem detection services**

I. Introduction

Quora is a website for people to share questions and answers with others. But sometimes inappropriate comments appear. Till now, the Quora has already implemented machine learning and hand-operated ways to decrease the possibility of insincere questions. But this is far from the goal. In order to combat insincere questions more efficiency, help Quora maintain their policy: "Be Nice, Be Respectful". We need to find more up-gradable ways to discover these ambiguous and confusing comments. Here is the link of this competition: https://www.kaggle.com/c/quora-insincere-questions-classification/data

The training dataset consists of three parts: id(numbers), question text(string), label(0 or 1) whereas test data lacking labels. By filling missing data, removing or replacing typos in an appropriate way, we can obtain a better result. Notice from the original data set, there are many disturbing characters. For the data pre-processing part, we majorly take four steps to clean the original data:

1) Obtain a vocabulary by loading words from word embedding(wiki-news-300d-1M.vec).
2) Get and analyze words in training data which are uncovered by "wiki" vocabulary.
3) Discard the punctuation, spaces and replace the misspell words into correct spell words.
4) Choose appropriate words in "wiki" vocabulary and replace the "cannot find" words with them.

II. Methods

A. LSTM

The defect of RNN is that with the growth of time period, it cannot get effect information from the time period a long time before shown as Fig.1. If we want to know more information with "t+1", we probably need to realize the meaning from "0" and "1" in the time process. Due to the long distance, the model learns from "0" and "1" cannot be expressed to the node with " t+1 ". As far as we can see, RNN can only memorize the short information sequence.

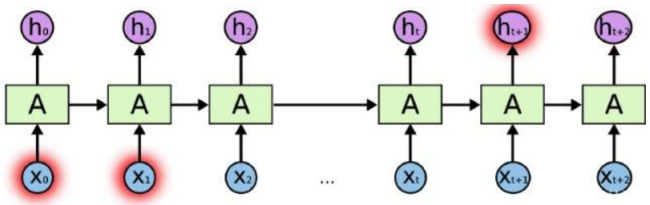


Fig. 1: RNN cannot get information with long period before

As a result LSTM networks came. LSTM is not a totally new module, it's a kind of evolution from RNN module shown as Fig.1. In order to keep and transfer information for a long period of time is the default behavior of these networks. Comparing with the structure, both LSTM and RNNs have a chain like shape. But the repeating module from two models are quite different. The original repeating module of RNNs only has a single tanh layer, but LSTM module contains four interacting layers.

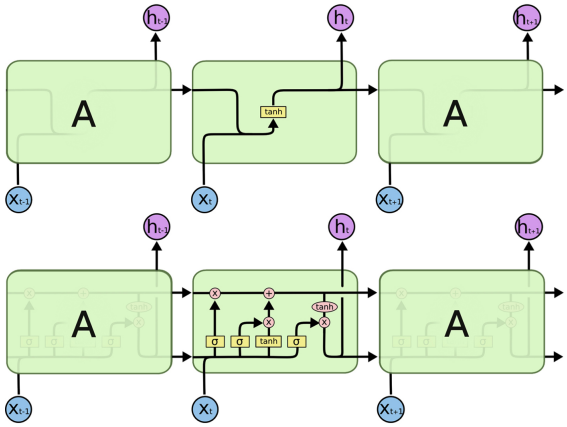


Fig. 2: RNNs module structure vs. LSTM module structure

The key point of LSTM is that it includes a ”coveyor belt” within the module structure. Information will transfer on this ”conveyor belt” and only few linear interaction can prevent the loss of information. There are three different gates collaborate together : ”Foget”, ”Refresh” and ”Output”.

B. GRU

When we train the LSTM module, we find that although the result comes from LSTM is much better than vallina RNN module, but the time cost is also higher. In that case, we notice that LSTM module has a variant module ”GRU”.

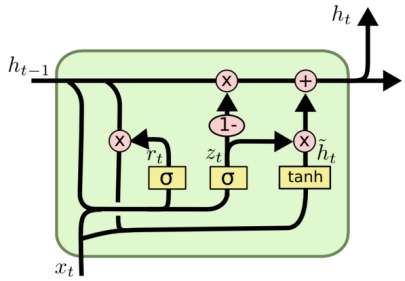


Fig. 3: GRU module

It’s obvious that inside of the GRU module, there are only two gates named ”Refresh” and ” Reset” instead of three gates in the LSTM module. The ”Refresh” gate decides how much previous information we need to go through and the ”Reset” gate decides how much former information the module should discard.
In theoretical, because of the decrease of parameters in the GRU module, the computational efficiency of GRU will higher than LSTM.

C. Attention

The attention mechanism was born to help memorize long source sentences in neural machine translation (NMT). Rather than building a single context vector out of the encoder’s last hidden state, the secret sauce invented by attention is to create shortcuts between the context vector and the entire source input. The weights of these shortcut connections are customizable for each output element. Here, we revise the attention mechanism to do binary classification instead.

III. Implementation

A. Data pre-processing

The first and most important part of our project is data preprocessing. We print the shape of training dataset and testing dataset on the first hand. In figure 1, the output shows that we have about 1306122 rows and 3 columns of data. The file train.csv has three columns: qid, question_text, target. ”qid” contains ids for every sentences which is useless for us. “Question_text” column is composed of sentences that we need to analysis and feed into our models. “Target” column has binary values. “1” means that this sentence is identified as insincere.

```
In [2]: train = pd.read_csv("quora-insincere-questions-classification/train.csv")
        test = pd.read_csv("quora-insincere-questions-classification/test.csv")
        print("Train shape : ",train.shape)
        print("Test shape : ",test.shape)

        Train shape :  (1306122, 3)
        Test shape :  (375806, 2)

In [20]: train[:2]

Out[20]:
```

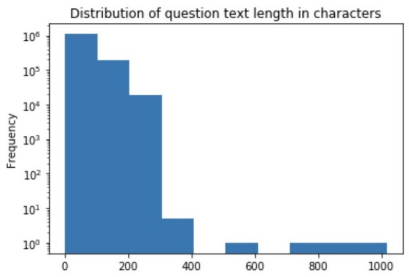
	qid	question_text	target
0	00002165364db923c7e6	How did Quebec nationalists see their province...	0
1	000032939017120e6e44	Do you have an adopted dog, how would you enco...	0

Fig. 4: Information of training dataset

By thoroughly looking at questions, we find two problems, one is the lengths of questions are not same, the other one is typos and new words may not appear in word embedding.

We generate two distributions of question text length in words and characters shown as Fig 5 below. The shapes of these two distributions are very similar. In order to decide how long should we set the model’s input, we then calculate the average length and max length in Fig 6.

```
In [21]: train['question_text'].apply(lambda x: len(x)).plot(kind='hist')
         plt.yscale('log')
         plt.title('Distribution of question text length in characters')
Out[21]: Text(0.5, 1.0, 'Distribution of question text length in characters')
```




```
In [22]: train['question_text'].apply(lambda x: len(x.split())).plot(kind='hist')
         plt.yscale('log')
         plt.title('Distribution of question text length in words')
Out[22]: Text(0.5, 1.0, 'Distribution of question text length in words')
```

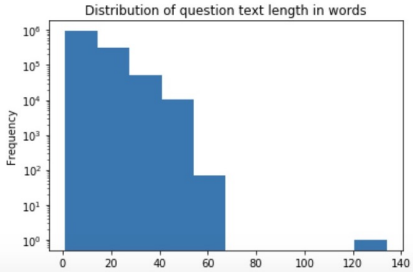


Fig. 5: Distribution of question text length in characters

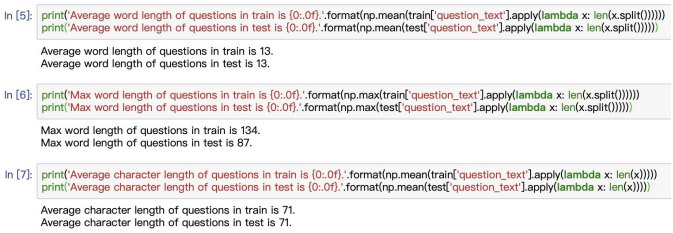


Fig. 6: Average length and max length of questions in training dataset and testing dataset

In the result above, the average character length in train and test are all 71, so we decide to set the maximum length of words in each sentence to 72.

For the purpose of dealing with typos and new words, the first thing we need to do is getting vocabularies for dataset and word embedding. To get the vocabulary for dataset, we build a function called build_vocab to collect not only words but also frequencies (Fig 7). To get the vocabulary for word embedding, we simply load from “wiki-news-300d-1M.vec” (Fig 8).

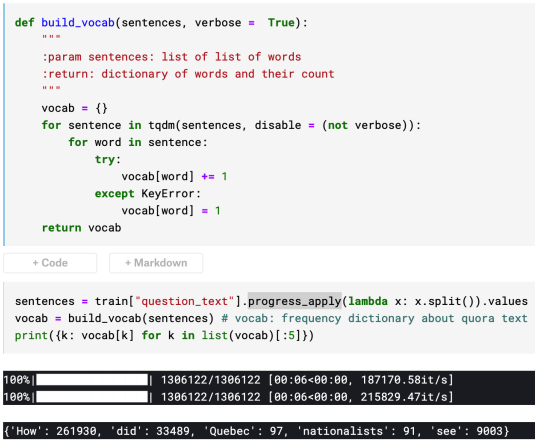


Fig. 7: Collect words and words frequency from training dataset

Fig. 8: Use KeyedVectors library to change words into one-dimensional vector

By using the check_coverage function we build, we get the uncovered sorted vocabulary called ”oov” (Fig 9). Fig 7 displays top 10 uncovered words, most of them are abbreviations and words with punctuations. After manually create a misspell dictionary containing abbreviations and common misspell words with their correct format (Fig 11), we simply remove other punctuations considering punctuations may not be very useful for classification (Fig 12). What’s more, we also pick 1000 words from ”oov” and compare each word among sentences in the training dataset to get the according index, label and frequency. (Fig.13)

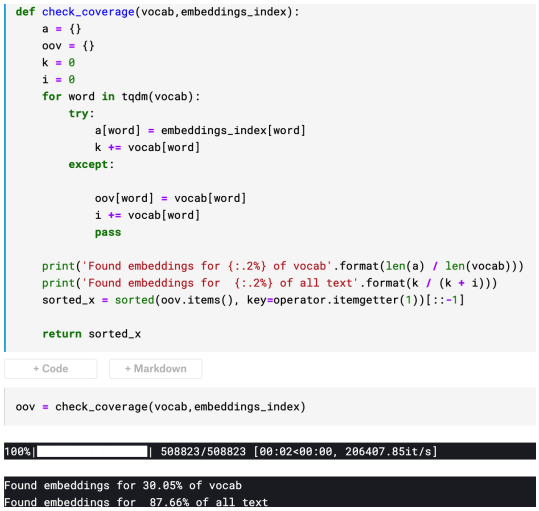


Fig. 9: Percentage of covered and uncovered words compare to the ”wiki” dataset

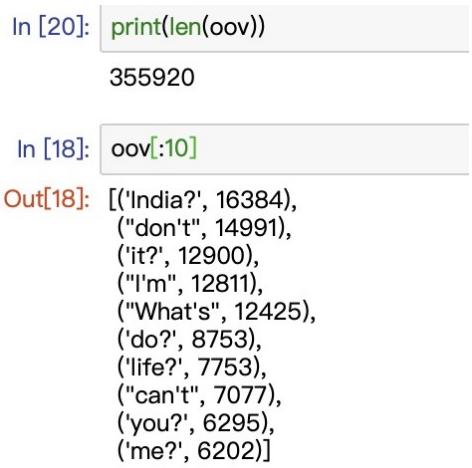


Fig. 10: Display top 10 frequency of uncovered words

Fig. 11: Manually create a misspell dictionary

```
In [17]:
# transfer this to kaggle code
def clean_text(x):
    x = str(x)
    for punct in '?!.,"#$%\'()*+-/:;<=>@[\\]^{|}~' + '“”’':
        x = x.replace(punct, ' ')
    for punct in '_`':
        x = x.replace(punct, ' ')
    return x

In [18]:
# transfer this to kaggle code
train["question_text"] = train["question_text"].progress_apply(lambda x: clean_text(x))
# leave codes below, don't transfer these
sentences = train["question_text"].apply(lambda x: x.split())
vocab = build_vocab(sentences)

100%|██████████| 1306122/1306122 [00:06<00:00, 211132.97it/s]
100%|██████████| 1306122/1306122 [00:03<00:00, 382303.74it/s]
```

Fig. 12: Split and replace punctuations

```
In [19]:
word = {}
for i in oov[:1000]:
    word[i[0]] = []
    num = 0
    for j in range(len(train['question_text'])):
        cnt = train['question_text'][j].count(i[0])
        num += cnt
        if cnt > 0:
            word[i[0]].append([j,train['target'][j],cnt])
        if num == i[1]:
            break
```

Fig. 13: Generate mistake embedding dictionary

When we ruled out the possibility of mistyping, the number of cannot-find-word remains high which may still affect the accuracy of our model. To eliminate this effect as much as possible, we come up with a method which replaces cannot-find-word with other words based on labels. We first sort the cannot-find-word with frequencies, and then choose the top 1000 separately and count how many times they appear in good_sentence(label 0) and bad_sentence(label 1) in train dataset as shown in Fig 14. And calculate the ratio=bad_times/total times (shown in Fig 15).

```
In [28]:
bad_word_replace['bhakts'] = 'Marathis'
bad_word_replace['Bhakts'] = 'Marathis'
bad_word_replace['Skripal']='Africans'
bad_word_replace['rohingya']='Africans'
bad_word_replace['Trumpers']='liberals'
bad_word_replace['Strzok']='leftist'
bad_word_replace['bhakt'] = 'Marathis'
bad_word_replace['Brexiters']='Africans'
bad_word_replace['wumao']='liberals'
bad_word_replace['Tamilans']='liberals'
bad_word_replace['Feku']='leftist'
bad_word_replace['nibiru']='liberals'
bad_word_replace['Sanghis']='muslims'
bad_word_replace['rohingyas']='Africans'
bad_word_replace['hutiya']='leftist'
bad_word_replace['Gujratis']='leftist'
bad_word_replace['Rejuvalex']='muslims'
bad_word_replace['Tambrahms']='liberals'
bad_word_replace['cucks']='muslims'
bad_word_replace['thighing']='liberals'
bad_word_replace['seziure']='leftist'
bad_word_replace['Pribumi']='leftist'
bad_word_replace['Ghazwa']='muslims'

In [29]:
train["question_text"] = train["question_text"].progress_apply(lambda x: correct_spelling(x, bad_word_replace))
sentences = train["question_text"].apply(lambda x: x.split())
vocab = build_vocab(sentences)

100%|██████████| 1306122/1306122 [00:06<00:00, 198423.52it/s]
100%|██████████| 1306122/1306122 [00:03<00:00, 372896.75it/s]

In [30]:
train["question_text"] = train["question_text"].progress_apply(lambda x: correct_spelling(x, verygood_word_replace))
sentences = train["question_text"].apply(lambda x: x.split())
vocab = build_vocab(sentences)

100%|██████████| 1306122/1306122 [03:00<00:00, 7255.28it/s]
100%|██████████| 1306122/1306122 [00:03<00:00, 382755.32it/s]
```

Fig. 14: Replace cannot-find-word with other words base on labels

```
In [23]:
c = 0
for i in data:
    bad_times = count_times(data[i])
    print(i)
    print('good_sentences: ',len(data[i]) - bad_times,'\tbad_sentences: ',bad_times, '\tbad/(good+bad): ',bad_times/len(data[i]))

    # comment these two lines to see the whole info
    c += 1
    if c == 5:
        break

Quorans
good_sentences:  629    bad_sentences:  205   bad/(good+bad):  0.24580335731414868
BITSAT
good_sentences:  545    bad_sentences:  0     bad/(good+bad):  0.0
COMEDK
good_sentences:  347    bad_sentences:  1     bad/(good+bad):  0.0028735632183908046
KVPY
good_sentences:  343    bad_sentences:  0     bad/(good+bad):  0.0
Quoran
good_sentences:  238    bad_sentences:  66    bad/(good+bad):  0.21710526315789475

In [24]:
len1 = 0
thre = 0.01
verygood_word_replace = {}
for i in data:
    bad_times = count_times(data[i])
    if bad_times/len(data[i])<thre:
        verygood_word_replace[i]='fresher'
        print(i)
        print('good_sentences: ',len(data[i]) - bad_times,'\tbad_sentences: ',bad_times, '\tbad/(good+bad): ',bad_times/len(data[i]))
        len1 += 1
print("\nnumbers of words which bad/total exceed threshold:",thre,'is:',len1)

BITSAT
good_sentences:  545    bad_sentences:  0     bad/(good+bad):  0.0
COMEDK
good_sentences:  347    bad_sentences:  1     bad/(good+bad):  0.0028735632183908046
KVPY
good_sentences:  343    bad_sentences:  0     bad/(good+bad):  0.0
WBJEE
good_sentences:  229    bad_sentences:  0     bad/(good+bad):  0.0
mtech
good_sentences:  216  bad_sentences:  0     bad/(good+bad):  0.0
articleship
good_sentences:  185  bad_sentences:  0     bad/(good+bad):  0.0
VITEEE
good_sentences:  181  bad_sentences:  1     bad/(good+bad):  0.005494505494505495
aadhar
good_sentences:  145  bad_sentences:  1     bad/(good+bad):  0.006849315068493315
marksheet
good_sentences:  130  bad_sentences:  0     bad/(good+bad):  0.0
UCEED
good_sentences:  121  bad_sentences:  0     bad/(good+bad):  0.0
```

```
In [25]:
len1 = 0
thre = 0.5
bad_word_replace={}
for i in data:
    bad_times = count_times(data[i])
    if bad_times/len(data[i])>thre:
        print(i)
        print('good_sentences: ',len(data[i]) - bad_times,'\tbad_sentences: ',bad_times, '\tbad/(good+bad): ',bad_times/len(data[i]))
        len1 += 1
print("\nnumbers of words which bad/total exceed threshold:",thre,'is:',len1)

bhakts
good_sentences:  11   bad_sentences:  52    bad/(good+bad):  0.8253968253968254
Bhakts
good_sentences:  11   bad_sentences:  38    bad/(good+bad):  0.7755102040816326
Skripal
good_sentences:  18   bad_sentences:  22    bad/(good+bad):  0.55
rohingya
good_sentences:  8    bad_sentences:  12    bad/(good+bad):  0.6
Trumpers
good_sentences:  6    bad_sentences:  14    bad/(good+bad):  0.7
Strzok
good_sentences:  6    bad_sentences:  11    bad/(good+bad):  0.6470588235294118
bhakt
good_sentences:  2    bad_sentences:  14    bad/(good+bad):  0.875
Brexiters
good_sentences:  6    bad_sentences:  7     bad/(good+bad):  0.5384615384615384
wumao
good_sentences:  3    bad_sentences:  9     bad/(good+bad):  0.75
Tamilans
good_sentences:  3    bad_sentences:  8     bad/(good+bad):  0.7272727272727273
Feku
good_sentences:  4    bad_sentences:  7     bad/(good+bad):  0.6363636363636364
nibiru
good_sentences:  2    bad_sentences:  6     bad/(good+bad):  0.75
Sanghis
good_sentences:  1    bad_sentences:  8     bad/(good+bad):  0.8888888888888888
```

Fig. 15: Calculate bad ratio of sentences within the total training dataset

At the same time, we randomly sample 1000 sentences respectively from good_sentence and bad_sentence. Collect words appear in these 2000 sentences and count times they appear in good_sentence(label 0) and bad_sentence(label 1) in train dataset. Then calculate the ratio of each word, shown in Fig 16. By setting the threshold of ratio, we can choose appropriate word with similar ratio as canno-find-word, we can replace cannot-find-word with them. For example, in Fig 16 the ratio of ‘bhakts’ is about 0.8254, and the word ‘Marathis’ has similar ratio value. As to this situation, it is theoretically feasible to replace ‘bhakts’ with ‘Marathis’

```
In [45]: for each in good_words:
             good_num = 0
             bad_num = 0
             for i in good_sentence.values():
                 good_num+=i[0].count(each)
             for i in bad_sentence.values():
                 bad_num+=i[0].count(each)
             if bad_num/(good_num+bad_num)<=0.001:
                 print(each, "\t",good_num,"\t", bad_num,"\t", bad_num/(good_num+bad_num))
```

```
cocci       7    0    0.0
Sputum      1    0    0.0
farro       1    0    0.0
ECE   1188       1    0.0008410428931875525
subah       2    0    0.0
sapphire    21   0    0.0
gemstone    50   0    0.0
Marquesa    4    0    0.0
Montemayor       1    0    0.0
JAGs        2    0    0.0
ohms        41   0    0.0
```

```
In [52]: for each in bad_words:
             good_num = 0
             bad_num = 0
             for i in good_sentence.values():
                 good_num+=i[0].count(each)
             for i in bad_sentence.values():
                 bad_num+=i[0].count(each)
             if bad_num/(good_num+bad_num)>0.5:
                 print(each, "\t",good_num,"\t", bad_num,"\t", bad_num/(good_num+bad_num))
```

```
fuck   303  709  0.700592885375494
susmita     0    1    1.0
secreatly   0    1    1.0
idiots      39   157  0.8010204081632653
motherfucker     3    11   0.7857142857142857
Cuts   1    3    0.75
unseat      1    3    0.75
Yogendra    0    2    1.0
Palestinian      378  533  0.5850713501646543
Baldistan   0    1    1.0
Kashmiris   50   51   0.504950495049505
Marathis    4    19   0.8260869565217391
Gujaratis   10   17   0.6296296296296297
```

Fig. 16: Calculate bad ratio of sentences within the total training dataset

After all the above process are done, we successfully reduced the number of cannot-find-word from 355920 to 79883.

count the words that are not in wiki

```
uncor = []
for i in oov:
    uncor.append(i[0])
print(len(uncor))
```

```
79883
```

Fig. 17: The list length changes from 355920 to 79883

B. Embedding

Before we feed the training data into our model, the first thing after cleaning the data is to transfer string type data to vectors. The dictionary we use to convert string is wiki-news-300d-1M.vec which supplied by Kaggle. Wiki-news-300d-1M is a dictionary with one million words(including punctuations and numbers). And each key is one word, each value is a 1-dimension word vector with 300 numbers.

For the words in training data which can be found in wiki-news-dictionary, we replace them with vectors. For those can't be found in wiki-news-dictionary and also escaped from our preprocessing, we initialize them with a random vector which is based on the distribution of the words in text.

```
In [19]: embed_size = 300
embedding_path = "../input/quora-insincere-questions-classification/embeddings/wiki-news-300d-1M/wiki-news-300d-1M.
def get_coefs(word,*arr): return word, np.asarray(arr, dtype='float32')
embedding_index = dict(get_coefs(*o.split(" ")) for o in open(embedding_path, encoding='utf-8', errors='ignore') if
all_embs = np.stack(embedding_index.values())
emb_mean,emb_std = all_embs.mean(), all_embs.std()
word_index = tk.word_index
nb_words = min(max_features, len(word_index))
embedding_matrix = np.random.normal(emb_mean, emb_std, (nb_words + 1, embed_size))
for word, i in word_index.items():
    if i >= max_features: continue
    embedding_vector = embedding_index.get(word)
    if embedding_vector is not None: embedding_matrix[i] = embedding_vector
```

Fig. 18: Initialize training dataset with random vector

After all these steps, the data is ready to be fed into our models.

C. LSTM

```
In [21]: class NeuralNet(nn.Module):
    def __init__(self):
        super(NeuralNet, self).__init__()
        hidden_size = 128
        self.embedding = nn.Embedding(max_features, embed_size)# 120000, 300
        self.embedding.weight = nn.Parameter(torch.tensor(embedding_matrix, dtype=torch.float3
2))#[120001, 300]
        self.embedding.weight.requires_grad = False
        self.embedding_dropout = nn.Dropout2d(0.1)
        self.lstm = nn.LSTM(embed_size, hidden_size, bidirectional=True, batch_first=True)
        self.linear = nn.Linear(512, 16)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.1)
        self.out = nn.Linear(16, 1)

    def forward(self, x):
        h_embedding = self.embedding(x) #[512, 72, 300]
        h_embedding = torch.squeeze(self.embedding_dropout(torch.unsqueeze(h_embedding, 0)))#[5
12, 72, 300]
        h_lstm, _ = self.lstm(h_embedding)#[512, 72, 256]
        avg_pool = torch.mean(h_lstm, 1)#[512, 256]
        max_pool, _ = torch.max(h_lstm, 1)#[512, 256]
        conc = torch.cat((avg_pool, max_pool), 1)#[512, 512]
        conc = self.relu(self.linear(conc))#[512, 16]
        conc = self.dropout(conc)#[512, 16]
        out = self.out(conc)#[512, 1]
        return out
```

Fig. 19: One layer LSTM model

1) One layer LSTM: First model is just a simple one-layer LSTM model, after going through the LSTM layer, the output of that layer goes through two separate pooling layers in order to obtain information as much as possible. Then we concate the two outputs of the pooling layers and put it into an activation layer and then dropout followed by a linear layer.

There's only one hyperparameter we adjust in this project-number of epochs. In order to save time, we first tried 5, 7, 10 epochs to train our raw data without replacing misspelling words. The result is shown in Fig.20.

Model	epoch	Score
one-layer LSTM	5	0.62508
one-layer LSTM	7	0.63011
one-layer LSTM	10	0.61018

Fig. 20: Different epochs with one-layer LSTM

Obviously, 7-epoch LSTM has the highest score. And this is the reason that we choose seven epochs to train our following models. With preprocessed data fed into model, we get the result shown below. And the average time training one epoch is at around 75s.

Fig. 21: One-layer LSTM(7 epoch)

```
Fold 1
Epoch 1/7       loss=0.1475     val_loss=0.1184         val_f1=0.6291 best_t=0.23       time=7
6.10s
Epoch 2/7       loss=0.1264     val_loss=0.1112         val_f1=0.6498 best_t=0.23       time=7
4.10s
Epoch 3/7       loss=0.1192     val_loss=0.1095         val_f1=0.6627 best_t=0.22       time=7
3.72s
Epoch 4/7       loss=0.1152     val_loss=0.1029         val_f1=0.6686 best_t=0.27       time=7
6.08s
Epoch 5/7       loss=0.1115     val_loss=0.1012         val_f1=0.6712 best_t=0.35       time=7
3.40s
Epoch 6/7       loss=0.1074     val_loss=0.1023         val_f1=0.6703 best_t=0.30       time=7
6.17s
Epoch 7/7       loss=0.1041     val_loss=0.1051         val_f1=0.6731 best_t=0.23       time=7
3.33s
Validation loss:  0.10513302894618638
```

Fig. 22: Part training result. Average training time consuming for one epoch is around 75 second

2) Two and three layers LSTM: We also tried two -layer and three-layer LSTM models with seven epochs. With raw data, the two-layer LSTM got 0.63068 on final result which performs better than one-layer LSTM but the three-layer LSTM got 0.62601 which is worse than the two-layer LSTM model. The reason may be that three layers model is too much for the training data, the model got overfitting on the data.

```
In [24]:
class NeuralNet(nn.Module):
    def __init__(self):
        super(NeuralNet, self).__init__()
        hidden_size = 128
        self.embedding = nn.Embedding(max_features, embed_size)# 120000, 300
        self.embedding.weight = nn.Parameter(torch.tensor(embedding_matrix, dtype=torch.float3
2))#[120001, 300]
        self.embedding.weight.requires_grad = False
        self.embedding_dropout = nn.Dropout2d(0.1)
        self.lstm = nn.LSTM(embed_size, hidden_size, bidirectional=True, batch_first=True)
        self.lstm2 = nn.LSTM(256, hidden_size, bidirectional=True, batch_first=True)
        self.linear = nn.Linear(512, 16)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.1)
        self.out = nn.Linear(16, 1)

    def forward(self, x):
        h_embedding = self.embedding(x) #[512, 72, 300]
        h_embedding = torch.squeeze(self.embedding_dropout(torch.unsqueeze(h_embedding, 0)))#[5
12, 72, 300]
        h_lstm, _ = self.lstm(h_embedding)#[512, 72, 256]
        h_lstm2, _ = self.lstm2(h_lstm)
        avg_pool = torch.mean(h_lstm2, 1)#[512, 256]
        max_pool, _ = torch.max(h_lstm2, 1)#[512, 256]
        conc = torch.cat((avg_pool, max_pool), 1)#[512, 512]
        conc = self.relu(self.linear(conc))#[512, 16]
        conc = self.dropout(conc)#[512, 16]
        out = self.out(conc)#[512, 1]
        return out
```

Fig. 23: Two-layer LSTM

After feeding preprocessed data to two-layer LSTM model, we got 0.68421 which is the highest score among all the LSTM models.

Fig. 24: Two-layer LSTM score

D. Simple GRU

Simply replacing nn.LSTM with nn.GRU give us one-layer GRU model. Compared to one-layer LSTM, we got a higher score(0.68206) and less training time(at around 44s).

Fig. 25: One-layer GRU socre

```
Fold 1
Epoch 1/7       loss=0.1518     time=43.79s
Epoch 2/7       loss=0.1303     time=45.83s
Epoch 3/7       loss=0.1251     time=43.85s
Epoch 4/7       loss=0.1214     time=43.50s
Epoch 5/7       loss=0.1179     time=43.23s
Epoch 6/7       loss=0.1146     time=44.26s
Epoch 7/7       loss=0.1108     time=43.65s
Validation loss:  0.10700189937665172
```

Fig. 26: Part training result. Average training time consuming for one epoch is around 44 second

E. Attention

The structure of this soft attention model is shown below. The score we got is 0.68506 which is the highest score we have.

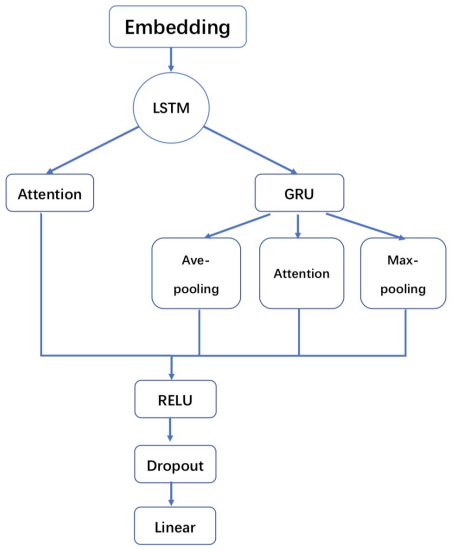


Fig. 27: Attention model

In the fig above, we add two weights matrix at the attention step. In training data, not all words' contributions are the same. The intuition to use attention model is to let model pay more attention on some 'useful' words instead of treat every word equally.

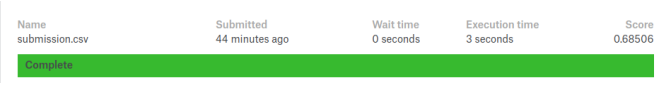


Fig. 28: Attention model score

F. Combination

The above three methods only use one kind of word-embedding, because this can only cover a part of words, so the two embedding methods are combined, and different weights are given, and finally get better results

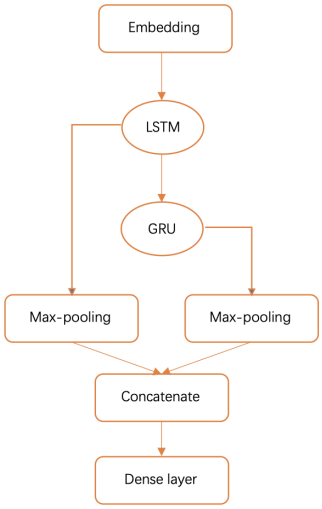


Fig. 29: Combination model

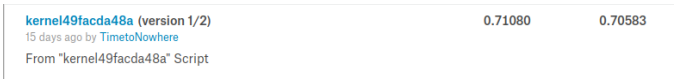


Fig. 30: Combination model score

IV. Application

In order to enable the model to be applied, we adopt a web-based approach to bring the model online. Anyone who visits our page through a browser can use our model. At the same time, we also designed the front-end interface and back-end processing logic. In the end, in order to test our model, users can customize input test problems, and its interactive interface is very friendly and beautiful.

A. Introduction

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.

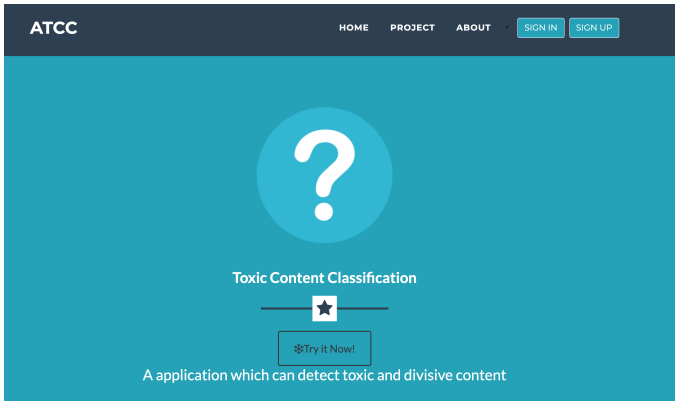


Fig. 31: UI interface

B. Request

When the user submits a question for input, the application will carry the question's input request to the corresponding url. First, the request interface is parsed through the url.py server. Second, import the trained model and process the problem text. After that, use view.py to apply the model to make predictions and get the results. Finally, return the results to the web interface.

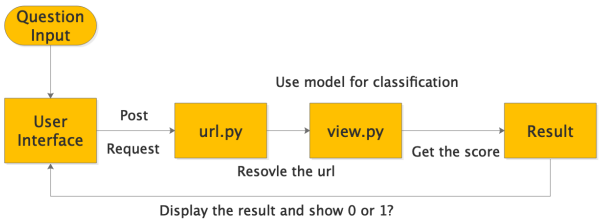


Fig. 32: Request process

C. Apply

Figure 33 shows some test questions and results, where a check mark indicates that the question is not an insincere question, the cross represents this question is an insincere question. It takes an average of 10 seconds from the time a user enters a question until the result is obtained. In 100 random test results, the accuracy rate is as high as 98%. Therefore, our harmful problem identification application is fast and accurate. In the future, it can provide corresponding services as an interface to other applications.

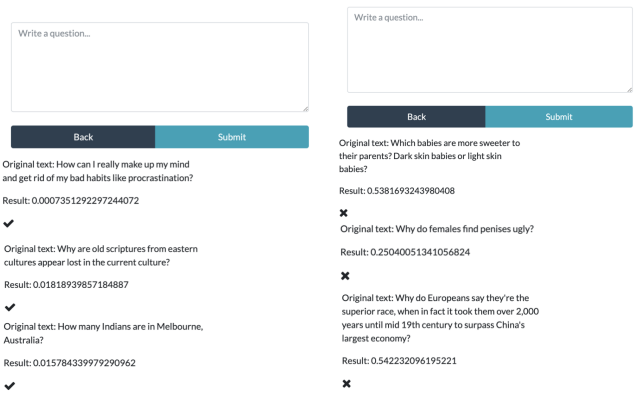


Fig. 33: Result for application

V. Reproduce

To reproduce the project follow the instruction on: https://github.com/nature1995/AI-Toxic-Content-Classification-in-Django

A. Model

1) Train: The dataset we use is on the kaggle competition: https://www.kaggle.com/c/quora-insincere-questions-classification. To train our models, after downloading the code, you can:

1)Upload to your own kaggle kernel and you'll be able to run it.

2)Download the dataset on your own machine and change the path in the code to where you store the dataset. Then run the code.

2) Use: You need to download below four things to one folder:

1)pre-trained model(link is above)

2)pre_trained_predict.ipynb

3)en_core_web_lg-2.2.5 (This is a folder on google drive: https://drive.google.com/open?id=1h8ygWJh1iKfgBBnZCOxHdEYFK5iT2YIl)

4)quora_dict.txt (https://drive.google.com/open?id=17zDlY0jpg0IdSRjv9ersaZPWHu_E3SSi)

In the pre_trained_predict.ipynb, change the variable s to the sentence you want to test and run it.

VI. Conclusion

In this project, we used three models and a combination method to train the model. The final trained model can have a good accuracy with 70%. We also save the model and put it on the server as an application to determine in real time

B. Application

For this application, you need to upload the model and website code to the AWS server, and access the external network by deploying Django web services, and then you need to add the model building part.

whether the sentence entered by the user is offensive. The average judgment speed is 15 seconds, which can correctly judge most of the input question.

Reference

[1] Goodfellow, I. & Bengio, Y. & Courville, A. (2016) Deep Learning. *Modern Practical Deep Networks*, pp. 367–403. Cambridge, MA: MIT Press.

[2] Supervise, ly., (2017) Towards Data Science: *Evolution: from vanilla RNN to LSTM & GRU.* [online]Available at ¡https://towardsdatascience.com/lecture-evolution-from-vanilla-rnn-to-gru-lstms¿ (02/12/2019 12:14)

[3] Hoffman, G.(2018). *Introduction to LSTMs with Pytorch.* O'Reilly AI Newsletter, 54, 66-80.

[4] Adrianna, X., (2017) More or Less?:*Talking about the difference with LSTM and GRU.* [online]Available at ¡https://blog.csdn.net/u012223913/article/details/77724621¿ (30/11/2019 17:34)

[5] Hao, L., (2018) Talking about Nature Language Processing: *Word Embedding*. [online]Available at ¡https://blog.csdn.net/L_R_H000/article/details/81320286¿ (02/12/2019 12:44)

[6] Xiaozhou, Y. & Feifei, T. & Rongzhi, Q. (2016). *Improvement of activation function in recurrent neuron network.* Journal of computer and modernization,12, 14-27.