2020

Text Classification Competition: Twitter Sarcasm Detection

CS410 - COURSE PROJECT FINAL DOCUMENT

MOHAN KHANAL (MKHANAL2@ILLINOIS.EDU)

Contents

How to use/run the code 2
Background 2
Model Design 2
Detail Approach / Code Walkthrough 4
Data Pre-processing 4
Data preparation for training 6
Model Building and Training 8
Prediction for Test set 14
Prediction Results 15
Other Approaches 15
Conclusion 16
References 16

How to use/run the code

The code is developed using python 3 (Jupiter notebook). There is folder called source_code on the GitHub (https://github.com/mkhanal2/CourseProject) where we put all the documentation and source-code for the project. Follow following instruction to run the code:

- Download the folder "source_code" from GitHub (link above)
- Makes sure you have folder called "data" under source_code folder which has (train.jsonl, test.jsonl and glove.twitter.27B.25d.txt) , all of these files and folder are already in GitHub.
- Now open Jupiter Notebook on you laptop (usually within Anaconda).
- Using Jupiter Notebook open the source code from the downloaded folder (source code file name: ***Project Source Code.ipynb***)
- After that you can run the code. Please make sure that all the packages used in the 2nd cell of the notebook are already installed. I have provided the instruction on 1st cell of the notebook on how to install the packages.

Background

As part of the final project for course CS410-Text Information System, I have done project for Twitter Sarcasm Detection which is part of the Text Classification competition.

As part of this project there were two sets of data file given to us. One was for training and another was for testing (for which we would have to do our prediction). These data files contained the twitter responses. These response text were in context to some conversation happening in twitter feed. These data file contained those context conversation as-well. So, we could use both response text and context text if we want to for training our model for this classification. Since this was a classification task, the training file also had the label for each data point as "SARCASM" or "NOT_SARCASM". Our job was to predict the same thing for all the records present in the testing file. Training file had 5000 labelled data-set and testing set had 1800 data-set. So, at the end we would have to predict the label for those 1800 test data-set.

This project required some machine learning task to train the model using the training data and finally predict the outcome for test-data set using that trained classification model. This document explains all the details around the different approaches that were carried out to train the model and will list out the final results that I got from those trained model.

Model Design

As part of this project, I have used recurrent neural network models called LSTM (Long Short Term Memory) model's. These models are mostly used for sequential data. Since, the text that we are using as our input is sequential and sarcasm might depend on how the sentence is structured, so using LSTM would help us utilize the sequential nature of the data.

As explained in previous section, we have “response” text as one set of input data that we can use and another is we have “context” text as another set of input that we can use. So, here were the first two sets of model that we can design using these inputs.

Model-1

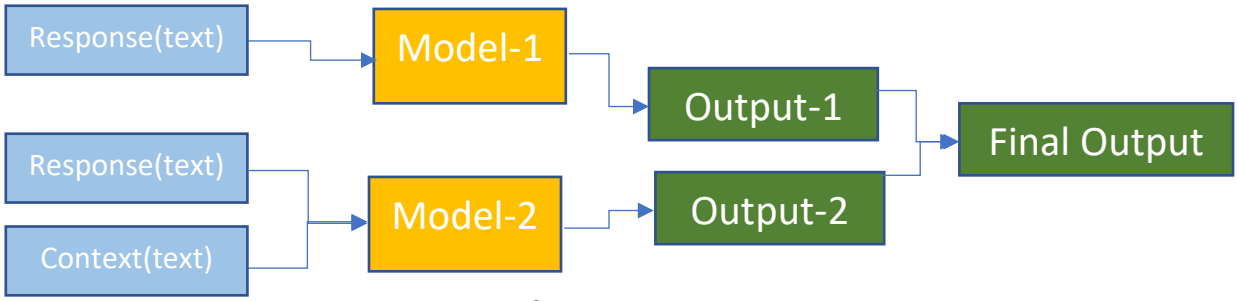
- Using response only as part of our input and feeding that data into the model to predict if that response is sarcastic or not. Below diagram gives the flow of the model.

Model-2

- Another approach is rather than only using the response as an input we can use both response and context as our input to train our model and get output from there. Below diagram show the design for the same.

Results Combined/Aggregated

- Finally we can decide to combine the results from Model-1 and Model-2 and then get our final aggregated results. This combined results approach, that is combining both Model-1 and Model-2 was able to beat the baseline score for me. Sperate Model-1 and Model-2 were not able to beat the baseline score. Below is the diagram for the combined results model. I will be discussing the detail around this model in another section.



Detail Approach / Code Walkthrough

As stated in the previous section I have used LSTM as a model for training my classification model. The program is written in python (Jupter Notebook). I will walk-over the detail steps that was carried out as part of my code in this section.

Here are the steps that were carried out , which is going to be covered in details:

1. Data Pre-processing → Reading the file, and cleaning the data for training
2. Data preparation for training → Creating the vocabulary, creating the input vector for text.
3. Model Building and Training → Building different models and training them with the data/label prepared
4. Prediction for Test set → Combing results from multiple model to improve prediction score.

Data Pre-processing

Data pre-processing is the first process that is carried out before building the model. As part of data pre-processing step , following steps are carried out:

- Read the training file provided ("train.jsonl" - 5000 data records). This is a json file, and contains attributes ("label", "response" and "context"). I have used pandas json file reader function (read_json) to read the json file.
- After reading the file, the emoji and hashtags are read separately using custom built functions for reading emoji and hashtags and placed on separate columns in pandas data frame, which could be utilized to use as input for a model. The final model used for this project doesn't utilize hashtags or emoji, but emoji's were utilized on some of my model's for testing, but was not carried out to the final model.
- After that the text "response" , which is one of the main input data-set for our model is cleaned, using the custom build function. This function will remove any junk characters and rephrase some of the wordings , trim the text to remove white-spaces, remove the stops words like (the, is, etc.) and lemmatize the text so that the similar type of words are represented the same.
- After that, context goes through the same clean-up process as that for response. Before that, context data is a list/array. Which means that there could be multiple level of replies for the tweet and the response above was the reply for the last reply from the context. So, we use the list object to get the last reply from it to set the context for us. After that it goes through the same clean-up process described above for response.
- Similar to the train data, test data goes through the same pre-processing step. "test.jsonl" (1500 data records) is similar structure of training data (only thing different is, this data doesn't have label).

Below is the code screenshot for the steps explained above.

Data-read and clean-up code (train.jsonl):

```
data = pd.read_json (r'.\data\train.jsonl',encoding='UTF-8', lines=True)
data['response_ht'] = data['response'].apply(lambda x: get_hash_tags(x))
data['response_emoji'] = data['response'].apply(lambda x: get_emoji(x))
data['response'] = data['response'].apply(lambda x: clean_text(x))

data['cntx'] = data['context'].apply(lambda x: x[len(x)-1] if (len(x) > 0) else '')
data['cntx_ht'] = data['cntx'].apply(lambda x: get_hash_tags(x))
data['cntx_emoji'] = data['cntx'].apply(lambda x: get_emoji(x))
data['cntx'] = data['cntx'].apply(lambda x: clean_text(x))

data['is_sarcastic'] = data['label'].apply(lambda x: 1 if(x == "SARCASM") else 0)

data['overall'] = data['cntx'] +" "+ data['response']
print("Training rows: ",len(data))
```

Clean-up functions:

```
def get_hash_tags(txt):
    hastags = re.findall(r"#(\w+)", txt)
    return hastags

def get_emoji(txt):
    return ' '.join(emoji.UNICODE_EMOJI[c] for c in txt if c in emoji.UNICODE_EMOJI)

def clean_text(txt):
    bef_txt = txt
    txt = txt.replace('@USER', '').lstrip().rstrip()
    txt = txt.lower()
    txt = re.sub(r"#(\w+)", "", txt)
    txt = re.sub(r"@(\w+)", "", txt)
    txt = re.sub(r"i'm", "i am", txt)
    txt = re.sub(r"he's", "he is",txt)
    txt = re.sub(r"she's", "she is", txt)
    txt = re.sub(r"i'm", "i am", txt)
    txt = re.sub(r"he's", "he is",txt)
    txt = re.sub(r"she's", "she is", txt)
    txt = re.sub(r"that's", "that is", txt)
    txt = re.sub(r"what's", "what is", txt)
    txt = re.sub(r"where's", "where is", txt)
    txt = re.sub(r"\'ll", " will", txt)
    txt = re.sub(r"\'ve", " have", txt)
    txt = re.sub(r"\'re", " are", txt)
    txt = re.sub(r"\'d", " would", txt)
    txt = re.sub(r"\'ve", " have", txt)
    txt = re.sub(r"won't", "will not", txt)
    txt = re.sub(r"don't", "do not", txt)
    txt = re.sub(r"did't", "did not", txt)
    txt = re.sub(r"did'nt", "did not", txt)
    txt = re.sub(r"can't", "can not", txt)
    txt = re.sub(r"it's", "it is", txt)
    txt = re.sub(r"couldn't", "could not", txt)
    txt = re.sub(r"have't", "have not", txt)
    txt = re.sub('[^a-zA-z0-9\s]','',txt)
    txt = re.sub(r'\d+', '',txt)
    txt = re.sub(r"[,.\"\'!@#$%^&*(){}?/;`~:<>+=-]", "", txt)

    txt = ' '.join([y for y in txt.split() if not y in string.punctuation])
    txt = ' '.join([y for y in txt.split() if not y in stopword])
    txt = ' '.join([lemmatizer.lemmatize(y) for y in txt.split()])
    #txt = ' '.join([stemmer.stem(y) for y in txt.split()]) -- Not using stemmer for this project.
    txt = txt.lstrip().rstrip()

    return txt
```

Data-read and clean-up code (test.jsonl):

```
pred_data = pd.read_json (r'.\data\test.jsonl',encoding='UTF-8', lines=True)

pred_data['response_ht'] = pred_data['response'].apply(lambda x: get_hash_tags(x))
pred_data['response_emoji'] = pred_data['response'].apply(lambda x: get_emoji(x))
pred_data['response'] = pred_data['response'].apply(lambda x: clean_text(x))

pred_data['cntx'] = pred_data['context'].apply(lambda x: x[len(x)-1] if (len(x) > 0) else '')
pred_data['cntx_ht'] = pred_data['cntx'].apply(lambda x: get_hash_tags(x))
pred_data['cntx_emoji'] = pred_data['cntx'].apply(lambda x: get_emoji(x))
pred_data['cntx'] = pred_data['cntx'].apply(lambda x: clean_text(x))

pred_data['overall'] = pred_data['cntx'] + " " + pred_data['response']
print("Prediction to be made for rows: ",len(pred_data))
```

Data preparation for training

After pre-processing for the data is completed, we would need to build a input vector that can be used for training. As currently after pre-processing we only have text data, and we would need to convert it into some numeric form to build the model. So, we would need to build a input vector represented in numeric form for this. There are multiple options for building a training vector. We can tokenize the words into integer's and then choose to use a embedding layer in our network to learn about the embedding vector. or we can choose to use some pre-trained embedding vector. While both approaches were tried as part of this project, I found that pre-trained embedding vector had better results than the self-trained one, so I am using the pre-trained embedding vectors. I am using the pre-trained vector from "GloVe". This embedding layer part is covered in more detail in next section. Here are the steps required to build a input vector from the pre-processed text.

- After text pre-processing is completed, we need to build a word vocabulary which contains all the words that we are using for training and testing as-well.
- The word vocabulary in then tokenized (into integers).
- Then we are going to set the length of the input vector that we are going to feed to our model/network. While looking into pre-processed response and context, the max length of response was around 200 range and for context was around 250 range. So, I am using input length of 200 for response and 250 for context.
- X and X_cntx vector with length 200 and 250 are developed. X is for response and X_cntx is for context. The vector X may look line this [0 0 0 ….. 25 34 26 27], where 0 at the left are '0' padded to make the length of the vector 200. The integer on the vector are the ones that were tokenized form of the word, so each word is converted in specific integer by the tokenize function. Similarly X_cntx is in similar structure, only difference it's its length is 250.

Below is the code screenshot for the steps explained above.

Build a word vocabulary (code)

```
vocabulary = {}
for text in data['overall']:
    for word in text.split(" "):
        if word in vocabulary:
            vocabulary[word] = vocabulary[word] + 1
        else:
            vocabulary[word] = 1
for text in pred_data['overall']:
    for word in text.split(" "):
        if word in vocabulary:
            vocabulary[word] = vocabulary[word] + 1
        else:
            vocabulary[word] = 1
print("Number of words in vocabulary: ", len(vocabulary))
```

```
Number of words in vocabulary:  17980
```

Find max length's for input vector's (Code)

```
data_columns_len = {c: data[c].map(lambda x: len(str(x))).max() for c in data.columns}
pd.Series(data_columns_len).sort_values(ascending =False)
```

```
context           5881
overall            397
cntx               246
response           209
cntx_emoji         184
response_emoji     184
response_ht         71
cntx_ht             69
label               11
is_sarcastic         1
dtype: int64
```

Building Input Vector's (Code)

```
max_features = len(vocabulary) + 1
vec_len = 200
tokenizer = Tokenizer(num_words=max_features, split=' ')
tokenizer.fit_on_texts(vocabulary.keys())
word_index = tokenizer.word_index

X = tokenizer.texts_to_sequences(data['response'].values)
X = pad_sequences(X, vec_len)
X_cntx = tokenizer.texts_to_sequences(data['cntx'].values)
X_cntx = pad_sequences(X_cntx, 250)

print(len(X))
print(len(X[0]))
print(len(X_cntx[0]))
```

```
5000
200
250
```

Model Building and Training

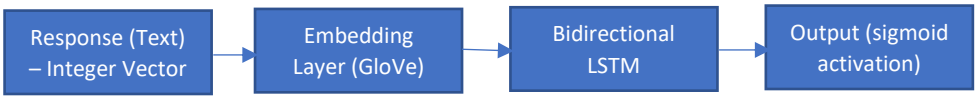
After data clean-up and preparing for the input vector, we now need to represent our word integers as vectors. What this will do is that it will build a vector which will have related word together and unrelated word farther apart, that way the numeric representation of the words becomes more meaningful for training. As previously mentioned, we have two ways to do that. Once is to train that word-vector using training data (Embedding layer) another is to use the pre-trained data. For this project I tried both, at the end the pre-trained model worked better than the self-training. So, I am using the pre-trained data for word-embedding. I am using **GloVe** (Global Vectors for Word Representation) for that. GloVe has multiple files that we can use. For this project I have use the glove twitter data-set with 25d vector. (***glove.twitter.27B.25d.txt***)

Glove site: https://nlp.stanford.edu/projects/glove/

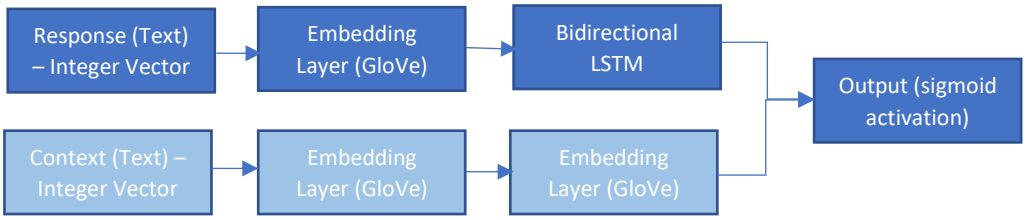
Using GloVe we will be building a embedded layer in our model, what that embedding layer will do is, it will take the input of 1-d integer vector that we build (X, X_cntx) and then convert them into 2-d vector's where each integer is converted into another vector with length 25. That 2-d vector is then passed to the neural network model.

After embedding layer is built we will then build two different model's one with "Response" as an input and another with "Context" as input. Below is the detail diagram of both the model's. As mentioned previously I am using LSTM model for this project.

Model-1:



Model-2:



Below is the details steps done as part of model build and training:

- The input-vector which is a integer conversion from text is first converted into pre-trained word-vector's. I am using GloVe (Global Vector for Word Representation) for this.
- We need to read the GloVe vector from a file. After that all the words that were part of our word vocabulary is been fetched and embedding matrix is built with that.
- The embedding matrix build will be used to create a embedding layer for our neural network model. This embedding layer's job would be to convert the integer vector into the GloVe vector representation, which can then be feed into neural network (LSTM).
- After building the embedding layer, we now build an actual model.
- Frist model, is a simple LSTM model which takes the response as input pass it to embedding layer and then into the LSTM network and finally a output is generated from that model.
- Second model is a combination model, where I have built a two different LSTM model's one for Response as input (similar to first model) and another model is also similar LSTM but with Context as input. Finally these two LSTM's are combined to get the output for this second model.
- After both the model's are built. We now have to prepare the training and testing data-set. As we have prepared the vectors X, X_cntx which has 5000 data-elements previously. We will split this data into 4000 training set and 1000 test-set (This is the test set used in training for validation).
- After the split of training and test-set for validation. We now train both the model's model-1 and model-2 with these data.

Below is the code snapshot for the above explained steps.

Reading GloVe vector from the downloaded files (Code):

```
embeddings_index = {}
embedding_dim = 25
GLOVE_DIR = "..\data\glove.twitter.27B"
f = open(os.path.join(GLOVE_DIR, 'glove.twitter.27B.25d.txt'), encoding = "utf-8")
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))
```

```
Found 1193514 word vectors.
```

Building embedding matrix and Embedding Layer with GloVe vector(Code):

```
embedding_matrix = np.zeros((len(word_index) + 1, embedding_dim))
accp = 0
rej = 0
rej_corr = 0
rej_not_corr = 0
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        accp+=1
        embedding_matrix[i] = embedding_vector
    else:
        rej+=1
        #corr_word = spell(word)
        #embedding_vector = embeddings_index.get(corr_word)
        if embedding_vector is not None:
            print(rej, word, corr_word)
            rej_corr+=1
            embedding_matrix[i] = embedding_vector
        else:
            rej_not_corr+=1
print("Rejected:" , rej, "Accepted:", accp, "Reject Corrected:", rej_corr, "Reject NC:", rej_not_corr)
embedding_layer = Embedding(len(word_index) + 1,
                            embedding_dim,
                            weights=[embedding_matrix],
                            input_length=X.shape[1],
                            trainable=False)
embedding_layer_cntx = Embedding(len(word_index) + 1,
                            embedding_dim,
                            weights=[embedding_matrix],
                            input_length=X_cntx.shape[1],
                            trainable=False)
```

```
Rejected: 1930 Accepted: 16045 Reject Corrected: 0 Reject NC: 1930
```

Building First Model (Code):

```
hidden_units = 128

model_1 = Sequential()
model_1.add(embedding_layer)
model_1.add(Bidirectional(LSTM(hidden_units, dropout=0.2, recurrent_dropout=0.2, return_sequences=False)))
model_1.add(Dense(1, activation='sigmoid'))
model_1.compile(loss='binary_crossentropy', optimizer='adam', metrics=['acc'])
print('Summary of the built model...')
print(model_1.summary())
```

```
Summary of the built model...
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 200, 25)           449400
_________________________________________________________________
bidirectional_1 (Bidirection (None, 256)               157696
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 257
=================================================================
Total params: 607,353
Trainable params: 157,953
Non-trainable params: 449,400
_________________________________________________________________
None
```

Building Second Model (Code):

```
hidden_units = 128
Response_In = Input(shape=(X.shape[1],))
Response = Embedding(len(word_index) + 1, embedding_dim, weights=[embedding_matrix], input_length=X.shape[1],
                            trainable=False)(Response_In)
Response = Bidirectional(LSTM(hidden_units, dropout=0.2, recurrent_dropout=0.2, return_sequences=False))(Response)
Response_Model = Model(inputs=Response_In, outputs=Response)

Cntx_In = Input(shape=(X_cntx.shape[1],))
Cntx = Embedding(len(word_index) + 1, embedding_dim, weights=[embedding_matrix],input_length=X_cntx.shape[1],
                            trainable=False)(Cntx_In)
Cntx = Bidirectional(LSTM(hidden_units, dropout=0.2, recurrent_dropout=0.2, return_sequences=False))(Cntx)
Cntx_Model = Model(inputs=Cntx_In, outputs=Cntx)
combined = concatenate([Response_Model.output, Cntx_Model.output], axis=-1)
combined = Dense(hidden_units, kernel_initializer="he_normal", activation='sigmoid')(combined)
y = Dense(1, activation='sigmoid')(combined)
model_2 = Model(inputs=[Response_Model.input, Cntx_Model.input], outputs=y)
model_2.compile(loss = 'binary_crossentropy', optimizer='adam',metrics = ['accuracy'])
print(model_2.summary())
```

```
Model: "model_3"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            (None, 200)          0
__________________________________________________________________________________________________
input_2 (InputLayer)            (None, 250)          0
__________________________________________________________________________________________________
embedding_3 (Embedding)         (None, 200, 25)      449400      input_1[0][0]
__________________________________________________________________________________________________
embedding_4 (Embedding)         (None, 250, 25)      449400      input_2[0][0]
__________________________________________________________________________________________________
bidirectional_2 (Bidirectional) (None, 256)          157696      embedding_3[0][0]
__________________________________________________________________________________________________
bidirectional_3 (Bidirectional) (None, 256)          157696      embedding_4[0][0]
__________________________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 512)          0           bidirectional_2[0][0]
                                                                 bidirectional_3[0][0]
__________________________________________________________________________________________________
dense_5 (Dense)                 (None, 128)          65664       concatenate_1[0][0]
__________________________________________________________________________________________________
dense_6 (Dense)                 (None, 1)            129         dense_5[0][0]
==================================================================================================
Total params: 1,279,985
Trainable params: 381,185
Non-trainable params: 898,800
```

Training and Test(Validation) data split from training data (Code)

```
Y = data['is_sarcastic'].values
X_train, X_test, X_cntx_train, X_cntx_test, Y_train, Y_test, data_train, data_test = train_test_split(X,
                                                                    X_cntx, Y, data, test_size = 0.2, random_state = 42)
print(X_train.shape,Y_train.shape)
print(X_test.shape,Y_test.shape)
```

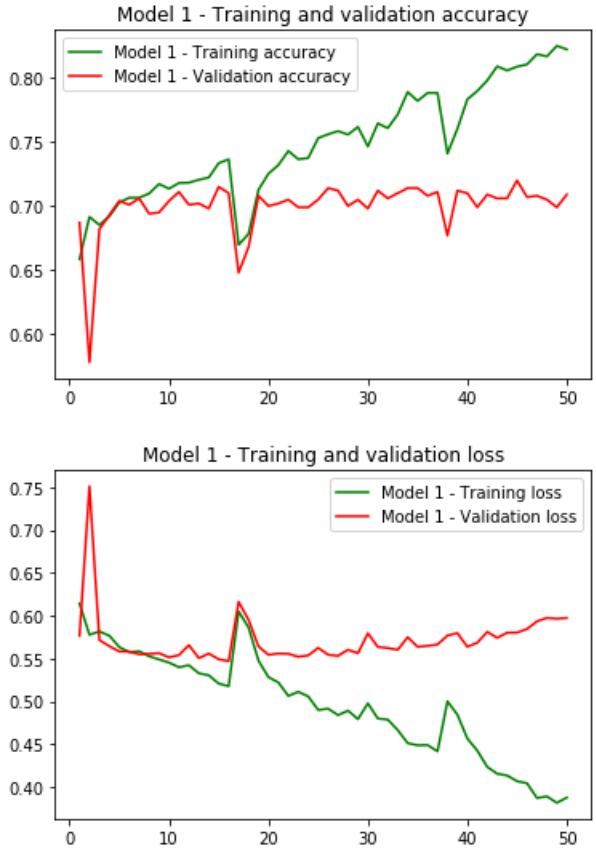
```
(4000, 200) (4000,)
(1000, 200) (1000,)
```

Training Model-1 (Code):

```
batch_size = 32
model_1_train = model_1.fit(X_train, Y_train, epochs = 50, batch_size=batch_size,
                            validation_data=(X_test, Y_test), verbose = 2)
```

```
Train on 4000 samples, validate on 1000 samples
Epoch 1/50
 - 117s - loss: 0.6141 - acc: 0.6585 - val_loss: 0.5765 - val_acc: 0.6870
Epoch 2/50
 - 113s - loss: 0.5777 - acc: 0.6915 - val_loss: 0.7511 - val_acc: 0.5780
Epoch 3/50
 - 113s - loss: 0.5816 - acc: 0.6852 - val_loss: 0.5715 - val_acc: 0.6820
Epoch 4/50
 - 113s - loss: 0.5766 - acc: 0.6920 - val_loss: 0.5644 - val_acc: 0.6930
Epoch 5/50
 - 113s - loss: 0.5634 - acc: 0.7028 - val_loss: 0.5582 - val_acc: 0.7040
Epoch 6/50
 - 113s - loss: 0.5574 - acc: 0.7065 - val_loss: 0.5581 - val_acc: 0.7010
Epoch 7/50
 - 113s - loss: 0.5585 - acc: 0.7065 - val_loss: 0.5549 - val_acc: 0.7060
```

Below is the graph drawn from the training of Model-1:

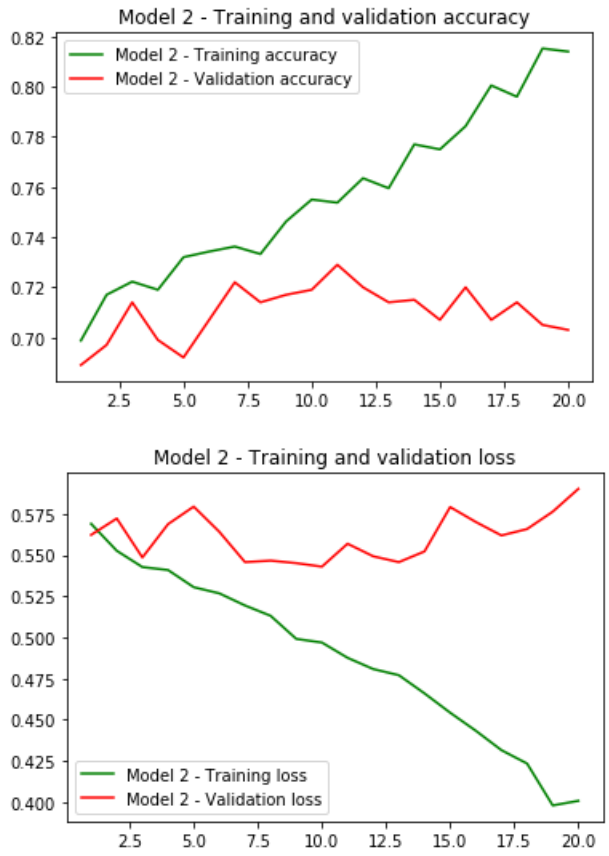


Training Model-2 (Code):

```
model_2_train = model_2.fit([X_train, X_cntx_train], Y_train, epochs = 20, batch_size=batch_size,
            validation_data=([X_test, X_cntx_test], Y_test), verbose = 2)
```

```
Train on 4000 samples, validate on 1000 samples
Epoch 1/20
 - 166s - loss: 0.5689 - accuracy: 0.6988 - val_loss: 0.5622 - val_accuracy: 0.6890
Epoch 2/20
 - 170s - loss: 0.5526 - accuracy: 0.7170 - val_loss: 0.5721 - val_accuracy: 0.6970
Epoch 3/20
 - 168s - loss: 0.5427 - accuracy: 0.7222 - val_loss: 0.5485 - val_accuracy: 0.7140
Epoch 4/20
 - 167s - loss: 0.5409 - accuracy: 0.7190 - val_loss: 0.5687 - val_accuracy: 0.6990
Epoch 5/20
 - 167s - loss: 0.5305 - accuracy: 0.7320 - val_loss: 0.5793 - val_accuracy: 0.6920
Epoch 6/20
 - 167s - loss: 0.5267 - accuracy: 0.7343 - val_loss: 0.5640 - val_accuracy: 0.7070
Epoch 7/20
 - 167s - loss: 0.5194 - accuracy: 0.7362 - val_loss: 0.5457 - val_accuracy: 0.7220
```

Below is the graph drawn from the training of Model-2:



Prediction for Test set

Now, with both the model built and trained with training data, we will now predict the results for the test-set (1800 data set's that were read from train.josnl). Below is the details steps of what we do for prediction:

- As we did for the training set we will first vectorized the text into integers before passing to the model as that is the input to our model. We build both response and context vectors as Z, and Z_cntx.
- After building a vector now we pass the vector (Z) i.e. response only into model_1 for prediction. Similar we pass (Z and Z_cntx both) into second model for prediction. With this we now have two sets of prediction which is from model-1 and model-2.
- Finally we combine the model-1 and model-2 prediction into single column, and then use some threshold defined to predict the result.
- Since we are combining the results form model-1 and model-2 the aggregated value of the output will range from 0 to 2. The half-way cut-off value is 1.0, but instead of using 1.0 as our cut-off we will be using 0.5 as our cut-off to determine if it's sarcastic or not, we are doing this because this will increased our re-call by a lot with very less impact in the precision (that is what I found during the training/testing). This will increase our overall scoring of the model.
- Finally, we will generate a file "answer.txt" with all this prediction for each of the data-set given and then the file is uploaded into GitHub to get score from LiveDataLab.

Below is code for the steps explained above:

Building Input Vector for test data set (Code):

```
Z = tokenizer.texts_to_sequences(pred_data['response'].values)
Z = pad_sequences(Z, vec_len)
Z_cntx = tokenizer.texts_to_sequences(pred_data['cntx'].values)
Z_cntx = pad_sequences(Z_cntx, 250)
```

Predicting using Model_1 and Model_2 (Code):

```
Z_label = model_1.predict(Z)
Z_cntx_label = model_2.predict([Z, Z_cntx])
```

Combine Results from Model_1 and Model_2 (Code):

```
pred_data["score"] = Z_label
pred_data["cntx_score"] = Z_cntx_label
pred_data["total_score"] = pred_data["score"] + pred_data["cntx_score"]

pred_data['pred'] =pred_data["total_score"].apply(lambda x: 'SARCASM' if (x >= 0.5) else 'NOT_SARCASM')
```

Exporting the results (answer.txt) (Code):

```
export_data = pred_data[["id", "pred"]]
export_data.to_csv(path_or_buf='../answer.txt', sep=',',index=False)
```

Prediction Results

There were multiple runs / iteration that was carried out as part of this project to beat the baseline score. The model explained above was able to beat the baseline score, that is combining two different model's and aggregating the result to get the final output. Below is the screenshot from the LiveDataLab that shows the overall score that I had which is above the baseline score.

This score is as of 12/11/2020:

Rank	Username	Submission Number	precision	recall	f1	completed
57	mkhanal2	82	0.5975433526011561	0.9188888888888889	0.7241681260945709	1

I have also uploaded the answers.txt file in the GitHub with the source-code.

Other Approaches

Apart from the approach that was discuss above (which was able to beat the benchmark score), I tried other approaches as-well. Other approaches that I tried were not able to beat the benchmark score.

- Other Approach -1 (LSTM – "response" only model)
    - This approach is similar to the model-1 that I implemented above, but this approach alone was not able to beat the baseline score. The best f1 score for this model was around 0.70 range.
    - I tried this model with both self-training embedded layer and pre-trained embedded layer.
- Other Approach -2 (LSTM – "response" and "context")
    - This approach is similar to the model-2 that I implemented above, but this approach alone was not able to beat the baseline score. The best f1 score for this was also around 0.70 range.
    - I tried this model with both self-training embedded layer and pre-trained embedded layer.
- Other Approach -3 (LSTM – "response" and "emoji" model)
    - This model was based on model-1 above, and also one additional model which was used to train the data with "emoji's" only , since the emoji containing response were limited, so this model was also not able to beat the baseline score. Although this model was better from previous two approaches, the best f1 score for this was around 0.71 range.
    - I tried this model with both self-training embedded layer and pre-trained embedded layer.

Conclusion

After testing through multiple models of LSTM , the combined model with “response” only as input and “response and context” as input was able to beat the baseline score. So, combining different model’s with different inputs and then combining the results of output made model more efficient. Similar we could further tune/iterate through this model , or introduce new features like emoji’s to possibly improve the score for this model. Also, there are other model that can be tried with this data-set , like BERT. Due to the limited time for this project, my work has been only limited to LSTM model explained above.

References

There were lots of learning and references that I took as part of this project, as LSTM / neural-network was kind of new topic for me. I have also taken the code snippet from some of the references list below. Listed below are websites/codes that I took reference and learned from:

- https://adventuresinmachinelearning.com/keras-lstm-tutorial/
- https://machinelearningmastery.com/crash-course-deep-learning-natural-language-processing/
- https://www.aclweb.org/anthology/C16-1231.pdf
- https://machinelearningmastery.com/timedistributed-layer-for-long-short-term-memory-networks-in-python/
- https://www.kaggle.com/mkowoods/deep-learning-lstm-for-tweet-classification
- https://github.com/AniSkywalker/SarcasmDetection
- https://www.kaggle.com/c/tweet-sentiment-extraction/discussion/143281
- https://arxiv.org/pdf/1911.10401.pdf
- https://www.aclweb.org/anthology/2020.figlang-1.11.pdf
- https://github.com/MirunaPislar/Sarcasm-Detection
- https://www.youtube.com/watch?v=pMjT8GIX0co
- https://towardsdatascience.com/lstm-vs-bert-a-step-by-step-guide-for-tweet-sentiment-analysis-ced697948c47?gi=5e1b3ad1bacc
- https://www.pyimagesearch.com/2019/02/04/keras-multiple-inputs-and-mixed-data/
- https://github.com/Suji04/NormalizedNerd/blob/master/Introduction%20to%20NLP/Sarcasm%20is%20very%20easy%20to%20detect%20GloVe%2BLSTM.ipynb
- https://rcciit.org/students_projects/projects/cse/2018/GR6.pdf