Formalizing Informal Text using Natural Language Processing

In [2]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import joblib
import re
import os
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from nltk.translate.bleu_score import sentence_bleu
import nlpaug.augmenter.word as naw
import nlpaug.augmenter.sentence as nas
title_font = {'family': 'serif', 'color': 'darkred', 'weight': 'bold', 'size': 18}
label_font = {'family': 'Arial', 'weight': 'normal', 'size': 16}
import warnings
warnings.filterwarnings("ignore")
%matplotlib inline
```

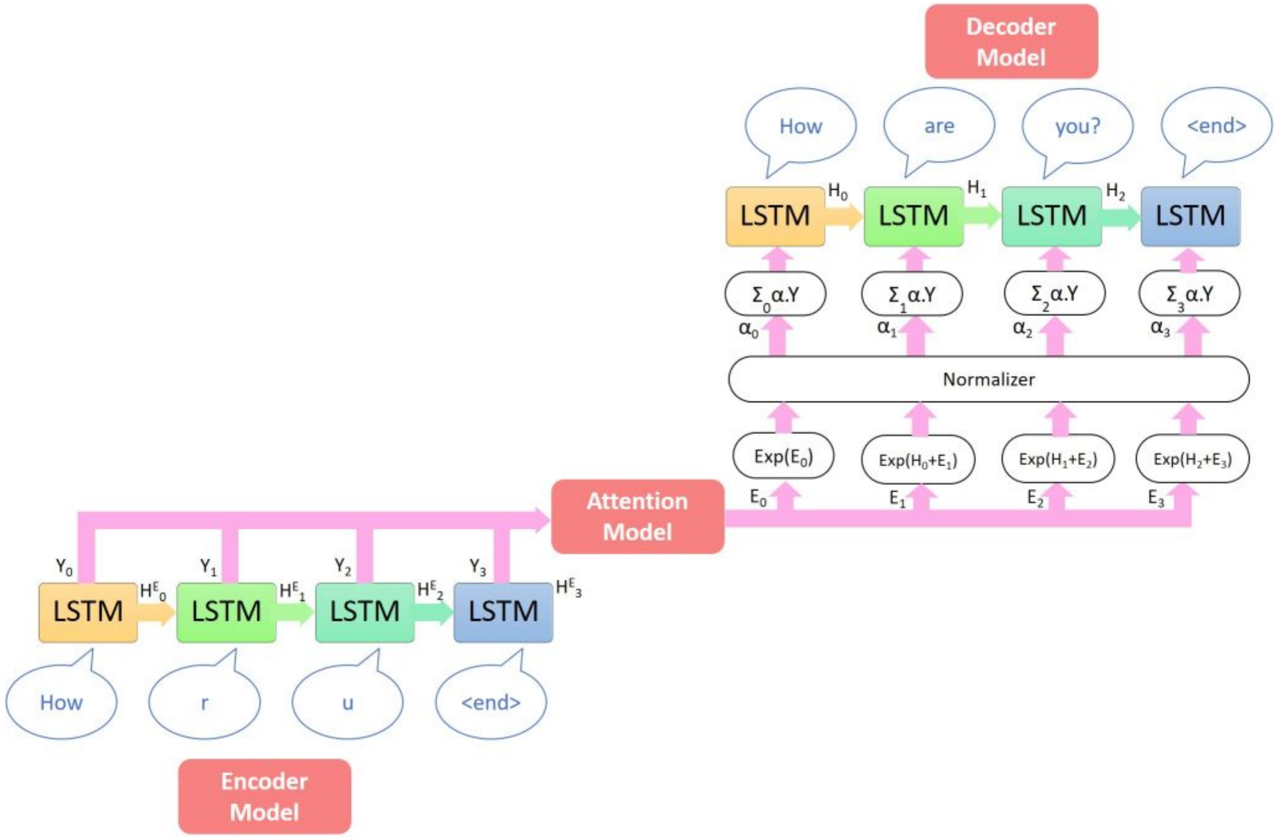
1. Attention Model:

While simple encoder decoder seq2seq model works well for shorter sequences, it badly struggles for longer sequences. This is because of the fact that output token at a particular timestep might be dependent on a token parsed by the encoder a while back. But decoder model only gets to know the output token of the current step. This is where Attention model comes in. It introduces a simple architecture between encoder and decoder to enable the decoder to consider weighted outputs of encoder of all the previous timesteps.

In [1]:

```
from IPython.display import Image
Image(filename = "Attention.jpg")
```

Out[1]:



The figure above shows the mechanism of attention based encoder decoder model. As you can see, the encoder model only differs from a simple encoder decoder model in that it generates an output for each timestep alongwith the hidden state. All the encoder outputs are then fed to the attention model, where the weights corresponding to all the encoder outputs are calculated to enable the decoder to focus on certain tokens while making predictions. The input to the decoder is then computed by weighing the ground truth token with exponent of a concatenated output of hidden state at previous timestep and attention weights to make a prediction.

The important part of attention model is to calculate the weights of output encoder tokens also known as attention weights. These weights are computed by using specific scoring functions. We will consider three types of scoring functions in this case study namely Dot, General and Concat.

$$
\begin{aligned}
H_t &= \sum_i \alpha \,.\, Y && && (1)\\
\alpha_t &= \frac{exp(E_t)}{\sum exp(E_t)} && && (2)\\
& && && (3)\\
E_t &= {H_t}^T\, Y_i && \text{Dot} && (4)\\
&= {H_t}^T\ W\, Y_i && \text{General} && (5)\\
&= v^T\, tanh(W\,[H_t,\, Y_i]) && \text{Concat} && (6)
\end{aligned}
$$

Hence, in total we will train three Attention based Encoder Decoder models using Dot, General and Concat scoring functions.

2. Loading and Preprocessing Data:

As we saw with simple encoder decoder model, the data scarcity is hampering the performance badly. Hence, here we will try data augmnetation techniques as well.

2.1. Loading data:

First, we will load the dataset and won't preprocess it here as we have to augment it as well.

In [3]:

```
# Reading the file
f = open("en2cn-2k.en2nen2cn", "r", encoding = 'utf-8')
text = f.read()
# Removing last instance after splitting as it is empty string
text = text.split('\n')[:-1]
# Creating the pandas dataframe
data = [[text[i], text[i+1]] for i in list(range(0, 6000, 3))]
df = pd.DataFrame(data, columns = ['Informal text', 'Formal text'])
df.head()
```

Out[3]:

	Informal text	Formal text
0	U wan me to "chop" seat 4 u nt?	Do you want me to reserve seat for you or not?
1	Yup. U reaching. We order some durian pastry a...	Yeap. You reaching? We ordered some Durian pas...
2	They become more ex oredi... Mine is like 25.....	They become more expensive already. Mine is li...
3	I'm thai. what do u do?	I'm Thai. What do you do?
4	Hi! How did your week go? Haven heard from you...	Hi! How did your week go? Haven't heard from y...

2.2. Augmenting the Dataset:

For data augmentation, we can use `nlpaug` library. For our purpose, synonym augmentation and spelling augmentation are the suitable techniques. For each formal sentence in the dataset, we will first add synonym augmented pairs to get 4000 instances. On top of that, we will apply spelling augmnetations to get 8000 instances in total.

In [4]:

```
# Applying Synonym augmentation
aug = naw.SynonymAug(aug_src = 'wordnet')
for text in df['Formal text'].values:
    augmented = pd.DataFrame({"Informal text":[aug.augment(text)], "Formal text":[text]
    df = df.append(augmented, ignore_index = True)
# Applying Spelling augmentation
aug = naw.SpellingAug()
for text in df['Formal text'].values:
    augmented = pd.DataFrame({"Informal text":[aug.augment(text)], "Formal text":[text]
    df = df.append(augmented, ignore_index = True)
df.tail()
```

Out[4]:

	Informal text	Formal text
7995	Hmm. I thinnk I usually bock on weekends. It d...	Hmm. I think I usually book on weekends. It de...
7996	Can you ask there whether they have for any sm...	Can you ask them whether they have for any sms...
7997	We are nier Coca already.	We are near Coca already.
7998	Hall elevem. Got lectures. And forget about co...	Hall eleven. Got lectures. And forget about co...
7999	I bring for you. ia can dont'n promese you 100...	I bring for you. I can not promise you 100% to...

2.3. Preprocessing the Dataset:

The models we will design are known as sequence to sequence models as we are providing a text sequence as input and expect the text sequence as output. For that, the input to the encoder should be encoded with start of sentence and end of sentence tokens as it will enable encoder to know span of each sentence. We can use '<' and '>' tokens for initiation and termination respectively. For decoder, input should be appended with '<' token at the beginning and output should be appended with '>' token at the end.

In [5]:

```
# Creating encoder inp, decoder inp and decoder_out
encoder_inp = '<' + df['Informal text'].astype(str) + '>'
decoder_inp = '<' + df['Formal text'].astype(str)
```

```
decoder_out = df['Formal text'].astype(str) + '>'
# Creating the dataframe
preprocessed_data = pd.DataFrame()
preprocessed_data['encoder_inp'] = encoder_inp
preprocessed_data['decoder_inp'] = decoder_inp
preprocessed_data['decoder_out'] = decoder_out
preprocessed_data.head()
```

Out[5]:

	encoder_inp	decoder_inp	decoder_out
0	<U wan me to "chop" seat 4 u nt?>	<Do you want me to reserve seat for you or not?	Do you want me to reserve seat for you or not?>
1	<Yup. U reaching. We order some durian pastry ...	<Yeap. You reaching? We ordered some Durian pa...	Yeap. You reaching? We ordered some Durian pas...
2	<They become more ex oredi... Mine is like 25....	<They become more expensive already. Mine is l...	They become more expensive already. Mine is li...
3	<I'm thai. what do u do?>	<I'm Thai. What do you do?	I'm Thai. What do you do?>
4	<Hi! How did your week go? Haven heard from yo...	<Hi! How did your week go? Haven't heard from ...	Hi! How did your week go? Haven't heard from y...

2.4. Splitting the data into training and validation sets:

Before we split the data, we will look at the distribution of lengths of encoder inp, decoder inp and decoder_out to get the idea of input shape we will need to embed our data into.

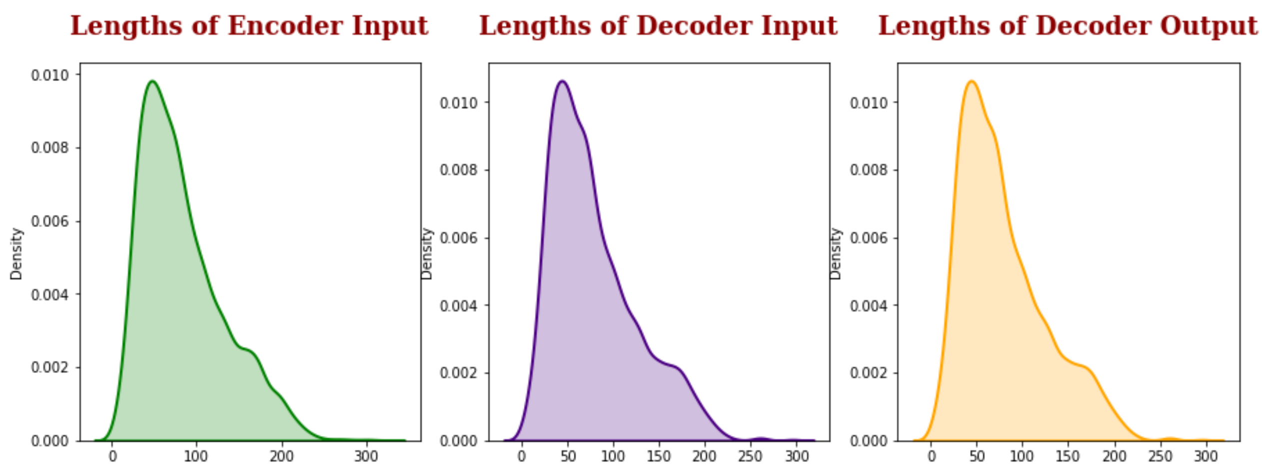
In [ ]:

```
# creating axes to draw plots
fig, ax = plt.subplots(1, 3)

# plotting the distributions
sns.distplot(preprocessed_data['encoder_inp'].apply(len).values, hist = False, kde = Tr
sns.distplot(preprocessed_data['decoder_inp'].apply(len).values, hist = False, kde = Tr
sns.distplot(preprocessed_data['decoder_out'].apply(len).values, hist = False, kde = Tr

# adding titles to the subplots
ax[0].set_title("Lengths of Encoder Input", fontdict = title_font, pad = 20.0)
ax[1].set_title("Lengths of Decoder Input", fontdict = title_font, pad = 20.0)
ax[2].set_title("Lengths of Decoder Output", fontdict = title_font, pad = 20.0)

# rescaling the figure
fig.set_figheight(5)
fig.set_figwidth(15)
```



As we can see, most of the sentences are of length around 50 and almost all the sentences have lengths less than 200. Hence, we can filter out the sentences which are of length more than 200.

In [6]:

```
# Filtering out sentences of length more than 200
preprocessed_data = preprocessed_data[preprocessed_data['encoder_inp'].apply(len) <= 20
preprocessed_data = preprocessed_data[preprocessed_data['decoder_inp'].apply(len) <= 20
preprocessed_data = preprocessed_data[preprocessed_data['decoder_out'].apply(len) <= 20
preprocessed_data.head()
```

Out[6]:

	encoder_inp	decoder_inp	decoder_out
0	<U wan me to "chop" seat 4 u nt? >	<Do you want me to reserve seat for you or not?	Do you want me to reserve seat for you or not?>
1	<Yup. U reaching. We order some durian pastry ...	<Yeap. You reaching? We ordered some Durian pa...	Yeap. You reaching? We ordered some Durian pas...
2	<They become more ex oredi... Mine is like 25....	<They become more expensive already. Mine is l...	They become more expensive already. Mine is li...
3	<I'm thai. what do u do?>	<I'm Thai. What do you do?	I'm Thai. What do you do?>
4	<Hi! How did your week go? Haven heard from yo...	<Hi! How did your week go? Haven't heard from ...	Hi! How did your week go? Haven't heard from y...

We can now split the data into train, validation and test sets. As we have less data, we will split with about 90:05:05 split to use more data to train the model.

In [ ]:

```
train, validation = train_test_split(preprocessed_data, test_size=0.025, random_state =
train, test = train_test_split(train, test_size=0.025, random_state = 859)
joblib.dump(train, 'train.pkl')
joblib.dump(validation, 'validation.pkl')
joblib.dump(test, 'test.pkl')
print(f"Shape of Training set: {train.shape}")
print(f"Shape of Validation set: {validation.shape}")
print(f"Shape of Test set: {test.shape}")
```

```
# for one sentence we will be adding <end> token so that the tokanizer learns the word
# with this we can use only one tokenizer for both encoder output and decoder output
train.iloc[0]['encoder_inp']= str(train.iloc[0]['encoder_inp']) + '>'
```

```
Shape of Training set: (7403, 3)
Shape of Validation set: (195, 3)
Shape of Test set: (190, 3)
```

2.5. Tokenizing data:

Tokenizing the data means, encoding the sentences with numbers. The numbers are assigned by an unique id from the vocabulary. So, the particular sentence will be encoded by unique ids of words occuring in that sentence. We will create the two tokenizers each for informal and formal data.

In [ ]:

```
# Tokenizing informal data with case preservation and excluding common punctuations lik
tknizer_informal = Tokenizer(filters = '"#$%&()*+-/=@[\\]^_`{|}~\t\n', lower = False, c
tknizer_informal.fit_on_texts(train['encoder_inp'].values)
joblib.dump(tknizer_informal, 'tknizer_informal.pkl')
# Tokenizing formal data with case preservation and excluding common punctuations like
tknizer_formal = Tokenizer(filters = '"#$%&()*+-/=@[\\]^_`{|}~\t\n', lower = False, cha
# Introducing '<end>' token on first sentence so that vocabulary learns it
train['decoder_inp'].iloc[0] = train['decoder_inp'].iloc[0] + '>'
tknizer_formal.fit_on_texts(train['decoder_inp'].values)
joblib.dump(tknizer_formal, 'tknizer_formal.pkl')
# Printing sizes of vocabularies
vocab_size_informal = len(tknizer_informal.word_index.keys())
print(f"Vocab size of Informal text: {vocab_size_informal}")
vocab_size_formal = len(tknizer_formal.word_index.keys())
print(f"Vocab size of Formal text: {vocab_size_formal}")
```

```
Vocab size of Informal text: 118
Vocab size of Formal text: 92
```

2.6. Padding data:

Padding refers to appending a common id (i.e. generally 0) to make all the sentences of same length. As we saw earlier, we can make the sentence lengths as 200.

In [ ]:

```
# Encoding the sentences by numerical ids in place of words
encoder_seq = tknizer_informal.texts_to_sequences(train['encoder_inp'].values)
decoder_inp_seq = tknizer_formal.texts_to_sequences(train['decoder_inp'].values)
decoder_out_seq = tknizer_formal.texts_to_sequences(train['decoder_out'].values)
# Padding the sentences to make all the sentences of same length
encoder_seq = pad_sequences(encoder_seq, maxlen = 200, dtype='int32', padding='post')
decoder_inp_seq = pad_sequences(decoder_inp_seq, maxlen = 200, dtype='int32', padding='
decoder_out_seq = pad_sequences(decoder_out_seq, maxlen = 200, dtype='int32', padding='
```

3. Designing the Data Pipeline:

We have to build a data pipeline to train the model as model expects tuples of length batch size of preprocessed data at runtime. We will load the source and target tokenizers and pad the data into sequences. Then, feed it according to the batch size.

3.1. Preprocessing the Data:

We will first convert sentences into sequences by tokenizing and padding.

In [ ]:

```
class Dataset:
    '''
    Generic class used to preprocess the data
    '''
    def __init__(self, data, tknizer_informal, tknizer_formal, max_len):
        '''
        This method intializes input sequences and the tokenizers
        '''
        self.encoder_inps = data['encoder_inp'].values
        self.decoder_inps = data['decoder_inp'].values
        self.decoder_outs = data['decoder_out'].values
        self.tknizer_informal = tknizer_informal
        self.tknizer_formal = tknizer_formal
        self.max_len = max_len

    def __getitem__(self, i):
        '''
        This method tokenizes the data and pads it with zeros to make all the sequences
        '''
        # Tokenizing the sequences by passing them in lists as required by tokenizer
        self.encoder_inp_seq = self.tknizer_informal.texts_to_sequences([self.encoder_i
        self.decoder_inp_seq = self.tknizer_formal.texts_to_sequences([self.decoder_inp
        self.decoder_out_seq = self.tknizer_formal.texts_to_sequences([self.decoder_out
        # Padding the sequences with zeros
        self.encoder_inp_seq = pad_sequences(self.encoder_inp_seq, maxlen = self.max_le
        self.decoder_inp_seq = pad_sequences(self.decoder_inp_seq, maxlen = self.max_le
        self.decoder_out_seq = pad_sequences(self.decoder_out_seq, maxlen = self.max_le
        return self.encoder_inp_seq, self.decoder_inp_seq, self.decoder_out_seq

    def __len__(self):
        '''
        This method is required by model.fit method at runtime to keep logs
        '''
        return len(self.encoder_inps)
```

3.2. Creating Dataloader:

we will now design a dataloader which shuffles the preprocessed dataset and returns the tuple of form ([[encoder_inp], [decoder_inp]], decoder_out) at runtime

In [ ]:

```
class Dataloader(tf.keras.utils.Sequence):
    '''
    Keras Dataloader instance to feed the model with preprocessed data at runtime
    '''
    def __init__(self, dataset, batch_size = 1):
        '''
        this method initializes preprocessed dataset and batch size
        '''
        self.dataset = dataset
        self.batch_size = batch_size
        self.indexes = np.arange(len(self.dataset.encoder_inps))

    def __getitem__(self, i):
        '''
        This method is used to pack the input data in tuples of form ([[encoder_inp], [
        '''
        # Tracking indices of start and stop
        start = i * self.batch_size
        stop = (i + 1) * self.batch_size
        data = []
        for j in range(start, stop):
            data.append(self.dataset[j])
        # Creating data in tuples of form ([[encoder_inp], [decoder_inp]], decoder_out)
        batch = [np.squeeze(np.stack(samples, axis = 1), axis = 0) for samples in zip(*
        return tuple([[batch[0],batch[1]],batch[2]])

    def __len__(self):
        '''
        This method is required by model.fit method at runtime to keep logs
        '''
        return len(self.indexes) // self.batch_size

    def on_epoch_end(self):
        '''
        This method is a callback to shuffle the indices of data on each epoch
        '''
        self.indexes = np.random.permutation(self.indexes)
```

In [ ]:

```
# Defining parameters
BATCH_SIZE = 128
MAX_LEN = 200
# Preprocessing data
train_dataset = Dataset(train, tknizer_informal, tknizer_formal, MAX_LEN)
validation_dataset  = Dataset(validation, tknizer_formal, tknizer_formal, MAX_LEN)
# Creating Dataloader
train_dataloader = Dataloader(train_dataset, batch_size = BATCH_SIZE)
validation_dataloader = Dataloader(validation_dataset, batch_size = BATCH_SIZE)
# Checking the dimensions
print(train_dataloader[0][0][0].shape, train_dataloader[0][0][1].shape, train_dataloade
```

```
(128, 200) (128, 200) (128, 200)
```

4. Designing the Attention based Encoder Decoder Model:

4.1. Designing Encoder:

The encoder will take sequential word embeddings of the source sentences as input at each time step, and encode its information in encoded vector using current state and LSTM hidden state. Hence, at the output of encoder, we get an encoded vector of source sentence which can be thought of as latent information vector.

In [ ]:

```
class Encoder(tf.keras.Model):
    '''
    Encoder model takes a input sequence and returns Encoder outputs as encoder_final_h
    '''
    def __init__(self, inp_vocab_size, embedding_dim, lstm_size, input_length):
        '''
          This method intializes the Encoder model.
        '''
        super().__init__()
        # Initializing the parameters
        self.inp_vocab_size = inp_vocab_size
        self.embedding_dim = embedding_dim
        self.lstm_size = lstm_size
        self.input_length = input_length
        # Initializing Embedding layer
        self.embedding = Embedding(input_dim = self.inp_vocab_size, output_dim = self.e
                                    input_length = self.input_length, mask_zero = True,
        #Intializing Encoder LSTM layer
        self.lstm1 = LSTM(self.lstm_size, return_state = True, return_sequences = True,
        self.lstm2 = LSTM(self.lstm_size, return_state = True, return_sequences = True,

    def call(self, input_sequence, states):
        '''
          This method takes a sequence input and the initial states of the Encoder as i
          Sequence input is passed to the Embedding layer and initial states are passed
          It returns Encoder outputs as last time step's hidden and current states.
        '''
        # Passing input sequence to embedding layer
        input_embedded = self.embedding(input_sequence)
        # Passing embedidng layer output to lstm layer
        self.enc_output, self.last_hidden_state, self.last_current_state = self.lstm1(i
        self.enc_output, self.last_hidden_state, self.last_current_state = self.lstm2(s
        # Returning the outputs
        return self.enc_output, self.last_hidden_state, self.last_current_state

    def initialize_states(self,batch_size):
      '''
        Given a batch size this method will return intial hidden state and intial curre
        If batch size is 32, Hidden state is zeros of size [32,lstm_units], current sta
      '''
      self.first_hidden_state, self.first_current_state = tf.zeros([batch_size, self.ls
      # Returning the initializations
      return self.first_hidden_state, self.first_current_state
```

4.2. Designing Attention Model:

Attention model takes two inputs in the form of decoder hidden state of previous timestep and encoder output and calculates attention weights.

In [ ]:

```
class Attention(tf.keras.Model):
    '''
        Attention model takes two inputs in the form of decoder_hidden_state, encoder_o
        returns context vector and attention weights(softmax - scores).
    '''
    def __init__(self, lstm_size, scoring_function):
        super(Attention, self).__init__()
        # Initializing the parameters
        self.lstm_size = lstm_size
        self.scoring_function = scoring_function
        # Initializing weights for 'dot' scoring function
        if self.scoring_function=='dot':
            self.V = tf.keras.layers.Dense(1)
        # Initializing weights for 'general' scoring function
        if scoring_function == 'general':
            self.W = tf.keras.layers.Dense(lstm_size)
        # Initializing weights for 'concat' scoring function
        if scoring_function == 'concat':
            self.W1 = tf.keras.layers.Dense(lstm_size)
            self.W2 = tf.keras.layers.Dense(lstm_size)
            self.V = tf.keras.layers.Dense(1)

    def call(self,decoder_hidden_state,encoder_output):
        '''
            Attention model takes two inputs in the form of decoder_hidden_state and al
            Based on the scoring function we will find the score or similarity between
            Multiply the score function with your encoder_outputs to get the context ve
            Returns context vector and attention weights(softmax - scores)
        '''
        if self.scoring_function == 'dot':
            # Implement Dot score function here
            query_with_time_axis = tf.expand_dims(decoder_hidden_state, 1)
            score = self.V(tf.linalg.matmul(encoder_output, query_with_time_axis, trans

        elif self.scoring_function == 'general':
            # Implement General score function here
            decoder_hidden_state = tf.expand_dims(decoder_hidden_state, axis = 2)
            output = self.W(encoder_output)
            score = tf.keras.layers.Dot(axes=(2, 1))([output, decoder_hidden_state])
        if self.scoring_function == 'concat':
            # Implement General score function here
            decoder_hidden_state = tf.expand_dims(decoder_hidden_state, 1)
            score = self.V(tf.nn.tanh(self.W1(decoder_hidden_state) + self.W2(encoder_o
        # Calculating context vector and attention weights
        attention_weights = tf.nn.softmax(score, axis=1)
        context_vector = attention_weights * encoder_output
        context_vector = tf.reduce_sum(context_vector, axis=1)
        return context_vector, attention_weights
```

4.3. Designing Timestep Decoder:

For each time step, Timestep decoder will implement concatenation operation on output of previous timestep of decoder and attention weights computed by attention model.

In [ ]:

```
class Timestep_Decoder(tf.keras.Model):
    '''
       Timestep Decoder model takes one input token at a time and returns final hidden
    '''
    def __init__(self, out_vocab_size, embedding_dim, input_length, lstm_size, scoring_
        # Initialize the parameters
        super().__init__()
        self.out_vocab_size = out_vocab_size
        self.embedding_dim = embedding_dim
        self.input_length = input_length
        self.lstm_size = lstm_size
        self.scoring_function = scoring_function
        self.attention = Attention(self.lstm_size, self.scoring_function)
        self.embedding_matrix = embedding_matrix
        # Initializing Embedding layer based on the availability of Embedding matrix
        if self.embedding_matrix is None:
            self.embedding = Embedding(input_dim = self.out_vocab_size, output_dim = se
                                       input_length = self.input_length, mask_zero = Tr
        else:
            self.embedding = Embedding(input_dim = self.out_vocab_size, output_dim = se
                                       input_length = self.input_length, mask_zero = Tr
                                       embeddings_initializer = tf.keras.initializers.C
        #Intialize Decoder LSTM layer
        self.lstm1 = LSTM(self.lstm_size, return_sequences=True, return_state=True, nam
        self.lstm2 = LSTM(self.lstm_size, return_sequences=True, return_state=True, nam
        #Intialize Dense layer(tar_vocab_size) without activation='softmax'
        self.dense = Dense(out_vocab_size)


    def call(self, input_token, encoder_output, encoder_hidden, encoder_current):
        '''
            Timestep decoder model generates final lstm unit hidden state depending on
        '''
        # Passing the decoder_input to the embedding layer
        embedded_token = self.embedding(input_token)
        # Passing encoder hidden and encoder output to attention model to get context v
        context_vector, attention_weights = self.attention(encoder_hidden, encoder_outp
        # Reshaping context vector for concatenation
        query_with_time_axis = tf.expand_dims(context_vector, 1)
        # Concatenating context vector and embedded token
        out_concat = tf.concat([query_with_time_axis, embedded_token], axis = -1)
        # Getting final lstm hidden state
        dec_output, encoder_hidden, encoder_current = self.lstm1(out_concat, [encoder_h
        dec_output, encoder_hidden, encoder_current = self.lstm2(dec_output, [encoder_h
        # Weighing decoder output by output vocabulary size
        out = self.dense(tf.reshape(dec_output, (-1, dec_output.shape[2])))
        # Returning the output
        return out, encoder_hidden, encoder_current
```

4.4. Designing Decoder:

Decoder model simply calls timestep decoder at each timestep and generates the final output tokens.

In [ ]:

```
class Decoder(tf.keras.Model):
    '''
        Decoder model generates the final output tokens by passing it all the encoder s
    '''
    def __init__(self, out_vocab_size, embedding_dim, input_length, lstm_size, scoring_
        super().__init__()
        # Intializing the parameters
        self.out_vocab_size = out_vocab_size
        self.embedding_dim = embedding_dim
        self.input_length = input_length
        self.lstm_size = lstm_size
        self.scoring_function = scoring_function
        self.embedding_matrix = embedding_matrix
        # Initializing Timestep decoder instance
        self.timestepdecoder = Timestep_Decoder(self.out_vocab_size, self.embedding_dim
                                                self.lstm_size, self.scoring_function,

    def call(self, decoder_input, encoder_output, encoder_hidden, encoder_current):
        '''
            According to the length of the decoder input sequence, returns a tensor of
        '''
        # Initializing an empty Tensor array, that will store the outputs at each and e
        all_outputs = tf.TensorArray(tf.float32, size = tf.shape(decoder_input)[1], nam
        # Iterating till the length of the decoder input
        for timestep in range(tf.shape(decoder_input)[1]):
            # Calling the Timestep Decoder for each token in decoder input
            output, encoder_hidden, encoder_current = self.timestepdecoder(decoder_inpu
            # Storing the output in tensorarray
            all_outputs = all_outputs.write(timestep, output)
        # Reshaping the tensor array
        all_outputs = tf.transpose(all_outputs.stack(), [1,0,2])
        # Returning the tensor array
        return all_outputs
```

4.5. Designing Final Model Architechture:

Attention based Encoder Decoder model gets the tuple of input sequences as input and implements the Encoder, Attention, Timestep Decoder and Decoder models using subclassing API.

In [ ]:

```
class Attention_Based_Encoder_Decoder(tf.keras.Model):
    '''
    The Attention_Based_Encoder_Decoder Model initializes both Encoder and Decoder Mode
    '''
    def __init__(self, input_length, inp_vocab_size, out_vocab_size, lstm_size, scoring
        '''
           This method intializes the both the Encoder and Decoder models
```

```
        '''
        super().__init__()
        # Initializing the parameters
        self.input_length = input_length
        self.inp_vocab_size = inp_vocab_size + 1
        self.out_vocab_size = out_vocab_size + 1
        self.lstm_size = lstm_size
        self.scoring_function = scoring_function
        self.batch_size = batch_size
        self.embedding_dim = embedding_dim
        self.embedding_matrix = embedding_matrix
        #Creating Encoder model object
        self.encoder = Encoder(inp_vocab_size = self.inp_vocab_size, embedding_dim = se
        #Creating Decoder model object
        self.decoder = Decoder(out_vocab_size = self.out_vocab_size, embedding_dim = se
                               scoring_function = self.scoring_function, input_length =

    def call(self, data):
        '''
        This method takes data from data pipeline in tuples of length 2, where first is
        encoder_inp is fed to Encoder model object alongwith initial states whereas dec
        Encoder last hidden and current states.
        The Model then returns normalized output probabilities of tokens in target voca
        '''
        # Unpacking data
        enc_inp, dec_inp = data[0], data[1]
        # Initializing Encoder initial states
        initial_state = self.encoder.initialize_states(self.batch_size)
        # Calling Encoder model object
        encoder_output, encoder_hidden, encoder_current = self.encoder(enc_inp, initial
        # Calling Decoder model object
        final_output = self.decoder(dec_inp, encoder_output, encoder_hidden, encoder_cu
        return final_output
```

5. Designing the Model Pipeline:

5.1. Creating Custom Loss Function:

We will now create a custom loss Function that will mask the padded zeros while for more reliable loss calculation.

In [ ]:

```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits = True, reducti

@tf.function
def loss_function(real, pred):
    # Custom loss function that will not consider the loss for padded zeros.
    # Refer https://www.tensorflow.org/tutorials/text/nmt_with_attention
    # optimizer = tf.keras.optimizers.Adam()
    mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss_ = loss_object(real, pred)
    mask = tf.cast(mask, dtype=loss_.dtype)
    loss_ *= mask
    return tf.reduce_mean(loss_)
```

5.2. Creating Tensorboard Callback:

To keep track of loss while training the model, we will create a Tensorboard callback by providing log directory.

In [ ]:

```
def create_tensorboard_cb(model):
    '''
    Takes path string as input and returns tensorboard callback initialized in that pat
    '''
    import time
    root_logdir = os.path.join(os.curdir, model)
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    logdir = os.path.join(root_logdir, run_id)
    return tf.keras.callbacks.TensorBoard(logdir, histogram_freq = 1)
```

5.3. Creating Predict Function:

The `predict` function will take as informal input sentence and model instance with which to predict as input and return the output as prediction.

In [ ]:

```
def predict(input_sentence, model):
    '''
    Takes input sentence and model instance as inputs and predicts the output.
    The prediction is done by using following steps:
    Step A. Given input sentence, preprocess the punctuations, convert the sentence int
    Step B. Pass the input_sequence to encoder. we get encoder_outputs, last time step
    Step C. Initialize index of '<' as input to decoder. and encoder final states as in
    Step D. Till we reach max_length of decoder or till the model predicted word '>':
            pass the inputs to timestep decoder at each timestep, update the hidden sta
    Step E. Return the predicted sentence.
    '''
    # Tokenizing and Padding the sentence
    inputs = [tknizer_informal.word_index.get(i, 0) for i in input_sentence]
    inputs = tf.keras.preprocessing.sequence.pad_sequences([inputs], maxlen = MAX_LEN,
    inputs = tf.convert_to_tensor(inputs)
    # Initializing result string and hidden states
    result = ''
    hidden = tf.zeros([1, UNITS]), tf.zeros([1, UNITS])
    # Getting Encoder outputs
    enc_out, state_h, state_c = model.encoder(inputs, hidden)
    dec_hidden = [state_h, state_c]
    dec_input = tf.expand_dims([tknizer_formal.word_index['<']], 0)
    # Running loop until max length or the prediction is '>' token
    for t in range(MAX_LEN):
        # Getting Decoder outputs fot timestep t
        output, state_h, state_c = model.decoder.timestepdecoder(dec_input, enc_out, st
        # Getting token index having highest probability
        predicted_id = tf.argmax(output[0]).numpy()
        # Getting output token
        if tknizer_formal.index_word.get(predicted_id, '') == '>':
            break
        else:
            result += tknizer_formal.index_word.get(predicted_id, '')
```

```
            dec_input = tf.expand_dims([predicted_id], 0)
    # Postprocessing the result string to remove spaces between punctuations
    return result
```

6. Training the Model using Dot Scoring Function:

6.1. Compiling and Fitting the model:

We can now train the model by using model fit method.

In [ ]:

```
tf.random.set_seed(859)
# Defining model parameters
UNITS = 200
EPOCHS = 50
TRAIN_STEPS = train.shape[0]//BATCH_SIZE
VALID_STEPS = validation.shape[0]//BATCH_SIZE
# Defining model instance with 'dot' scoring function
model_dot  = Attention_Based_Encoder_Decoder(input_length = MAX_LEN, inp_vocab_size = v
                                             out_vocab_size = vocab_size_formal, lstm_s
                                             scoring_function = 'dot', batch_size = BAT
                                             embedding_dim = vocab_size_formal, embeddi
# Compiling the model using 'adam' optimizer and custom loss function
optimizer = tf.keras.optimizers.Adam(0.01)
model_dot.compile(optimizer = optimizer, loss = loss_function)
# Creating callbacks to control model training
learning_rate_cb = tf.keras.callbacks.ReduceLROnPlateau(monitor = 'val_loss', factor =
tensorboard_cb = create_tensorboard_cb("Model_Dot_logs")
stopper_cb = tf.keras.callbacks.EarlyStopping(monitor = 'val_loss', patience = 2, verbo
checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("Model_Dot.h5",
                                                   save_best_only = True, save_weights
# Fitting the model on training data
model_dot.fit(train_dataloader, steps_per_epoch = TRAIN_STEPS, epochs = EPOCHS,
              callbacks = [learning_rate_cb, tensorboard_cb, stopper_cb, checkpoint_cb]
              validation_data = validation_dataloader, validation_steps = VALID_STEPS)
model_dot.summary()
```

```
Epoch 1/50
57/57 [==============================] - 122s 2s/step - loss: 1.1210 - val_loss: 1.0406
Epoch 2/50
57/57 [==============================] - 102s 2s/step - loss: 0.8188 - val_loss: 0.8296
Epoch 3/50
57/57 [==============================] - 101s 2s/step - loss: 0.6637 - val_loss: 0.6886
Epoch 4/50
57/57 [==============================] - 102s 2s/step - loss: 0.5683 - val_loss: 0.6106
Epoch 5/50
57/57 [==============================] - 102s 2s/step - loss: 0.5132 - val_loss: 0.5731
Epoch 6/50
57/57 [==============================] - 102s 2s/step - loss: 0.4758 - val_loss: 0.5366
Epoch 7/50
57/57 [==============================] - 105s 2s/step - loss: 0.4455 - val_loss: 0.5196
Epoch 8/50
57/57 [==============================] - 102s 2s/step - loss: 0.4224 - val_loss: 0.5007
Epoch 9/50
57/57 [==============================] - 101s 2s/step - loss: 0.4036 - val_loss: 0.4801
Epoch 10/50
57/57 [==============================] - 98s 2s/step - loss: 0.3874 - val_loss: 0.4739
Epoch 11/50
```

```
57/57 [==============================] - 99s 2s/step - loss: 0.3733 - val_loss: 0.4621
Epoch 12/50
57/57 [==============================] - 99s 2s/step - loss: 0.3595 - val_loss: 0.4504
Epoch 13/50
57/57 [==============================] - 99s 2s/step - loss: 0.3481 - val_loss: 0.4464
Epoch 14/50
57/57 [==============================] - 103s 2s/step - loss: 0.3384 - val_loss: 0.4411
Epoch 15/50
57/57 [==============================] - 100s 2s/step - loss: 0.3313 - val_loss: 0.4270
Epoch 16/50
57/57 [==============================] - 99s 2s/step - loss: 0.3221 - val_loss: 0.4309

Epoch 00016: ReduceLROnPlateau reducing learning rate to 0.004999999888241291.
Epoch 17/50
57/57 [==============================] - 99s 2s/step - loss: 0.2907 - val_loss: 0.3985
Epoch 18/50
57/57 [==============================] - 99s 2s/step - loss: 0.2697 - val_loss: 0.3969
Epoch 19/50
57/57 [==============================] - 99s 2s/step - loss: 0.2599 - val_loss: 0.3947
Epoch 20/50
57/57 [==============================] - 98s 2s/step - loss: 0.2541 - val_loss: 0.3950

Epoch 00020: ReduceLROnPlateau reducing learning rate to 0.0024999999441206455.
Epoch 21/50
57/57 [==============================] - 98s 2s/step - loss: 0.2340 - val_loss: 0.3792
Epoch 22/50
57/57 [==============================] - 98s 2s/step - loss: 0.2201 - val_loss: 0.3791

Epoch 00022: ReduceLROnPlateau reducing learning rate to 0.0012499999720603228.
Epoch 23/50
57/57 [==============================] - 98s 2s/step - loss: 0.2079 - val_loss: 0.3774
Epoch 24/50
57/57 [==============================] - 98s 2s/step - loss: 0.2010 - val_loss: 0.3817

Epoch 00024: ReduceLROnPlateau reducing learning rate to 0.0006249999860301614.
Epoch 25/50
57/57 [==============================] - 97s 2s/step - loss: 0.1944 - val_loss: 0.3799

Epoch 00025: ReduceLROnPlateau reducing learning rate to 0.0003124999930150807.
Restoring model weights from the end of the best epoch.
Epoch 00025: early stopping
Model: "attention__based__encoder__decoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoder (Encoder)            multiple                  566148
_________________________________________________________________
decoder (Decoder)            multiple                  742451
=================================================================
Total params: 1,308,599
Trainable params: 1,308,599
Non-trainable params: 0
_________________________________________________________________
```

The model achieves the loss of 0.3774 on validation set which is better than that of simple encoder decoder model.

6.2. Calculating the BLEU Score:

We can now calculate the BLEU score to quantify the model performance.

In [ ]:

```
# Removing '<' and '>' tokens and postprocessing punctuations to make plain texts
def rem(s):
    if s.startswith('<'):
        s = s[1:]
    if s.endswith('>'):
        s = s[:-1]
    return s
test['informals'] = test['encoder_inp'].apply(rem)
test['formals'] = test['decoder_inp'].apply(rem)
def predictor(s):
    # Modifing predictor using dot scoring function
    result = predict(s, model_dot)
    return result
test['predictions'] = test['informals'].apply(predictor)
# Process inputs for Bleu score
def convert_formals(s):
    return [s.split()]
def convert_predictions(s):
    return s.split()
test['formals'] = test['formals'].apply(convert_formals)
test['predictions'] = test['predictions'].apply(convert_predictions)
bleu_scores = [sentence_bleu(test['formals'].iloc[i], test['predictions'].iloc[i]) for
print(f"Mean Bleu score of predictions: {np.mean(bleu_scores)}")
```

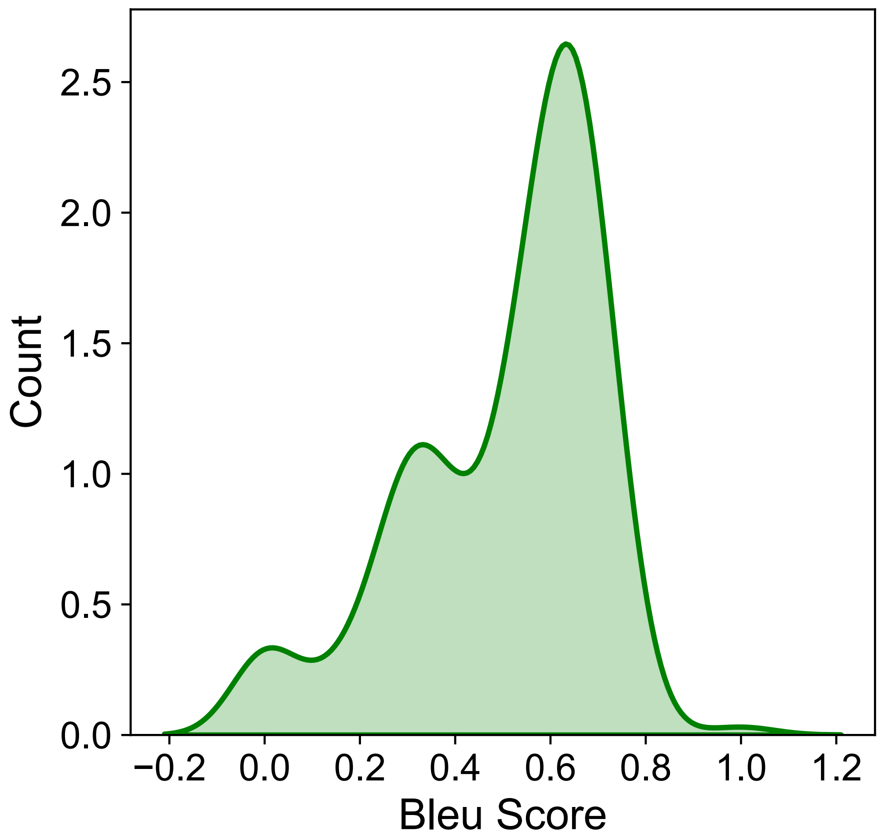
```
Mean Bleu score of predictions: 0.5053083753162261
```

The model achieves the BLEU score of 0.505 on test set which is significantly better than that of baseline encoder decoder model. Let us check the distribution of the bleu scores.

In [8]:

```
plt.figure(figsize = (5, 5))
ax = sns.distplot(bleu_scores, hist = False, kde = True, kde_kws = {'shade': True, 'lin
plt.title("Distribution of Bleu Scores by Dot Model", fontdict = title_font, pad = 20.0
plt.xlabel("Bleu Score", fontdict = label_font)
plt.ylabel("Count", fontdict = label_font)
for label in (ax.get_xticklabels() + ax.get_yticklabels()):
    label.set_fontname('Arial')
    label.set_fontsize(14)
plt.show()
```

Distribution of Bleu Scores by Dot Model



The distribution shows that the model with dot scoring function achieves the bleu score of around 0.7 for majority of the sentences. Let us generate a random prediction using this model.

In [ ]:

```
print("Informal Sentence: wat r ya talkin abt")
print(f"Formal Prediction: {predict('wat r ya talkin abt', model_dot)}")
```

```
Informal Sentence: wat r ya talkin abt
Formal Prediction: We are on the more about the night .
```

The model corrected the informal words 'r' and 'abt' to 'are' and 'about' respectively. However, it struggled to correct other words. It may be due to the fewer occurences of the words in the target vocabulary. This issue can be overcome by training the model on large dataset.

7. Training the Model using General Scoring Function:

7.1. Compiling and Fitting the model:

We can now train the model by using model fit method.

In [ ]:

```
tf.random.set_seed(859)
# Defining model instance with 'general' scoring function
model_general  = Attention_Based_Encoder_Decoder(input_length = MAX_LEN, inp_vocab_size
                                                 out_vocab_size = vocab_size_formal, lstm_s
                                                 scoring_function = 'general', batch_size =
                                                 embedding_dim = vocab_size_formal, embeddi
# Compiling the model using 'adam' optimizer and custom loss function
optimizer = tf.keras.optimizers.Adam(0.01)
model_general.compile(optimizer = optimizer, loss = loss_function)
# Creating callbacks to control model training
learning_rate_cb = tf.keras.callbacks.ReduceLROnPlateau(monitor = 'val_loss', factor =
tensorboard_cb = create_tensorboard_cb("Model_General_logs")
stopper_cb = tf.keras.callbacks.EarlyStopping(monitor = 'val_loss', patience = 3, verbo
checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("Model_General.h5",
                                                   save_best_only = True, save_weights
# Fitting the model on training data
model_general.fit(train_dataloader, steps_per_epoch = TRAIN_STEPS, epochs = EPOCHS,
              callbacks = [learning_rate_cb, tensorboard_cb, stopper_cb, checkpoint_cb]
              validation_data = validation_dataloader, validation_steps = VALID_STEPS)
model_general.summary()
```

```
Epoch 1/50
57/57 [==============================] - 129s 2s/step - loss: 1.1200 - val_loss: 1.0382
Epoch 2/50
57/57 [==============================] - 111s 2s/step - loss: 0.8096 - val_loss: 0.7968
Epoch 3/50
57/57 [==============================] - 110s 2s/step - loss: 0.6537 - val_loss: 0.6525
Epoch 4/50
57/57 [==============================] - 110s 2s/step - loss: 0.5507 - val_loss: 0.6274
Epoch 5/50
57/57 [==============================] - 110s 2s/step - loss: 0.4919 - val_loss: 0.5500
Epoch 6/50
57/57 [==============================] - 110s 2s/step - loss: 0.4539 - val_loss: 0.5078
Epoch 7/50
57/57 [==============================] - 109s 2s/step - loss: 0.4239 - val_loss: 0.4900
Epoch 8/50
57/57 [==============================] - 109s 2s/step - loss: 0.4014 - val_loss: 0.4712
Epoch 9/50
57/57 [==============================] - 109s 2s/step - loss: 0.3807 - val_loss: 0.4567
Epoch 10/50
57/57 [==============================] - 109s 2s/step - loss: 0.3664 - val_loss: 0.4445
Epoch 11/50
57/57 [==============================] - 109s 2s/step - loss: 0.3547 - val_loss: 0.4362
Epoch 12/50
57/57 [==============================] - 111s 2s/step - loss: 0.3449 - val_loss: 0.4233
Epoch 13/50
57/57 [==============================] - 110s 2s/step - loss: 0.3371 - val_loss: 0.4203
Epoch 14/50
57/57 [==============================] - 110s 2s/step - loss: 0.3281 - val_loss: 0.4101
Epoch 15/50
57/57 [==============================] - 110s 2s/step - loss: 0.3215 - val_loss: 0.4102
Epoch 16/50
57/57 [==============================] - 109s 2s/step - loss: 0.3175 - val_loss: 0.4020
Epoch 17/50
57/57 [==============================] - 109s 2s/step - loss: 0.3095 - val_loss: 0.4126
Epoch 18/50
57/57 [==============================] - 109s 2s/step - loss: 0.3135 - val_loss: 0.4118

Epoch 00018: ReduceLROnPlateau reducing learning rate to 0.004999999888241291.
Epoch 19/50
57/57 [==============================] - 112s 2s/step - loss: 0.2767 - val_loss: 0.3613
Epoch 20/50
```

```
57/57 [==============================] - 112s 2s/step - loss: 0.2516 - val_loss: 0.3505
Epoch 21/50
57/57 [==============================] - 111s 2s/step - loss: 0.2419 - val_loss: 0.3466
Epoch 22/50
57/57 [==============================] - 110s 2s/step - loss: 0.2364 - val_loss: 0.3398
Epoch 23/50
57/57 [==============================] - 114s 2s/step - loss: 0.2353 - val_loss: 0.3473
Epoch 24/50
57/57 [==============================] - 112s 2s/step - loss: 0.2307 - val_loss: 0.3449

Epoch 00024: ReduceLROnPlateau reducing learning rate to 0.0024999999441206455.
Epoch 25/50
57/57 [==============================] - 111s 2s/step - loss: 0.2088 - val_loss: 0.3082
Epoch 26/50
57/57 [==============================] - 109s 2s/step - loss: 0.1913 - val_loss: 0.2959
Epoch 27/50
57/57 [==============================] - 109s 2s/step - loss: 0.1842 - val_loss: 0.2935
Epoch 28/50
57/57 [==============================] - 109s 2s/step - loss: 0.1803 - val_loss: 0.2904
Epoch 29/50
57/57 [==============================] - 109s 2s/step - loss: 0.1780 - val_loss: 0.2919
Epoch 30/50
57/57 [==============================] - 111s 2s/step - loss: 0.1759 - val_loss: 0.2853
Epoch 31/50
57/57 [==============================] - 110s 2s/step - loss: 0.1726 - val_loss: 0.2886
Epoch 32/50
57/57 [==============================] - 109s 2s/step - loss: 0.1703 - val_loss: 0.2829
Epoch 33/50
57/57 [==============================] - 109s 2s/step - loss: 0.1667 - val_loss: 0.2790
Epoch 34/50
57/57 [==============================] - 109s 2s/step - loss: 0.1623 - val_loss: 0.2805
Epoch 35/50
57/57 [==============================] - 110s 2s/step - loss: 0.1639 - val_loss: 0.2845

Epoch 00035: ReduceLROnPlateau reducing learning rate to 0.0012499999720603228.
Epoch 36/50
57/57 [==============================] - 109s 2s/step - loss: 0.1506 - val_loss: 0.2563
Epoch 37/50
57/57 [==============================] - 108s 2s/step - loss: 0.1379 - val_loss: 0.2512
Epoch 38/50
57/57 [==============================] - 108s 2s/step - loss: 0.1325 - val_loss: 0.2493
Epoch 39/50
57/57 [==============================] - 109s 2s/step - loss: 0.1307 - val_loss: 0.2484
Epoch 40/50
57/57 [==============================] - 109s 2s/step - loss: 0.1300 - val_loss: 0.2472
Epoch 41/50
57/57 [==============================] - 108s 2s/step - loss: 0.1270 - val_loss: 0.2427
Epoch 42/50
57/57 [==============================] - 108s 2s/step - loss: 0.1246 - val_loss: 0.2459
Epoch 43/50
57/57 [==============================] - 109s 2s/step - loss: 0.1259 - val_loss: 0.2400
Epoch 44/50
57/57 [==============================] - 109s 2s/step - loss: 0.1243 - val_loss: 0.2400
Epoch 45/50
57/57 [==============================] - 110s 2s/step - loss: 0.1217 - val_loss: 0.2392
Epoch 46/50
57/57 [==============================] - 109s 2s/step - loss: 0.1197 - val_loss: 0.2396
Epoch 47/50
57/57 [==============================] - 110s 2s/step - loss: 0.1166 - val_loss: 0.2358
Epoch 48/50
57/57 [==============================] - 110s 2s/step - loss: 0.1154 - val_loss: 0.2348
Epoch 49/50
57/57 [==============================] - 108s 2s/step - loss: 0.1152 - val_loss: 0.2373
Epoch 50/50
57/57 [==============================] - 108s 2s/step - loss: 0.1141 - val_loss: 0.2378
```

```
Epoch 00050: ReduceLROnPlateau reducing learning rate to 0.0006249999860301614.
Model: "attention__based__encoder__decoder_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoder_4 (Encoder)          multiple                  566148
_________________________________________________________________
decoder_4 (Decoder)          multiple                  782649
=================================================================
Total params: 1,348,797
Trainable params: 1,348,797
Non-trainable params: 0
_________________________________________________________________
```

In [89]:

```
# Fitting the model on training data
model_general.fit(train_dataloader, steps_per_epoch = TRAIN_STEPS, epochs = EPOCHS,
              callbacks = [learning_rate_cb, tensorboard_cb, stopper_cb, checkpoint_cb]
              validation_data = validation_dataloader, validation_steps = VALID_STEPS)
model_general.summary()
```

```
Epoch 1/50
57/57 [==============================] - 121s 2s/step - loss: 0.1078 - val_loss: 0.2256
Epoch 2/50
57/57 [==============================] - 114s 2s/step - loss: 0.1016 - val_loss: 0.2244
Epoch 3/50
57/57 [==============================] - 113s 2s/step - loss: 0.0984 - val_loss: 0.2227
Epoch 4/50
57/57 [==============================] - 114s 2s/step - loss: 0.0968 - val_loss: 0.2236
Epoch 5/50
57/57 [==============================] - 114s 2s/step - loss: 0.0957 - val_loss: 0.2241

Epoch 00005: ReduceLROnPlateau reducing learning rate to 0.0003124999930150807.
Epoch 6/50
57/57 [==============================] - 113s 2s/step - loss: 0.0936 - val_loss: 0.2196
Epoch 7/50
57/57 [==============================] - 113s 2s/step - loss: 0.0917 - val_loss: 0.2214
Epoch 8/50
57/57 [==============================] - 112s 2s/step - loss: 0.0906 - val_loss: 0.2213

Epoch 00008: ReduceLROnPlateau reducing learning rate to 0.00015624999650754035.
Epoch 9/50
57/57 [==============================] - 112s 2s/step - loss: 0.0893 - val_loss: 0.2209
Restoring model weights from the end of the best epoch.
Epoch 00009: early stopping
Model: "attention__based__encoder__decoder_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoder_4 (Encoder)          multiple                  566148
_________________________________________________________________
decoder_4 (Decoder)          multiple                  782649
=================================================================
Total params: 1,348,797
Trainable params: 1,348,797
Non-trainable params: 0
_________________________________________________________________
```

The model achieves the loss of 0.2196 on validation set which is better than the model using dot scoring function.

7.2. Calculating the BLEU Score:

We can now calculate the BLEU score to quantify the model performance.

In [90]:

```
# Removing '<' and '>' tokens and postprocessing punctuations to make plain texts
def rem(s):
    if s.startswith('<'):
        s = s[1:]
    if s.endswith('>'):
        s = s[:-1]
    return s
test['informals'] = test['encoder_inp'].apply(rem)
test['formals'] = test['decoder_inp'].apply(rem)
def predictor(s):
    # Modifing predictor using general scoring function
    result = predict(s, model_general)
    return result
test['predictions'] = test['informals'].apply(predictor)
# Process inputs for Bleu score
def convert_formals(s):
    return [s.split()]
def convert_predictions(s):
    return s.split()
test['formals'] = test['formals'].apply(convert_formals)
test['predictions'] = test['predictions'].apply(convert_predictions)
bleu_scores = [sentence_bleu(test['formals'].iloc[i], test['predictions'].iloc[i]) for
print(f"Mean Bleu score of predictions: {np.mean(bleu_scores)}")
```

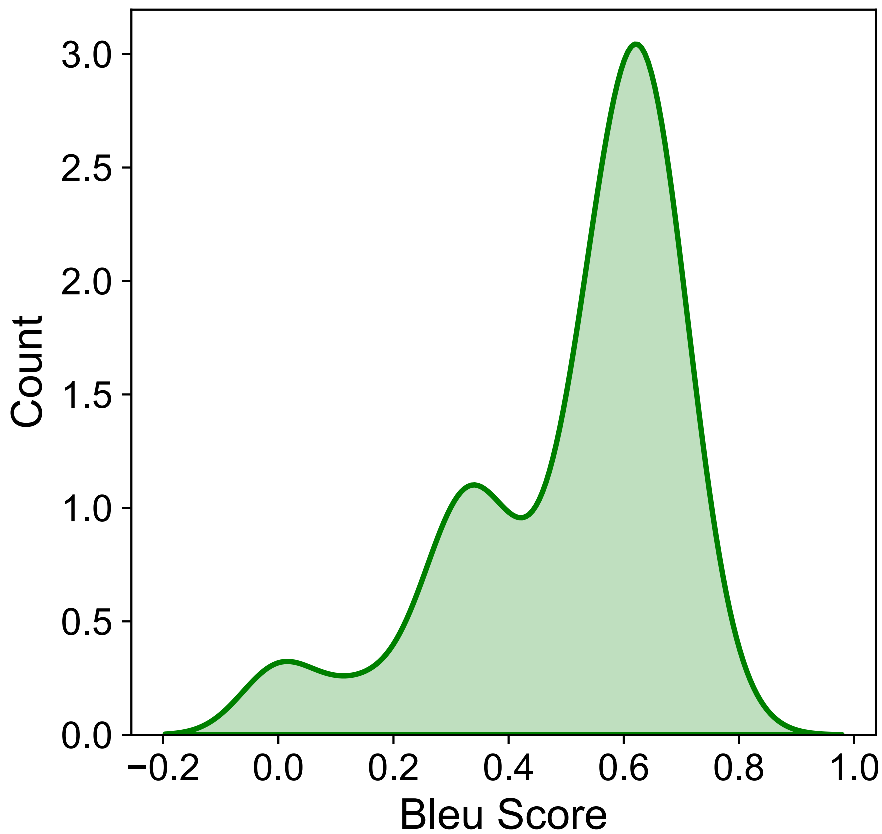
```
Mean Bleu score of predictions: 0.5096058330933692
```

The model achieves the BLEU score of 0.5096 on test set which is significantly better than that of baseline encoder decoder model. Let us check the distribution of the bleu scores.

In [9]:

```
plt.figure(figsize = (5, 5))
ax = sns.distplot(bleu_scores, hist = False, kde = True, kde_kws = {'shade': True, 'lin
plt.title("Distribution of Bleu Scores by General Model", fontdict = title_font, pad =
plt.xlabel("Bleu Score", fontdict = label_font)
plt.ylabel("Count", fontdict = label_font)
for label in (ax.get_xticklabels() + ax.get_yticklabels()):
    label.set_fontname('Arial')
    label.set_fontsize(14)
plt.show()
```

Distribution of Bleu Scores by General Model



The distribution shows that the model with general scoring function achieves the bleu score of around 0.6 for majority of the sentences but is somewhat better than the dot model. Let us generate a random prediction using this model.

In [93]:
```
print("Informal Sentence: wat r ya talkin abt")
print(f"Formal Prediction: {predict('wat r ya talkin abt', model_general)}")
```

```
Informal Sentence: wat r ya talkin abt
Formal Prediction: Can you come online ?
```

While the model did not produce the excatly accurate expected result, it did a great job to formalize the input sentence by preserving its meaning.

8. Training the Model using Concat Scoring Function:

8.1. Compiling and Fitting the model:

We can now train the model by using model fit method.

```
In [ ]: # Defining model instance with 'concat' scoring function
        model_concat  = Attention_Based_Encoder_Decoder(input_length = MAX_LEN, inp_vocab_size
                                                        out_vocab_size = vocab_size_formal, lstm_s
                                                        scoring_function = 'concat', batch_size =
                                                        embedding_dim = vocab_size_formal, embeddi
        # Compiling the model using 'adam' optimizer and custom loss function
        optimizer = tf.keras.optimizers.Adam(0.01)
        model_concat.compile(optimizer = optimizer, loss = loss_function)
        # Creating callbacks to control model training
        learning_rate_cb = tf.keras.callbacks.ReduceLROnPlateau(monitor = 'val_loss', factor =
        tensorboard_cb = create_tensorboard_cb("Model_Concat_logs")
        stopper_cb = tf.keras.callbacks.EarlyStopping(monitor = 'val_loss', patience = 2, verbo
        checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("Model_Concat.h5",
                                                           save_best_only = True, save_weights
        # Fitting the model on training data
        model_concat.fit(train_dataloader, steps_per_epoch = TRAIN_STEPS, epochs = EPOCHS,
                         callbacks = [learning_rate_cb, tensorboard_cb, stopper_cb, checkpoint_cb]
                         validation_data = validation_dataloader, validation_steps = VALID_STEPS)
        model_concat.summary()
```

```
Epoch 1/50
57/57 [==============================] - 147s 2s/step - loss: 1.0973 - val_loss: 0.9655
Epoch 2/50
57/57 [==============================] - 126s 2s/step - loss: 0.7658 - val_loss: 0.7458
Epoch 3/50
57/57 [==============================] - 127s 2s/step - loss: 0.6057 - val_loss: 0.6164
Epoch 4/50
57/57 [==============================] - 127s 2s/step - loss: 0.5203 - val_loss: 0.5795
Epoch 5/50
57/57 [==============================] - 126s 2s/step - loss: 0.4742 - val_loss: 0.5523
Epoch 6/50
57/57 [==============================] - 125s 2s/step - loss: 0.4432 - val_loss: 0.5233
Epoch 7/50
57/57 [==============================] - 125s 2s/step - loss: 0.4190 - val_loss: 0.5048
Epoch 8/50
57/57 [==============================] - 125s 2s/step - loss: 0.3997 - val_loss: 0.4880
Epoch 9/50
57/57 [==============================] - 127s 2s/step - loss: 0.3838 - val_loss: 0.4823
Epoch 10/50
57/57 [==============================] - 127s 2s/step - loss: 0.3695 - val_loss: 0.4747
Epoch 11/50
57/57 [==============================] - 125s 2s/step - loss: 0.3569 - val_loss: 0.4574
Epoch 12/50
57/57 [==============================] - 125s 2s/step - loss: 0.3482 - val_loss: 0.4640

Epoch 00012: ReduceLROnPlateau reducing learning rate to 0.004999999888241291.
Epoch 13/50
57/57 [==============================] - 127s 2s/step - loss: 0.3170 - val_loss: 0.4346
Epoch 14/50
57/57 [==============================] - 127s 2s/step - loss: 0.2972 - val_loss: 0.4300
Epoch 15/50
57/57 [==============================] - 126s 2s/step - loss: 0.2867 - val_loss: 0.4318

Epoch 00015: ReduceLROnPlateau reducing learning rate to 0.0024999999441206455.
Epoch 16/50
57/57 [==============================] - 126s 2s/step - loss: 0.2663 - val_loss: 0.4175
Epoch 17/50
57/57 [==============================] - 126s 2s/step - loss: 0.2530 - val_loss: 0.4208

Epoch 00017: ReduceLROnPlateau reducing learning rate to 0.0012499999720603228.
Epoch 18/50
57/57 [==============================] - 126s 2s/step - loss: 0.2393 - val_loss: 0.4160
Epoch 19/50
```

```
57/57 [==============================] - 126s 2s/step - loss: 0.2316 - val_loss: 0.4198

Epoch 00019: ReduceLROnPlateau reducing learning rate to 0.0006249999860301614.
Epoch 20/50
57/57 [==============================] - 126s 2s/step - loss: 0.2238 - val_loss: 0.4179

Epoch 00020: ReduceLROnPlateau reducing learning rate to 0.0003124999930150807.
Restoring model weights from the end of the best epoch.
Epoch 00020: early stopping
Model: "attention__based__encoder__decoder_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoder_3 (Encoder)          multiple                  566148
_________________________________________________________________
decoder_3 (Decoder)          multiple                  823050
=================================================================
Total params: 1,389,198
Trainable params: 1,389,198
Non-trainable params: 0
_________________________________________________________________
```

The model achieves the loss of 0.4160 on validation set which is not better than the model using general scoring function.

8.2. Calculating the BLEU Score:

We can now calculate the BLEU score to quantify the model performance.

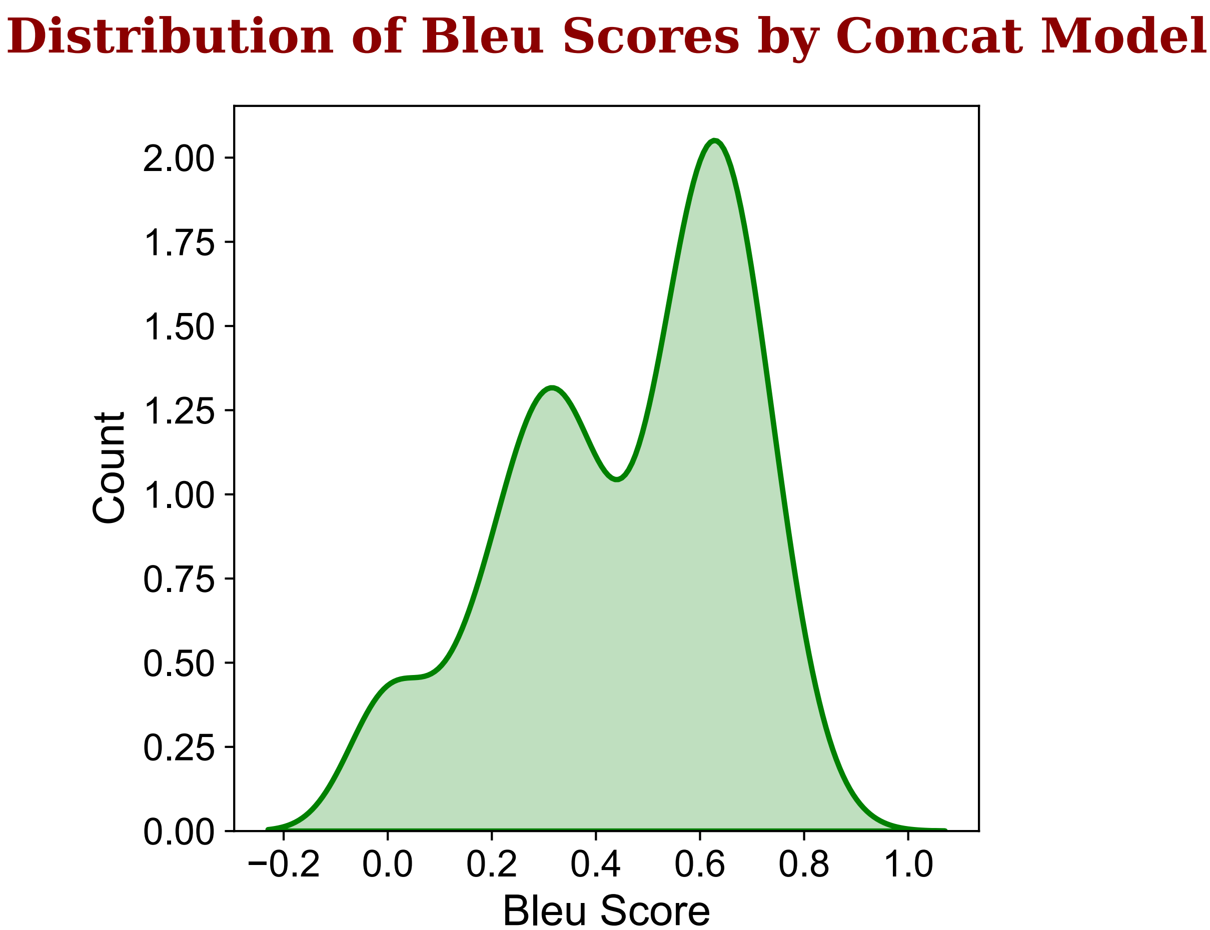
In [ ]:

```
# Removing '<' and '>' tokens and postprocessing punctuations to make plain texts
def rem(s):
    if s.startswith('<'):
        s = s[1:]
    if s.endswith('>'):
        s = s[:-1]
    return s
test['informals'] = test['encoder_inp'].apply(rem)
test['formals'] = test['decoder_inp'].apply(rem)
def predictor(s):
    # Modifing predictor using concat scoring function
    result = predict(s, model_concat)
    return result
test['predictions'] = test['informals'].apply(predictor)
# Process inputs for Bleu score
def convert_formals(s):
    return [s.split()]
def convert_predictions(s):
    return s.split()
test['formals'] = test['formals'].apply(convert_formals)
test['predictions'] = test['predictions'].apply(convert_predictions)
bleu_scores = [sentence_bleu(test['formals'].iloc[i], test['predictions'].iloc[i]) for
print(f"Mean Bleu score of predictions: {np.mean(bleu_scores)}")
```

```
Mean Bleu score of predictions: 0.45978226278062007
```

The model achieves the BLEU score of 0.459 on test set which is significantly better than that of baseline encoder decoder model. Let us check the distribution of the bleu scores.

In [12]:
```
plt.figure(figsize = (5, 5))
ax = sns.distplot(bleu_scores, hist = False, kde = True, kde_kws = {'shade': True, 'lin
plt.title("Distribution of Bleu Scores by Concat Model", fontdict = title_font, pad = 2
plt.xlabel("Bleu Score", fontdict = label_font)
plt.ylabel("Count", fontdict = label_font)
for label in (ax.get_xticklabels() + ax.get_yticklabels()):
    label.set_fontname('Arial')
    label.set_fontsize(14)
plt.show()
```



The distribution shows that the model with concat scoring function achieves the bleu score of around 0.65 for majority of the sentences but is somewhat better than the general model. Let us generate a random prediction using this model.

In [ ]:
```
print("Informal Sentence: wat r ya talkin abt")
print(f"Formal Prediction: {predict('wat r ya talkin abt', model_concat)}")
```

```
Informal Sentence: wat r ya talkin abt
Formal Prediction: What are you staying .
```

The model corrected the informal words 'wat', 'r' and 'ya' to 'what', 'are' and 'you' respectively. It also corrected the capitalization. Also, it tried to preserve the meaning to output the word 'saying' but maybe it would have missed out on the particular character based on the probabilities. The model can be improved by training on the large dataset.

9. Summary:

The model with dot scoring function did not generate a satisfactory prediction. But model with general scoring function performed excepationally well in terms of meaning preservation also it achieved the lowest validation loss. Model with concat scoring function is performing well too. While data augmentation and introduction of attention model significantly improves the performance, it can be further improved by using large dataset. For now though, the model with general scoring function is suitable for our deployment purposes.

In [ ]: