notebook_final

December 12, 2020

1 Text Classification Competition: Sarcasm Detection

Chua Yeow Long, ylchua2@illinois.edu

In this notebook, i'll first try using a simple word count model using sklearn's CountVectorizer to perform sarcasm detection. Next, I'll use GLoVe embedding and add neural network layers towards the end, training just the added layers using the provided dataset. Finally, I'll fine tune BERT layers, just fine-tuning the entire BERT model which consists of hundreds of millions of parameters using the training dataset with a pretty beefy machine.

We'll first need to import the necessary libraries.

[2]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from nltk.corpus import stopwords
from nltk.util import ngrams
from sklearn.feature_extraction.text import CountVectorizer
from collections import defaultdict
from collections import  Counter
plt.style.use('ggplot')
stop=set(stopwords.words('english'))
import re
from nltk.tokenize import word_tokenize
import gensim
import string
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from tqdm import tqdm
from keras.models import Sequential
from keras.layers import Embedding,LSTM,Dense,SpatialDropout1D
from keras.initializers import Constant
from sklearn.model_selection import train_test_split
from keras.optimizers import Adam
```

We'll make use of pandas' read_json module to load the data into a pandas dataframe where we'll perform our work

```
[3]: df = pd.read_json('data/train.jsonl', lines=True)
```

```
[4]: df.head()
```

```
[4]:     label                                           response  \
   0  SARCASM  @USER @USER @USER I don't get this .. obviousl…
   1  SARCASM  @USER @USER trying to protest about . Talking …
   2  SARCASM  @USER @USER @USER He makes an insane about of …
   3  SARCASM  @USER @USER Meanwhile Trump won't even release…
   4  SARCASM  @USER @USER Pretty Sure the Anti-Lincoln Crowd…

                                                context
   0  [A minor child deserves privacy and should be …
   1  [@USER @USER Why is he a loser ? He's just a P…
   2  [Donald J . Trump is guilty as charged . The e…
   3  [Jamie Raskin tanked Doug Collins . Collins lo…
   4  [Man … y ’ all gone " both sides " the apoca…
```

1.1 Data-Preprocessing

We’ll just do a bit of preprocessing by removing non-alphabets and removing stopwords

```
[6]: from nltk.corpus import stopwords
     import re
```

```
[7]: def refineWords(s):

         letters_only = re.sub("[^a-zA-Z]", " ", str(s))
         words = letters_only.lower().split()
         stops = set(stopwords.words("english"))
         meaningful_words = [w for w in words if not w in stops]

         return( " ".join( meaningful_words ))
```

1.2 With or without context

We’ll just combine the context into the response. I suppose having extra information/context is always good.

```
[8]: df['response'] = df['response'].apply(refineWords)
     df['context'] = df['context'].apply(refineWords)

     df['response'] = df['context'] + ' ' + df['response']
```

We’ll need to change the predictor variable which consists of ‘SARCASM’ and ‘NON_SARCASM’ to ’1’s and ’0’s so that we can feed them into the model

[9]:
```
def sarcasm_mapping(input):
    if input == 'SARCASM':
        return 1
    else:
        return 0

df['label'] = df['label'].apply(sarcasm_mapping)
```

We initialize a simple countvectorizer from sklearn for our first model.

[10]:
```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer(analyzer = "word",   \
                             tokenizer = None,    \
                             preprocessor = None, \
                             stop_words = None,   \
                             max_features = 5000)
```

We duplicated copies of the training dataset so we have individual copies to work on for each model.

[ ]:
```
df_tmp = df.copy()
df_tmp_bert = df.copy()
df_tmp_xlm = df.copy()
```

1.3 Data-Preprocessing

We'll need to ensure our training data is in the correct format for further analysis

[ ]:
```
df["response"] = vectorizer.fit_transform(df["response"]).toarray()
df["context"] = vectorizer.fit_transform(df["context"]).toarray()
```

1.4 Modelling

We'll train our first model which is a simple word count model.

[11]:
```
from sklearn.ensemble import RandomForestClassifier

forest = RandomForestClassifier(n_estimators = 100)
features_forest = df[["response","context"]].values
my_forest = forest.fit(features_forest, df['label'])
```

We check the distribution of the target variable to see and indeed, the dataset is perfectly balanced.

[12]:
```
df['label'].value_counts()
```

[12]:
```
1    2500
0    2500
Name: label, dtype: int64
```

1.5 Data-Preprocessing for test set

We do a similar preprocessing for the test set as well.

```
[13]: test_df = pd.read_json('data/test.jsonl', lines=True)
```

```
[14]: test_df.head()
```

```
[14]:          id                                           response  \
      0  twitter_1  @USER @USER @USER My 3 year old , that just fi…
      1  twitter_2  @USER @USER How many verifiable lies has he to…
      2  twitter_3  @USER @USER @USER Maybe Docs just a scrub of a…
      3  twitter_4  @USER @USER is just a cover up for the real ha…
      4  twitter_5  @USER @USER @USER The irony being that he even…

                                                   context
      0  [Well now that ’ s problematic AF <URL>, @USER…
      1  [Last week the Fake News said that a section o…
      2  [@USER Let ’ s Aplaud Brett When he deserves i…
      3  [Women generally hate this president . What's …
      4  [Dear media Remoaners , you excitedly sharing …
```

We simply combine the context into the response.

```
[15]: test_df['response'] = test_df['response'].apply(refineWords)
      test_df['context'] = test_df['context'].apply(refineWords)

      test_df['response'] = test_df['context'] + ' ' + test_df['response']
```

We duplicate the test dataset so that we have individual copies to work on

```
[16]: test_df_tmp = test_df.copy()
      test_df_tmp_bert = test_df.copy()
      test_df_tmp_xlm = test_df.copy()
```

We need to make sure that the test data is in the same format as the train data so we do essentially the same analysis

```
[ ]: test_df["response"] = vectorizer.fit_transform(test_df["response"]).toarray()
     test_df["context"] = vectorizer.fit_transform(test_df["context"]).toarray()
```

The input format needs to be in a form of an array

```
[17]: features_forest_test = test_df[["response","context"]].values
```

We perform predictions of the test features using the trained CountVectorizer model

```
[18]: my_prediction = my_forest.predict(features_forest_test)
```

We convert the predictions into a pandas dataframe so we can use to_csv to easily output our predictions in the right format

```
[20]: test_df['label'] = pd.DataFrame(my_prediction)
```

We do a value_count on the predictions to check how well the model worked. Well the model predicted 100% no sarcasm which is clearly problematic.

```
[21]: test_df['label'].value_counts()
```

```
[21]: 0    1800
      Name: label, dtype: int64
```

```
[22]: test_df.describe()
```

```
[22]:           response      context   label
      count  1800.000000  1800.000000  1800.0
      mean      0.002778     0.002222     0.0
      std       0.097170     0.074523     0.0
      min       0.000000     0.000000     0.0
      25%       0.000000     0.000000     0.0
      50%       0.000000     0.000000     0.0
      75%       0.000000     0.000000     0.0
      max       4.000000     3.000000     0.0
```

After we are done making predictions using the trained model we'll need to change the 1's and 0's of the label column back to sarcasm and non-sarcasm.

```
[23]: def sarcasm_reverse_mapping(input):
          if input == 1:
              return 'SARCASM'
          else:
              return 'NOT_SARCASM'

      test_df['label'] = test_df['label'].apply(sarcasm_reverse_mapping)
```

We make use of to_csv of the pandas library to create our answer.txt

```
[24]: #test_df[['id','label']].to_csv('answer.txt', index=False, header=None)
```

1.6 Modelling using GLoVe embedding and some neural network layers

First, we'll need to create the corpus. We'll need the tqdm module and NLTK's word tokenize as well as stopwords to preprocess our data.

```
[25]: def create_corpus(df):
          corpus=[]
          for tweet in tqdm(df['response']):
```

```
        words=[word.lower() for word in word_tokenize(tweet) if((word.
↪isalpha()==1) & (word not in stop))]
        corpus.append(words)
    return corpus
```

We concat/combine the training and test dataset so create a corpus of both the train and test datasets.

[26]:
```
df_tmp['response'] = df_tmp['response'].astype('string')
test_df_tmp['response'] = test_df_tmp['response'].astype('string')

df_new = df_tmp.append(test_df_tmp)

corpus=create_corpus(df_new)
```

```
100%|
| 6800/6800 [00:01<00:00, 3772.90it/s]
```

We'll first need to download the glove embedding and load it ensuring the correct formats,

[27]:
```
embedding_dict={}
with open('data/glove.6B.200d.txt','r', encoding="utf8") as f:
    for line in f:
        values=line.split()
        word=values[0]
        vectors=np.asarray(values[1:],'float32')
        embedding_dict[word]=vectors
f.close()
```

We first start by initializing a Keras tokenizer and train it with the corpus we obtained earlier. We perform truncating and padding to get sequences of the same length.

[28]:
```
MAX_LEN=50
tokenizer_obj=Tokenizer()
tokenizer_obj.fit_on_texts(corpus)
sequences=tokenizer_obj.texts_to_sequences(corpus)

tweet_pad=pad_sequences(sequences,maxlen=MAX_LEN,truncating='post',padding='post')
```

Our tokenizer has a word number count in the form of word_index. We'll need it later.

[29]:
```
word_index=tokenizer_obj.word_index
```

We'll need to make sure of tqdm module and GLoVe embedding to obtain the embedding matrix to be used to create an embedding layer using Keras.

[30]:
```
num_words=len(word_index)+1
embedding_matrix=np.zeros((num_words,200))
```

```
for word,i in tqdm(word_index.items()):
    if i > num_words:
        continue

    emb_vec=embedding_dict.get(word)
    if emb_vec is not None:
        embedding_matrix[i]=emb_vec
```

```
100%|                                        |
34445/34445 [00:00<00:00, 569822.52it/s]
```

We'll build our neural network sequentially. We first add the embedding layer and add LSTM layers of decreasing nodes with dropout to reduce overfitting.

[31]:
```
model=Sequential()

embedding=Embedding(num_words,200,embeddings_initializer=Constant(embedding_matrix),
                   input_length=MAX_LEN,trainable=False)

model.add(embedding)
model.add(SpatialDropout1D(0.10))
model.add(LSTM(128*2, dropout=0.10, recurrent_dropout=0.10,␣
 ↪return_sequences=True))
model.add(Dense(64*2, activation='relu'))
model.add(LSTM(128, dropout=0.10, recurrent_dropout=0.10,␣
 ↪return_sequences=True))
model.add(Dense(64, activation='relu'))
model.add(LSTM(64, dropout=0.10, recurrent_dropout=0.10, return_sequences=True))
model.add(Dense(32, activation='relu'))
model.add(LSTM(32, dropout=0.10, recurrent_dropout=0.10, return_sequences=True))
model.add(Dense(16, activation='relu'))
model.add(LSTM(16, dropout=0.10, recurrent_dropout=0.10))

model.add(Dense(1, activation='sigmoid'))
```

We'll make use of the Adam optimizer with a small learning rate. We'll compile the model specifying the loss, optimizer and metrics to optimize our model for.

[ ]:
```
optimzer=Adam(learning_rate=1e-5*10)

model.
 ↪compile(loss='binary_crossentropy',optimizer=optimzer,metrics=['accuracy'])
```

We split our data back into train and test datasets. The first 5000 is our training data the rest is test.

[32]:
```
train=tweet_pad[:5000]
test=tweet_pad[5000:]
```

We'll perform a train-validation split to obtain validation data.

[33]:
```
X_train,X_test,y_train,y_test=train_test_split(train,df_tmp['label'].
 ↪values,test_size=0.15)
```

We can now start training our model specifying the batch size, epochs and validation datasets.

[34]:
```
history=model.
 ↪fit(X_train,y_train,batch_size=4*16,epochs=1,validation_data=(X_test,y_test),verbose=2)
```

```
Train on 4250 samples, validate on 750 samples
Epoch 1/1
 - 11s - loss: 0.6914 - accuracy: 0.5214 - val_loss: 0.6810 - val_accuracy:
0.6293
```

We perform predictions using our trained model. As the predictions are not strictly 1's and 0's we'll just simply round them to the nearest integer.

[35]:
```
y_pre=model.predict(test)

test_df['label'] = np.round(y_pre).astype(int)
```

We'll need to do a mapping of 0's and 1's to 'NOT_SARCASM' and 'SARCASM'

[36]:
```
def sarcasm_reverse_mapping(input):
    if input == 1:
        return 'SARCASM'
    else:
        return 'NOT_SARCASM'

test_df['label'] = test_df['label'].apply(sarcasm_reverse_mapping)
```

We convert our predictions into a csv file with the right format with the following.

[37]:
```
#test_df[['id','label']].to_csv('answer.txt', index=False, header=None)
```

2 Modelling using BERT, retraining the entire BERT architecture to the data.

We'll use the official tokenization script from tensorflow. You can download the tokenization.py by using wget or downloading it manually.

[38]:
```
!wget --quiet https://raw.githubusercontent.com/tensorflow/models/master/
 ↪official/nlp/bert/tokenization.py
```

```
SYSTEM_WGETRC = c:/progra~1/wget/etc/wgetrc
syswgetrc = C:\Program Files (x86)\GnuWin32/etc/wgetrc
```

We'll import the necessary libraries needed for this section.

[39]:
```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import Model
from tensorflow.keras.callbacks import ModelCheckpoint
import tensorflow_hub as hub

import tokenization
```

We'll need to encode the input texts to BERT's input format. For each row of the training data, we first tokenize the text using NLTK tokenizer before proceeding to convert the obtained tokens into IDs.

[40]:
```
def bert_encode(texts, tokenizer, max_len=512):
    all_tokens = []
    all_masks = []
    all_segments = []

    for text in texts:
        text = tokenizer.tokenize(text)

        text = text[:max_len-2]
        input_sequence = ["[CLS]"] + text + ["[SEP]"]
        pad_len = max_len - len(input_sequence)

        tokens = tokenizer.convert_tokens_to_ids(input_sequence)
        tokens += [0] * pad_len
        pad_masks = [1] * len(input_sequence) + [0] * pad_len
        segment_ids = [0] * max_len

        all_tokens.append(tokens)
        all_masks.append(pad_masks)
        all_segments.append(segment_ids)

    return np.array(all_tokens), np.array(all_masks), np.array(all_segments)
```

We'll create the BERT layer specifying the inputs and creation of the BERT layer using the inputs to obtain the output sequence. Since we are re-training the entire BERT architecture to the dataset, we'll just add a sigmoid dense layer to perform classification. We specify the BERT model and the adam optimizer, optimization loss and accuracy metrics.

[41]:
```
def build_model(bert_layer, max_len=512):
    input_word_ids = Input(shape=(max_len,), dtype=tf.int32,␣
 ↪name="input_word_ids")
    input_mask = Input(shape=(max_len,), dtype=tf.int32, name="input_mask")
    segment_ids = Input(shape=(max_len,), dtype=tf.int32, name="segment_ids")

    _, sequence_output = bert_layer([input_word_ids, input_mask, segment_ids])
    clf_output = sequence_output[:, 0, :]

    # Without Dropout
    out = Dense(1, activation='sigmoid')(clf_output)

    model = Model(inputs=[input_word_ids, input_mask, segment_ids], outputs=out)
    model.compile(Adam(lr=learning_rate), loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])

    return model
```

We load BERT from TensorFlow Hub. TensorFlow Hub provides pre-trained models for us to use and we load the model using TensorFlow hub module.

[42]:
```
module_url = "https://tfhub.dev/tensorflow/bert_en_uncased_L-24_H-1024_A-16/1"

bert_layer = hub.KerasLayer(module_url, trainable=True)
```

We'll need to load the tokenizer from the BERT layer.

[43]:
```
vocab_file = bert_layer.resolved_object.vocab_file.asset_path.numpy()
do_lower_case = bert_layer.resolved_object.do_lower_case.numpy()
tokenizer = tokenization.FullTokenizer(vocab_file, do_lower_case)
```

Next, we process the text into BERT required formats such as tokens, masks and segment flags.

[44]:
```
train_input = bert_encode(df_tmp_bert.response.values, tokenizer, max_len=160)
test_input = bert_encode(test_df_tmp_bert.response.values, tokenizer,␣
 ↪max_len=160)
train_labels = df_tmp_bert.label.values
```

This are the parameters we'll need for our BERT model. Due to limited processing capacity, there aint much room for me to play with.

[45]:
```
Max_length = 42
Dropout_num = 0
learning_rate = 6e-6
valid = 0.15
epochs_num = 5
batch_size_num = 4
ids_error_corrected = True
```

We build our BERT model and note that we have 335 million traininable parameters.

[46]:
```
model_BERT = build_model(bert_layer, max_len=160)
model_BERT.summary()
```

```
Model: "model"
________________________________________________________________________________
```

```
------------------
Layer (type)                    Output Shape         Param #     Connected to
================================================================================
==================
input_word_ids (InputLayer)     [(None, 160)]        0

--------------------------------------------------------------------------------
------------------
input_mask (InputLayer)         [(None, 160)]        0

--------------------------------------------------------------------------------
------------------
segment_ids (InputLayer)        [(None, 160)]        0

--------------------------------------------------------------------------------
------------------
keras_layer (KerasLayer)        [(None, 1024), (None 335141889
input_word_ids[0][0]
input_mask[0][0]
segment_ids[0][0]

--------------------------------------------------------------------------------
------------------
tf_op_layer_strided_slice (Tens [(None, 1024)]       0
keras_layer[0][1]

--------------------------------------------------------------------------------
------------------
dense (Dense)                   (None, 1)            1025
tf_op_layer_strided_slice[0][0]
================================================================================
==================
Total params: 335,142,914
Trainable params: 335,142,913
Non-trainable params: 1
--------------------------------------------------------------------------------
------------------
```

We'll train the BERT model, serializing the best model to a file for later predictions. Finally, we get to train our BERT model.

[47]:
```
checkpoint = ModelCheckpoint('model_BERT.h5', monitor='val_loss',␣
 ↪save_best_only=True)

train_history = model_BERT.fit(
    train_input, train_labels,
    validation_split = valid,
    epochs = epochs_num,
    callbacks=[checkpoint],
    batch_size = batch_size_num
)
```

```
Train on 4250 samples, validate on 750 samples
```

```
Epoch 1/5
4250/4250 [==============================] - 303s 71ms/sample - loss: 0.5174 -
accuracy: 0.7588 - val_loss: 0.5598 - val_accuracy: 0.6080
Epoch 2/5
4250/4250 [==============================] - 278s 65ms/sample - loss: 0.4388 -
accuracy: 0.8024 - val_loss: 0.4221 - val_accuracy: 0.7653
Epoch 3/5
4250/4250 [==============================] - 265s 62ms/sample - loss: 0.3094 -
accuracy: 0.8727 - val_loss: 1.3605 - val_accuracy: 0.4267
Epoch 4/5
4250/4250 [==============================] - 265s 62ms/sample - loss: 0.1140 -
accuracy: 0.9614 - val_loss: 2.7396 - val_accuracy: 0.3747
Epoch 5/5
4250/4250 [==============================] - 265s 62ms/sample - loss: 0.0422 -
accuracy: 0.9861 - val_loss: 2.8826 - val_accuracy: 0.4853
```

We perform predictions by loading weights from the file we serialized the model earlier.

[48]:
```
model_BERT.load_weights('model_BERT.h5')
test_pred_BERT = model_BERT.predict(test_input)
test_pred_BERT_int = test_pred_BERT.round().astype('int')
```

We'll need to perform a mapping of the 1's and 0's back to 'SARCASM' and 'NOT_SARCASM' for submission.

[49]:
```
test_df['label'] = test_pred_BERT_int
def sarcasm_reverse_mapping(input):
    if input == 1:
        return 'SARCASM'
    else:
        return 'NOT_SARCASM'

test_df['label'] = test_df['label'].apply(sarcasm_reverse_mapping)
```

We generate the 'answer.txt' in the format required for submission.

[ ]:
```
#test_df[['id','label']].to_csv('answer.txt', index=False, header=None)
```