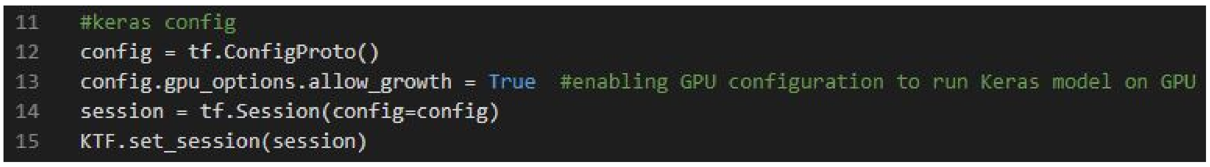
Basically the *lstm_keras.py* script leverages on Keras as the high-level API that runs Tensorflow as the backend. The methodology for Identifying and classifying toxic online comments is to use GloVe.

===



```
11  #keras config
12  config = tf.ConfigProto()
13  config.gpu_options.allow_growth = True  #enabling GPU configuration to run Keras model on GPU
14  session = tf.Session(config=config)
15  KTF.set_session(session)
```

In line 12, *config = tf.ConfigProto()*, it creates a Tensorflow configuration object.

In line 13, it is just enabling GPU resources on your computer to run the Keras model. The *allow_growth* is set to True here, it is to allocate only as much GPU on runtime allocations. What it does it that it slowly allocates more and more GPU memory as the Keras session is being run.

When we use Tensorflow, we must create a session. A *tf.Session()* is basically creating a Session object that encapsulates the environment in which Tensor objects are run. A session allows the execution of graphs, allocates resources in your computer and holds the values of intermediate results and variables. In line 14, the *session* in *session = tf.Session(config=config)* is the Session object created by *tf.Session()* constructor.

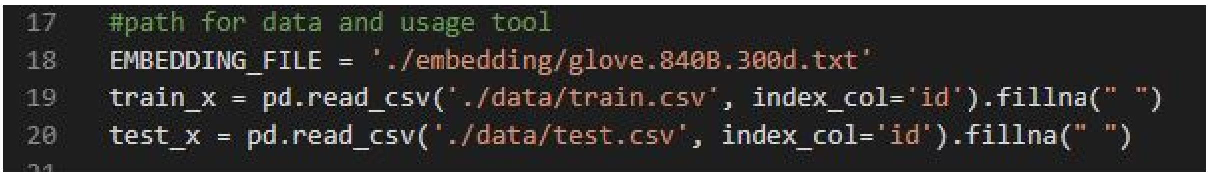
Lastly in line 15, *KTF.set_session(session),* a Keras session with Tensorflow running as backend is declared here, where KTF was earlier imported from the Keras library in line 9 *import keras.backend.tensorflow_backend as KTF*.

===

What is GloVe?

Short for "Global Vectors", it is a word embedding from text method that proposes **global matrix factorization** and **local context window methods.**

GloVe is a method that creates a vector space models of word semantics, which is called "word embeddings".

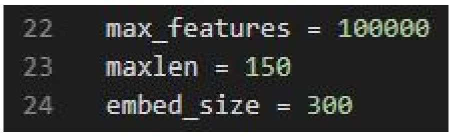


```
17  #path for data and usage tool
18  EMBEDDING_FILE = './embedding/glove.840B.300d.txt'
19  train_x = pd.read_csv('./data/train.csv', index_col='id').fillna(" ")
20  test_x = pd.read_csv('./data/test.csv', index_col='id').fillna(" ")
```

To use this method, we have to utilise the embedding file in the form of .txt file called *glove.840B.300d.txt* (can be downloaded from https://www.kaggle.com/takuok/glove840b300dtxt), and this is called in line 18, EMBEDDING_FILE = './embedding/glove.840B.300d.txt'.

In machine learning, we need 2 sets of datasets – train set and test set. *train_x* is our train set while *test_x* is our test set. In line 19 and 20, they basically use pd, which is actually the pandas library for providing data analysis tools, to read the respective csv files. The *fillna(“ “)* replace all NaN elements with " " whitespaces. What are NaN? They are values that are 'undefined', or just null or missing data.

==



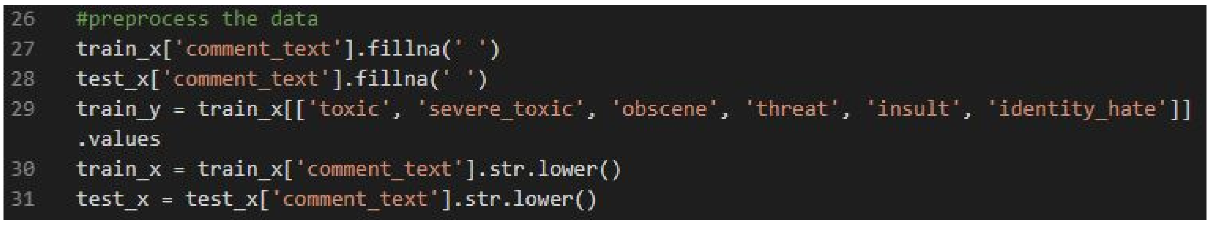
```
22    max_features = 100000
23    maxlen = 150
24    embed_size = 300
```

In line 22, *max_features* is the maximum input dimension, which in this case is the maximum number of words.

In line 23, *maxlen* is declared for an input as argument later in *pad_sequences()* function in line 41 and 42.

In line 24, *embed_size* is the output dimension.

==



```
26    #preprocess the data
27    train_x['comment_text'].fillna(' ')
28    test_x['comment_text'].fillna(' ')
29    train_y = train_x[['toxic', 'severe_toxic', 'obscene', 'threat', 'insult', 'identity_hate']]
      .values
30    train_x = train_x['comment_text'].str.lower()
31    test_x = test_x['comment_text'].str.lower()
```

Just like in any other machine learning/data science project, we must preprocess the dataset to transform raw data into an understandable format, because read-world data is often incomplete and inconsistent for training and testing.

In line 27 and 28, any NaN/null/missing data values in the “comment_text” column will be replaced by a ‘ ‘ whitespace, which is the fillna(‘ ‘) function.

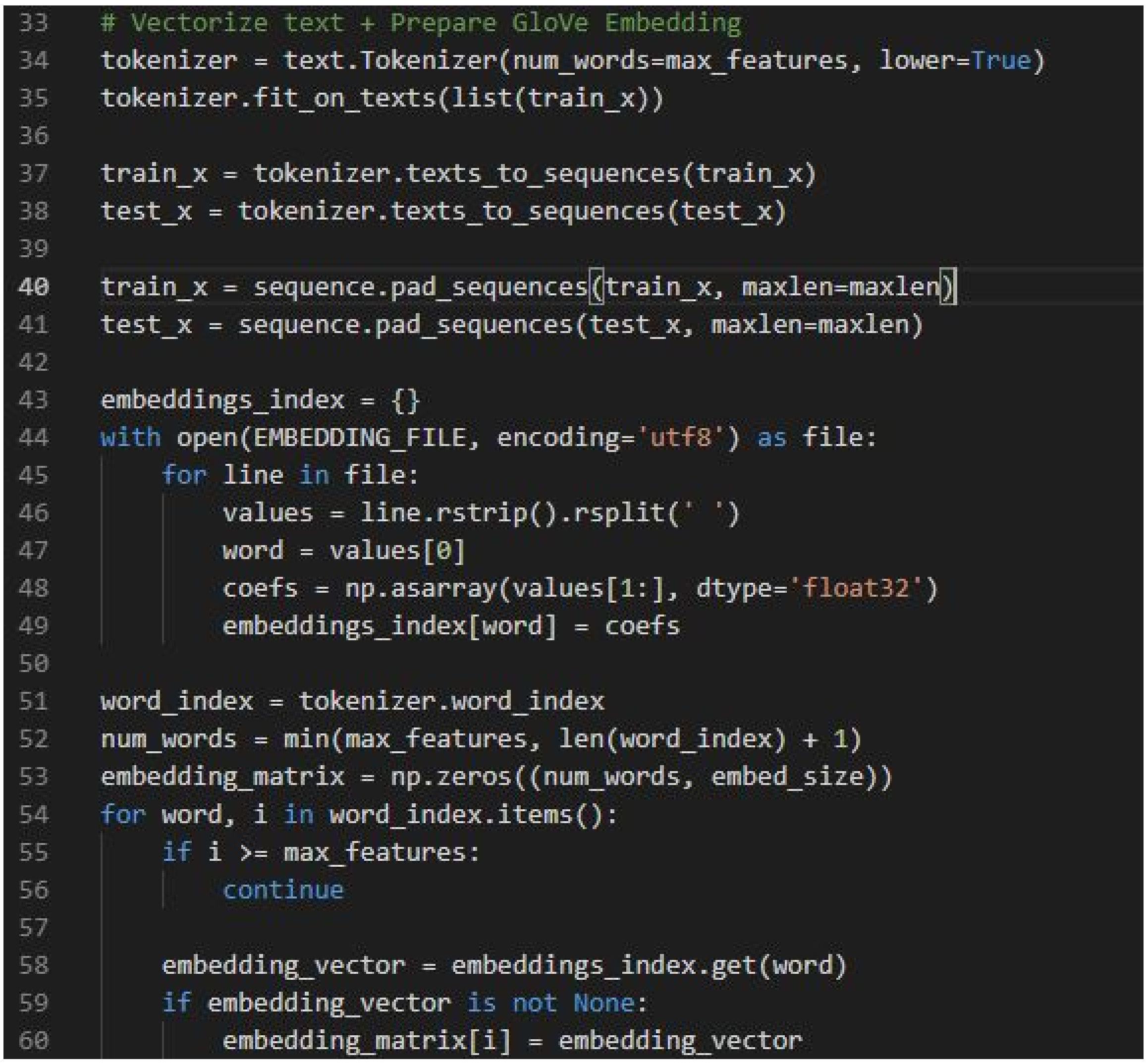
In line 29, *train_y* contains all the values in the respective columns in *train_x*. If you’re wondering what are train_x and train_x, you can see x as x-direction (row) and y as y-direction (column). So that means train_x contains just the headings ('comment_text', 'toxic', 'severe_toxic', 'obscene', 'threat', 'insult', 'identity_hate'), while train_y contains all the values in the respective heading.

In line 30 and 31, all values under ‘comment_text’ heading are set to lowercase for easier data processing (e.g. “HelloWorld” → “hello world”).

==

Here comes the most complex part, which is the most technical part of this Python script as it is based on the GloVe Embedding methodology.

I believe you do not necessarily have to understand line by line for this part as this is too technical. But I will try to explain as best I can.



```python
33  # Vectorize text + Prepare GloVe Embedding
34  tokenizer = text.Tokenizer(num_words=max_features, lower=True)
35  tokenizer.fit_on_texts(list(train_x))
36
37  train_x = tokenizer.texts_to_sequences(train_x)
38  test_x = tokenizer.texts_to_sequences(test_x)
39
40  train_x = sequence.pad_sequences(train_x, maxlen=maxlen)
41  test_x = sequence.pad_sequences(test_x, maxlen=maxlen)
42
43  embeddings_index = {}
44  with open(EMBEDDING_FILE, encoding='utf8') as file:
45      for line in file:
46          values = line.rstrip().rsplit(' ')
47          word = values[0]
48          coefs = np.asarray(values[1:], dtype='float32')
49          embeddings_index[word] = coefs
50
51  word_index = tokenizer.word_index
52  num_words = min(max_features, len(word_index) + 1)
53  embedding_matrix = np.zeros((num_words, embed_size))
54  for word, i in word_index.items():
55      if i >= max_features:
56          continue
57
58      embedding_vector = embeddings_index.get(word)
59      if embedding_vector is not None:
60          embedding_matrix[i] = embedding_vector
```

As declared earlier in Line 5 *from keras.preprocessing import text, sequence*, Line 34 uses the *text.Tokenizer* library, and what it does is to chop up the words in each row into different pieces. Basically, slicing the words in the “comment_text” row into individual words. The argument in *text.Tokenizer(num_words=max_features, lower=True)* , says that the maximum number of words in each row is max_features (100000), and all in lowercase. In line 35, the *fit_on_text()* method creates the vocabulary index based on word frequency, and a dictionary of key:value pair is created, where lower integer means more frequent word. For example a sentence like “Today is a hot day”, the word_index[“Today”]=0, word_index[“hot”]=3.

In line 37 and 38, *text_to_sequences()* method transforms each word in a sentence to a sequence of integers. Basically, it takes each word in the text and replaces it with its corresponding integer value from the word_index mentioned in the previous paragraph.

In line 40 and 41, sequence processing occurs here. The sequence comes from the library declared in line 5 *from keras.preprocessing import text, sequence*. What it *sequence.pad_sequences()* does is to take in a sequence of data-points gathered at equal intervals, along with time series parameters such as stride, length of history, etc., to produce batches for training/validation.

In line 43, *embeddings_index* is a dictionary declared (key:value pair).

From line 44 to 49, this is the preparation of the GloVe Embedding, where we extract the keys and values line by line in the *EMBEDDING_FILE* text file and store them into the *embeddings_index* dictionary.

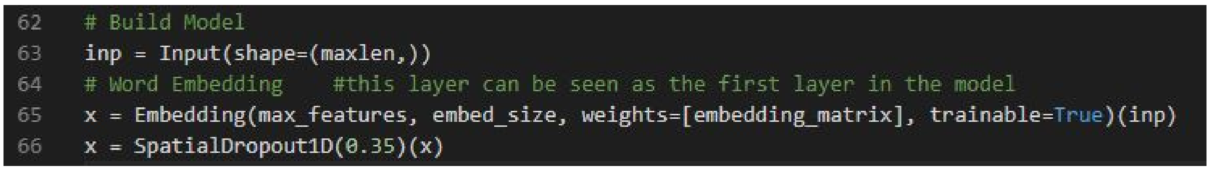
In line 52, *min(max_features, len(word_index) + 1)* is just taking the smaller value between these 2 arguments. The smaller or minimum value is then set as the value for *num_words.*

In line 53, *embedding_matrix = np.zeros((num_words, embed_size))* sets all the values to zeros with *num_words* and *embed_size* as the dimension (row x column → *num_words* x *embed_size*). The purpose of doing this is to initialize the *embedding_matrix* variable, usually we set to all zeros.

From line 54 to 60, a for loop is used to iterate every key and value pair in the *word_index* dictionary and then set those values into *embedding_matrix* dictionary*,* which was previously declared as all zeroes.

embedding_matrix will be used later on for the building of Keras model.

===



```
62  # Build Model
63  inp = Input(shape=(maxlen,))
64  # Word Embedding    #this layer can be seen as the first layer in the model
65  x = Embedding(max_features, embed_size, weights=[embedding_matrix], trainable=True)(inp)
66  x = SpatialDropout1D(0.35)(x)
```

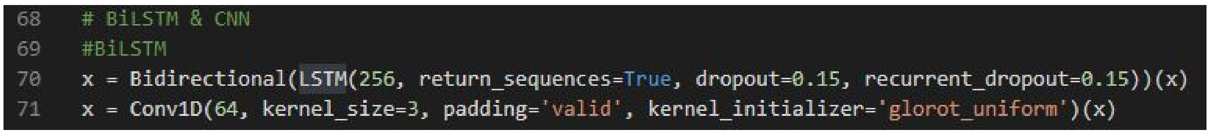
This is where the model is created.

In line 63, the input layer is created, with the *maxlen (*set as 150*)* as the number of inputs.

In line 65, this is supposedly the first layer of our model. The *Embedding()* function turns the indexes (positive integers) into dense vectors of fixed size (*embed_size*).

In line 66, *SpatialDropout1D()* is a method that does Dropout. A dropout in machine-learning is a method that prevents a neural network from overfitting. So usually it does dropout with a probability of say 0.5. It means at every iteration, only 0.5 of the nodes chosen by random in each layer are working. However, for *SpatialDropout1D()*, it is slightly different from the conventional Dropout. It drops the entire 1-dimensional (hence the name 1D) feature maps instead of individual nodes. In this case 0.35 is the probability of each feature map getting chosen to run.

===



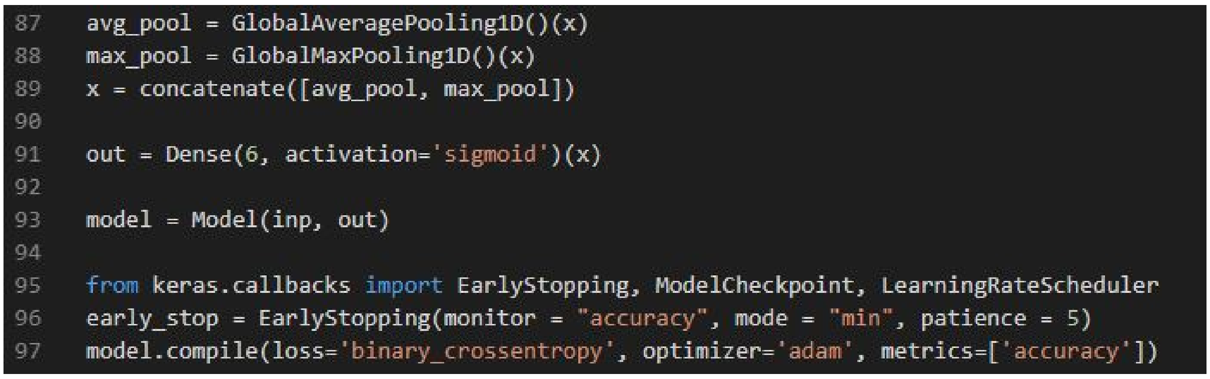
```
68    # BiLSTM & CNN
69    #BiLSTM
70    x = Bidirectional(LSTM(256, return_sequences=True, dropout=0.15, recurrent_dropout=0.15))(x)
71    x = Conv1D(64, kernel_size=3, padding='valid', kernel_initializer='glorot_uniform')(x)
```

BiLSTM is an extension of LSTM. The difference is that BiLSTM has 2 networks, one accessing past information in the forward direction while the other one accessing future information in the reverse direction.

In line 70, BiLSTM is created in *Bidirectional(LSTM(256, return_sequences=True, dropout=0.15, recurrent_dropout=0.15))(x)*, where there are 256 channels (inputs), dropout probability is 0.15.

In line 71, *Conv1D(64, kernel_size=3, padding='valid', kernel_initializer='glorot_uniform')(x)*, it is combining the earlier created BiLSTM with CNN, where Conv1D() is created. Conv1D is a 1-dimensioanl CNN (Convolutional Neural Network).

==



```
87    avg_pool = GlobalAveragePooling1D()(x)
88    max_pool = GlobalMaxPooling1D()(x)
89    x = concatenate([avg_pool, max_pool])
90
91    out = Dense(6, activation='sigmoid')(x)
92
93    model = Model(inp, out)
94
95    from keras.callbacks import EarlyStopping, ModelCheckpoint, LearningRateScheduler
96    early_stop = EarlyStopping(monitor = "accuracy", mode = "min", patience = 5)
97    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

In line 87 and 88, you need to understand the idea of pooling. Pooling is often used in CNN for object detect, basically getting the maximum or average values out of the pool of values. *GlobalAveragingPooling1D()* is the average value from the pool of values.

GlobalMaxPooling1D() is the maximum value from the pool of values.

In line 89, *concatenate()* is to merge layers with the *avg_pool* and *max_pool* that were created earlier.

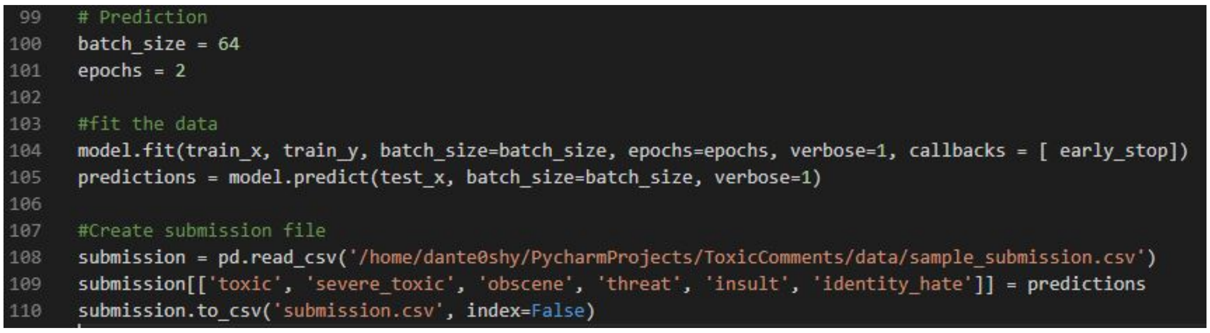
In line 91, the *Dense()* function applies the activation function of type ‘sigmoid’ to the output layer, x, with 6 possible outcomes ('toxic', 'severe_toxic', 'obscene', 'threat', 'insult', 'identity_hate').

In line 93, it’s just the creation of the model with the Model class with *inp* as the input layer and *out* as the output layer.

In line 96, *EarlyStopping()* function basically stopping the training once your validation accuracy starts to decrease. The *patience* here means that it can only tolerate 5 epochs of consecutive decrease in accuracy before stopping.

In line 97, the model is compiled with the *binary_crossentropy* (a method to calculate loss), *adam* as an optimizer to optimize the neural network.

==



```
 99  # Prediction
100  batch_size = 64
101  epochs = 2
102
103  #fit the data
104  model.fit(train_x, train_y, batch_size=batch_size, epochs=epochs, verbose=1, callbacks = [ early_stop])
105  predictions = model.predict(test_x, batch_size=batch_size, verbose=1)
106
107  #Create submission file
108  submission = pd.read_csv('/home/dante0shy/PycharmProjects/ToxicComments/data/sample_submission.csv')
109  submission[['toxic', 'severe_toxic', 'obscene', 'threat', 'insult', 'identity_hate']] = predictions
110  submission.to_csv('submission.csv', index=False)
```

In line 100 and 101, the batch_size and epochs are created. Batch size is the number of samples that will be passed through the network at each time (going by batches). So in this case, batch_size = 64, in each batch, there are 64 inputs passing through the network at each time. For epoch, 1 epoch means 1 forward pass and 1 backward pass of all the training samples. So for epochs = 2, it is required that the training is only done after 2 forward pass and 2 backward pass of all the training samples.

In line 104 and 105, this is where the neural network starts to train where the model.fit() is a function that's trying to fit the data onto a graph, and based on the outputs of the graph, the model predicts the best outcome from the 6 possible outcomes.

Lastly, line 107 to 110 are just creating a csv file to store our results based on the predictions returned by the model we have trained.