lstm_w2v_wiki

April 15, 2021

1 Emotion Classification in texts using LSTM and Word2Vec

1.0.1 Architecture:

(X) Text -> Embedding (W2V pretrained on wikipedia articles) -> Deep Network (LSTM/GRU) -> Fully connected (Dense) -> Output Layer (Softmax) -> Emotion class (Y)

Embedding Layer

- Word Embedding is a representation of text where words that have the similar meaning have a similar representation. We will use 300 dimentional word vectors pre-trained on wikipedia articles. We can also train the w2v model with our data, however our dataset is quite small and trained word vectors might not be as good as using pretrained w2v.

Deep Network

- Deep network takes the sequence of embedding vectors as input and converts them to a compressed representation. The compressed representation effectively captures all the information in the sequence of words in the text. The deep network part is usually an RNN or some forms of it like LSTM/GRU. The dropout can be added to overcome the tendency to overfit, a very common problem with RNN based networks.

Fully Connected Layer

- The fully connected layer takes the deep representation from the RNN/LSTM/GRU and transforms it into the final output classes or class scores. This component is comprised of fully connected layers along with batch normalization and optionally dropout layers for regularization.

Output Layer

- Based on the problem at hand, this layer can have either Sigmoid for binary classification or Softmax for both binary and multi classification output.

1.1 Workflow:

1. Import Data
2. Prepare the input data
3. Import pre-trained W2V
4. Create Neural Network Pipeline

5. Train The Model
6. Evaluate results

Let's start

1.2 1. Import Data

[2]:
```
import pandas as pd
import numpy as np

# text preprocessing
from nltk.tokenize import word_tokenize
import re

# plots and metrics
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix

# preparing input to our model
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical

# keras layers
from keras.models import Sequential
from keras.layers import Embedding, Bidirectional, LSTM, GRU, Dense
```

Defining vector space dimension and fixed input size

[3]:
```
# Number of labels: joy, anger, fear, sadness, neutral
num_classes = 5

# Number of dimensions for word embedding
embed_num_dims = 300

# Max input length (max number of words)
max_seq_len = 500

class_names = ['joy', 'fear', 'anger', 'sadness', 'neutral']
```

Importing our training and testing datasets

[4]:
```
data_train = pd.read_csv('data/data_train.csv', encoding='utf-8')
data_test = pd.read_csv('data/data_test.csv', encoding='utf-8')

X_train = data_train.Text
X_test = data_test.Text
```

```
y_train = data_train.Emotion
y_test = data_test.Emotion

data = data_train.append(data_test, ignore_index=True)
```

[5]:
```
print(data.Emotion.value_counts())
data.head(6)
```

```
joy        2326
sadness    2317
anger      2259
neutral    2254
fear       2171
Name: Emotion, dtype: int64
```

[5]:
```
   Emotion                                               Text
0  neutral   There are tons of other paintings that I thin...
1  sadness  Yet the dog had grown old and less capable , a...
2     fear  When I get into the tube or the train without ...
3     fear  This last may be a source of considerable disq...
4    anger  She disliked the intimacy he showed towards so...
5  sadness  When my family heard that my Mother's cousin w...
```

1.3 2. Prepare input data

To input the data to our NN Model we'll need some preprocessing: 1. Tokenize our texts and count unique tokens 2. Padding: each input (sentence or text) has to be of the same lenght 3. Labels have to be converted to integeres and categorized

Basic preprocessing and tokenization using nltk to double check that sentences are properly split into words. We could also add stopword removal but steps like stemming or lemmatization are not needed since we are using word2vec and words with the same stem can have a different meaning

[6]:
```
def clean_text(data):

    # remove hashtags and @usernames
    data = re.sub(r"(#[\d\w\.]+)", '', data)
    data = re.sub(r"(@[\d\w\.]+)", '', data)

    # tekenization using nltk
    data = word_tokenize(data)

    return data
```

*Making things easier for keras tokenizer

[7]:
```
texts = [' '.join(clean_text(text)) for text in data.Text]

texts_train = [' '.join(clean_text(text)) for text in X_train]
texts_test = [' '.join(clean_text(text)) for text in X_test]
```

[8]:
```
print(texts_train[92])
```

```
a bit ? I 'm extremely annoyed that he did n't phone me when he promised me that
he would ! He 's such a liar .
```

Tokenization + fitting using keras

[9]:
```
tokenizer = Tokenizer()
tokenizer.fit_on_texts(texts)

sequence_train = tokenizer.texts_to_sequences(texts_train)
sequence_test = tokenizer.texts_to_sequences(texts_test)

index_of_words = tokenizer.word_index

# vacab size is number of unique words + reserved 0 index for padding
vocab_size = len(index_of_words) + 1

print('Number of unique words: {}'.format(len(index_of_words)))
```

```
Number of unique words: 12087
```

Padding -> each input has the same length

We defined maximun number of words for our texts and input size to our model has to be fixed - padding with zeros to keep the same input lenght (longest input in our dataset is ~250 words)

[10]:
```
X_train_pad = pad_sequences(sequence_train, maxlen = max_seq_len )
X_test_pad = pad_sequences(sequence_test, maxlen = max_seq_len )

X_train_pad
```

[10]:
```
array([[    0,     0,     0, ...,   119,    51,   345],
       [    0,     0,     0, ...,    37,   277,   154],
       [    0,     0,     0, ...,    16,     2,  1210],
       ...,
       [    0,     0,     0, ...,   876,     4,   909],
       [    0,     0,     0, ...,     1,     6,   117],
       [    0,     0,     0, ..., 10258,   173,    13]], dtype=int32)
```

Categorize labels:

[11]:
```
encoding = {
    'joy': 0,
    'fear': 1,
    'anger': 2,
    'sadness': 3,
    'neutral': 4
}

# Integer labels
y_train = [encoding[x] for x in data_train.Emotion]
y_test = [encoding[x] for x in data_test.Emotion]
```

[12]:
```
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

y_train
```

[12]:
```
array([[0., 0., 0., 0., 1.],
       [0., 0., 0., 1., 0.],
       [0., 1., 0., 0., 0.],
       ...,
       [0., 0., 0., 1., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.]], dtype=float32)
```

1.4 2. Import pretrained word vectors

- Importing pretrained word2vec from file and creating embedding matrix
- We will later map each word in our corpus to existing word vector

[13]:
```
def create_embedding_matrix(filepath, word_index, embedding_dim):
    vocab_size = len(word_index) + 1  # Adding again 1 because of reserved 0␣
 ↪index
    embedding_matrix = np.zeros((vocab_size, embedding_dim))
    with open(filepath) as f:
        for line in f:
            word, *vector = line.split()
            if word in word_index:
                idx = word_index[word]
                embedding_matrix[idx] = np.array(
                    vector, dtype=np.float32)[:embedding_dim]
    return embedding_matrix
```

You can download and import any pre-trained word embeddings. I will use 300 dimentional w2v pre-trained on wikipedia articles. Download fast text english vectors: https://fasttext.cc/docs/en/english-vectors.html

[14]:
```
import urllib.request
import zipfile
import os

fname = 'embeddings/wiki-news-300d-1M.vec'

if not os.path.isfile(fname):
    print('Downloading word vectors...')
    urllib.request.urlretrieve('https://dl.fbaipublicfiles.com/fasttext/
 ↪vectors-english/wiki-news-300d-1M.vec.zip',
                              'wiki-news-300d-1M.vec.zip')
    print('Unzipping...')
    with zipfile.ZipFile('wiki-news-300d-1M.vec.zip', 'r') as zip_ref:
        zip_ref.extractall('embeddings')
    print('done.')

    os.remove('wiki-news-300d-1M.vec.zip')
```

[15]:
```
embedd_matrix = create_embedding_matrix(fname, index_of_words, embed_num_dims)
embedd_matrix.shape
```

[15]:
```
(12088, 300)
```

Some of the words from our corpus were not included in the pre-trained word vectors. If we inspect those words we'll see that it's mostly spelling errors. It's also good to double check the noise in our data f.e different languages or tokenizer errors.

[16]:
```
# Inspect unseen words
new_words = 0

for word in index_of_words:
    entry = embedd_matrix[index_of_words[word]]
    if all(v == 0 for v in entry):
        new_words = new_words + 1

print('Words found in wiki vocab: ' + str(len(index_of_words) - new_words))
print('New words found: ' + str(new_words))
```

```
Words found in wiki vocab: 11442
New words found: 645
```

1.5 3. Create LSTM Pipeline

1.5.1 Embedding Layer

We will use pre-trained word vectors. We could also train our own embedding layer if we don't specify the pre-trained weights

- **vocabulary size:** the maximum number of terms that are used to represent a text: e.g. if we set the size of the "vocabulary" to 1000 only the first thousand terms most frequent in the corpus will be considered (and the other terms will be ignored)

- **the maximum length:** of the texts (which must all be the same length)
- **size of embeddings:** basically, the more dimensions we have the more precise the semantics will be, but beyond a certain threshold we will lose the ability of the embedding to define a coherent and general enough semantic area
- **trainable:** True if you want to fine-tune them while training

[17]:
```
# Embedding layer before the actaul BLSTM
embedd_layer = Embedding(vocab_size,
                         embed_num_dims,
                         input_length = max_seq_len,
                         weights = [embedd_matrix],
                         trainable=False)
```

1.5.2 Model Pipeline

- the input is the first N words of each text (with proper padding)
- the first level creates embedding of words, using vocabulary with a certain dimension, and a given size of embeddings
- LSTM/GRU layer which will receive word embeddings for each token in the tweet as inputs. The intuition is that its output tokens will store information not only of the initial token, but also any previous tokens; In other words, the LSTM layer is generating a new encoding for the original input.
- the output level has a number of neurons equal to the classes of the problem and a "softmax" activation function

You can change GRU to LSTM. The results will be very similar but LSTM might take longer to train.

[19]:
```
# Parameters
gru_output_size = 128
bidirectional = True

# Embedding Layer, LSTM or biLSTM, Dense, softmax
model = Sequential()
model.add(embedd_layer)

if bidirectional:
    model.add(Bidirectional(GRU(units=gru_output_size,
                                dropout=0.2,
                                recurrent_dropout=0.2)))
else:
     model.add(GRU(units=gru_output_size,
                 dropout=0.2,
                 recurrent_dropout=0.2))

model.add(Dense(num_classes, activation='softmax'))
```

[20]:
```
model.compile(loss = 'categorical_crossentropy', optimizer = 'adam', metrics =␣
 ↪['accuracy'])
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 500, 300)          3626400
_________________________________________________________________
bidirectional_2 (Bidirection (None, 256)               329472
_________________________________________________________________
dense_2 (Dense)              (None, 5)                 1285
=================================================================
Total params: 3,957,157
Trainable params: 330,757
Non-trainable params: 3,626,400
_________________________________________________________________
```

1.6 3. Train the Model

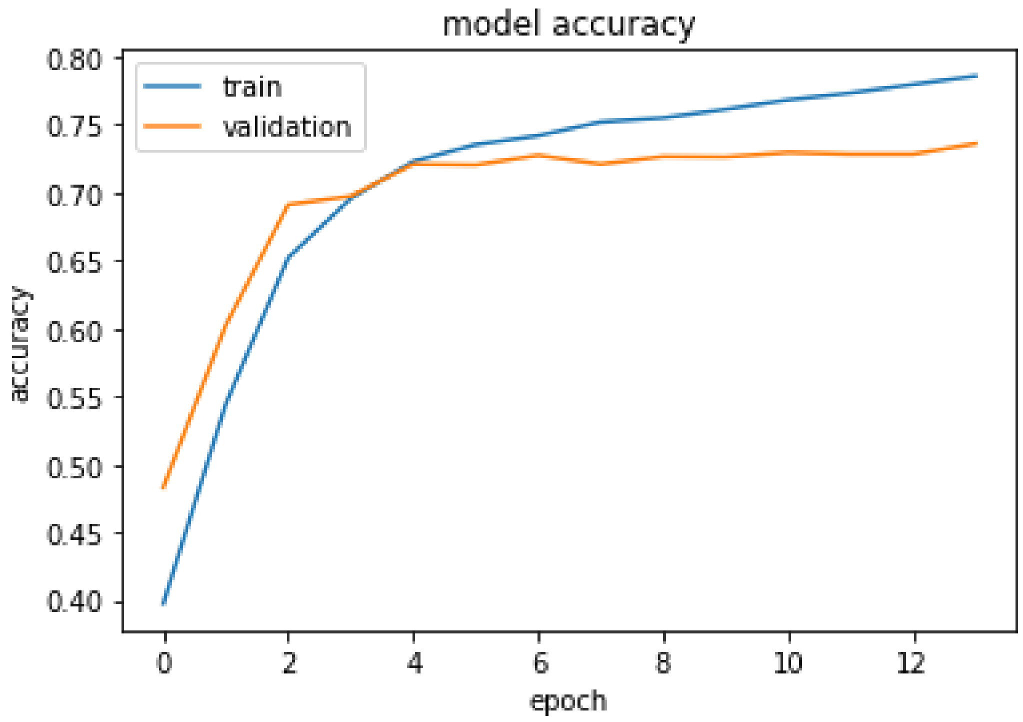
- **validation data**: use validation_split in order to estimate how well your model has been trained and adjust parameters or add dropout layers. After that we will train the model using the complete train set.
- **epochs**: 15 **batch_size**: 128

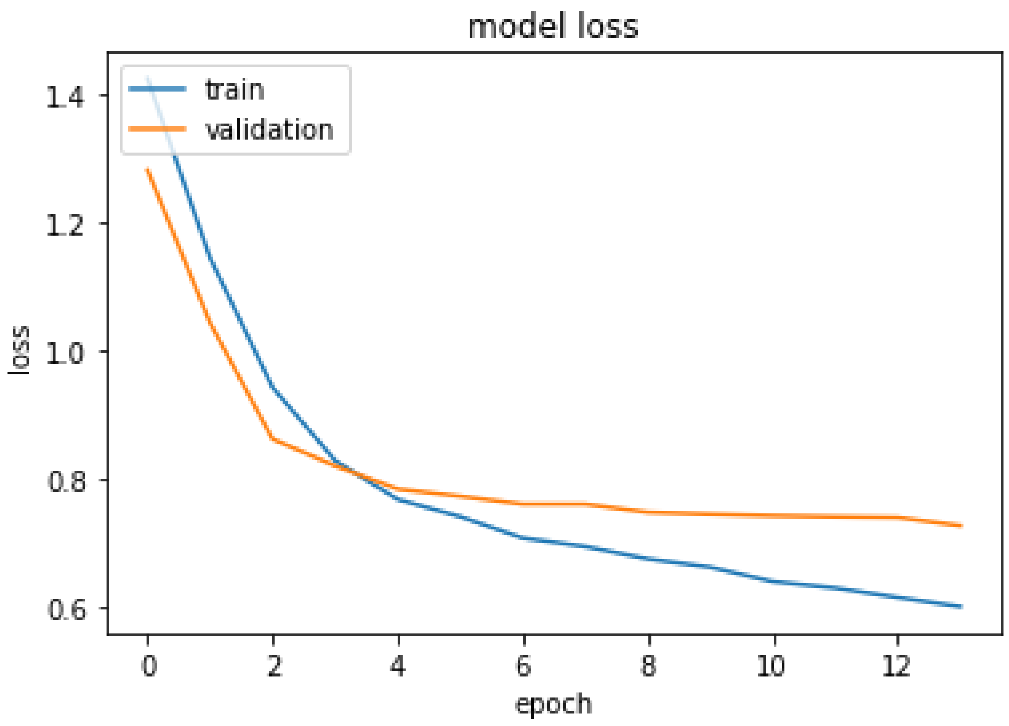
[21]:
```
batch_size = 128
epochs = 15

hist = model.fit(X_train_pad, y_train,
                 batch_size=batch_size,
                 epochs=epochs,
                 validation_data=(X_test_pad,y_test))
```

```
WARNING:tensorflow:From /anaconda3/lib/python3.7/site-
packages/tensorflow/python/ops/math_ops.py:3066: to_int32 (from
tensorflow.python.ops.math_ops) is deprecated and will be removed in a future
version.
Instructions for updating:
Use tf.cast instead.
Train on 7934 samples, validate on 3393 samples
Epoch 1/14
7934/7934 [==============================] - 178s 22ms/step - loss: 1.4235 -
acc: 0.3975 - val_loss: 1.2806 - val_acc: 0.4831
Epoch 2/14
7934/7934 [==============================] - 168s 21ms/step - loss: 1.1437 -
acc: 0.5441 - val_loss: 1.0423 - val_acc: 0.6021
Epoch 3/14
7934/7934 [==============================] - 173s 22ms/step - loss: 0.9415 -
```

```
acc: 0.6518 - val_loss: 0.8609 - val_acc: 0.6908
Epoch 4/14
7934/7934 [==============================] - 175s 22ms/step - loss: 0.8278 -
acc: 0.6952 - val_loss: 0.8196 - val_acc: 0.6970
Epoch 5/14
7934/7934 [==============================] - 173s 22ms/step - loss: 0.7671 -
acc: 0.7226 - val_loss: 0.7833 - val_acc: 0.7206
Epoch 6/14
7934/7934 [==============================] - 164s 21ms/step - loss: 0.7402 -
acc: 0.7349 - val_loss: 0.7720 - val_acc: 0.7200
Epoch 7/14
7934/7934 [==============================] - 163s 21ms/step - loss: 0.7068 -
acc: 0.7415 - val_loss: 0.7597 - val_acc: 0.7271
Epoch 8/14
7934/7934 [==============================] - 164s 21ms/step - loss: 0.6935 -
acc: 0.7516 - val_loss: 0.7597 - val_acc: 0.7206
Epoch 9/14
7934/7934 [==============================] - 164s 21ms/step - loss: 0.6742 -
acc: 0.7546 - val_loss: 0.7470 - val_acc: 0.7262
Epoch 10/14
7934/7934 [==============================] - 164s 21ms/step - loss: 0.6612 -
acc: 0.7609 - val_loss: 0.7440 - val_acc: 0.7259
Epoch 11/14
7934/7934 [==============================] - 163s 21ms/step - loss: 0.6388 -
acc: 0.7678 - val_loss: 0.7416 - val_acc: 0.7291
Epoch 12/14
7934/7934 [==============================] - 182s 23ms/step - loss: 0.6288 -
acc: 0.7729 - val_loss: 0.7400 - val_acc: 0.7280
Epoch 13/14
7934/7934 [==============================] - 186s 23ms/step - loss: 0.6139 -
acc: 0.7792 - val_loss: 0.7389 - val_acc: 0.7280
Epoch 14/14
7934/7934 [==============================] - 196s 25ms/step - loss: 0.5997 -
acc: 0.7854 - val_loss: 0.7262 - val_acc: 0.7356
```

[22]:
```
#  "Accuracy"
plt.plot(hist.history['acc'])
plt.plot(hist.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()

# "Loss"
plt.plot(hist.history['loss'])
plt.plot(hist.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```





1.7 4. Evaluation

[23]:
```
predictions = model.predict(X_test_pad)
predictions = np.argmax(predictions, axis=1)
predictions = [class_names[pred] for pred in predictions]
```

[24]:
```
print("Accuracy: {:.2f}%".format(accuracy_score(data_test.Emotion, predictions)␣
 ↪* 100))
print("\nF1 Score: {:.2f}".format(f1_score(data_test.Emotion, predictions,␣
 ↪average='micro') * 100))
```

```
Accuracy: 73.56%

F1 Score: 73.56
```

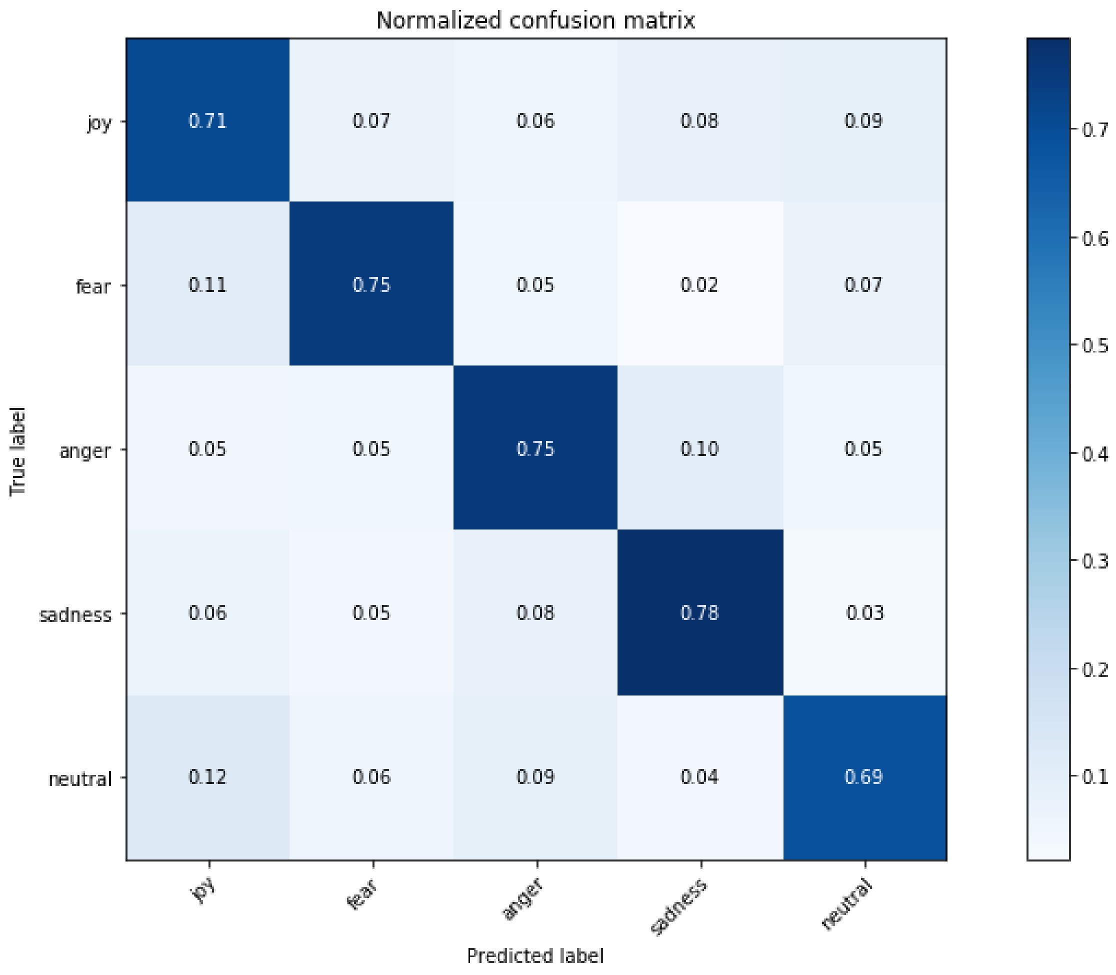
Plotting confusion Matrix:

[25]:
```
def plot_confusion_matrix(y_true, y_pred, classes,
                          normalize=False,
                          title=None,
                          cmap=plt.cm.Blues):
    '''
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    '''
    if not title:
        if normalize:
            title = 'Normalized confusion matrix'
        else:
            title = 'Confusion matrix, without normalization'

    # Compute confusion matrix
    cm = confusion_matrix(y_true, y_pred)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    fig, ax = plt.subplots()

    # Set size
    fig.set_size_inches(12.5, 7.5)
    im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
    ax.figure.colorbar(im, ax=ax)
    ax.grid(False)

    # We want to show all ticks...
    ax.set(xticks=np.arange(cm.shape[1]),
           yticks=np.arange(cm.shape[0]),
           # ... and label them with the respective list entries
           xticklabels=classes, yticklabels=classes,
           title=title,
           ylabel='True label',
           xlabel='Predicted label')

    # Rotate the tick labels and set their alignment.
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
             rotation_mode="anchor")

    # Loop over data dimensions and create text annotations.
    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            ax.text(j, i, format(cm[i, j], fmt),
                    ha="center", va="center",
                    color="white" if cm[i, j] > thresh else "black")
    fig.tight_layout()
    return ax
```

[26]:
```
print("\nF1 Score: {:.2f}".format(f1_score(data_test.Emotion, predictions,␣
 ↪average='micro') * 100))
```

```
# Plot normalized confusion matrix
plot_confusion_matrix(data_test.Emotion, predictions, classes=class_names,␣
 ↪normalize=True, title='Normalized confusion matrix')
plt.show()
```

```
F1 Score: 73.56
```



Lets try other inputs:

```
[27]: print('Message: {}\nPredicted: {}'.format(X_test[4], predictions[4]))
```

```
Message: My boyfriend didn't turn up after promising that he was coming.
Predicted: sadness
```

```
[28]: import time

      message = ['delivery was hour late and my pizza was cold!']

      seq = tokenizer.texts_to_sequences(message)
      padded = pad_sequences(seq, maxlen=max_seq_len)

      start_time = time.time()
      pred = model.predict(padded)

      print('Message: ' + str(message))
      print('predicted: {} ({:.2f} seconds)'.format(class_names[np.argmax(pred)],␣
       ↪(time.time() - start_time)))
```

```
Message: ['delivery was hour late and my pizza was cold!']
predicted: anger (0.05 seconds)
```

2 Done

Save the model for later use

```
[29]: # creates a HDF5 file 'my_model.h5'
      model.save('models/biLSTM_w2v.h5')
```

```
[30]: from keras.models import load_model
      predictor = load_model('models/biLSTM_w2v.h5')
```