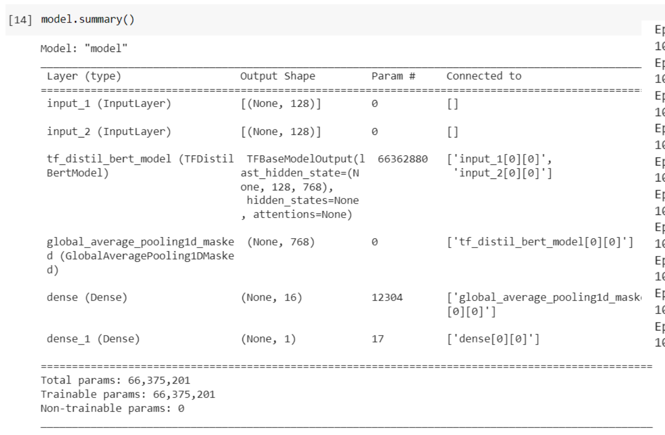
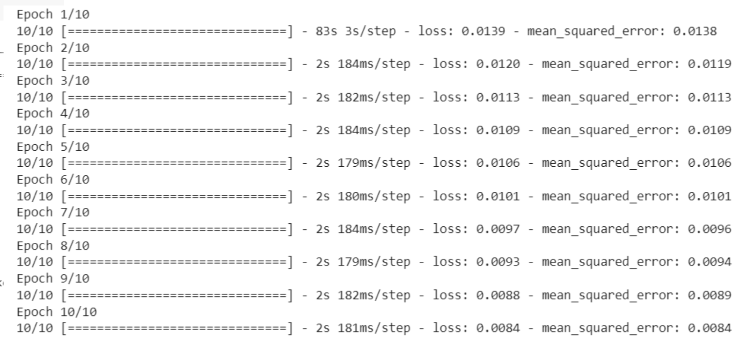
Part A: Social Media Preprocessing

1.

A. Model w/o preprocessing

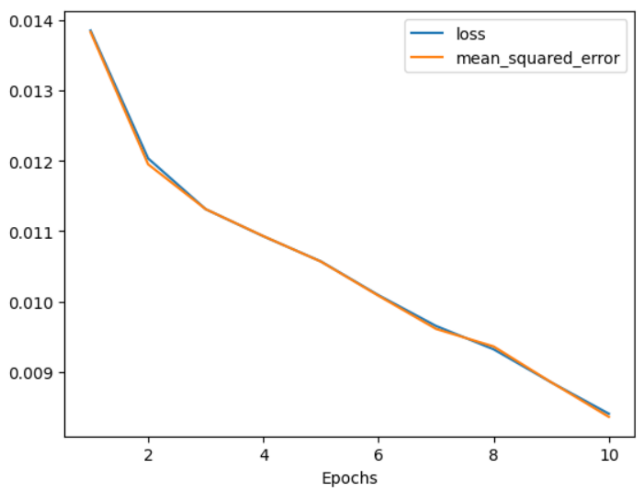


```
[14] model.summary()

Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, 128)]        0           []

 input_2 (InputLayer)           [(None, 128)]        0           []

 tf_distil_bert_model (TFDistil  TFBaseModelOutput(l  66362880   ['input_1[0][0]',
 BertModel)                     ast_hidden_state=(N               'input_2[0][0]']
                                one, 128, 768),
                                 hidden_states=None
                                , attentions=None)

 global_average_pooling1d_maske  (None, 768)         0           ['tf_distil_bert_model[0][0]']
 d (GlobalAveragePooling1DMaske
 d)

 dense (Dense)                  (None, 16)           12304       ['global_average_pooling1d_mask
                                                                 [0][0]']

 dense_1 (Dense)                (None, 1)            17          ['dense[0][0]']

==================================================================================================
Total params: 66,375,201
Trainable params: 66,375,201
Non-trainable params: 0
__________________________________________________________________________________________________
```



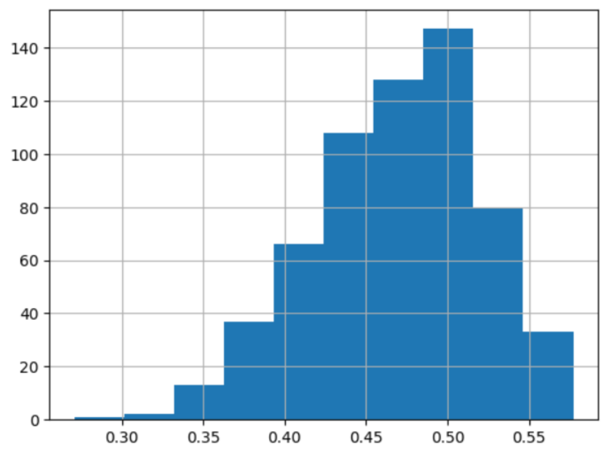
```
Epoch 1/10
10/10 [==============================] - 83s 3s/step - loss: 0.0139 - mean_squared_error: 0.0138
Epoch 2/10
10/10 [==============================] - 2s 184ms/step - loss: 0.0120 - mean_squared_error: 0.0119
Epoch 3/10
10/10 [==============================] - 2s 182ms/step - loss: 0.0113 - mean_squared_error: 0.0113
Epoch 4/10
10/10 [==============================] - 2s 184ms/step - loss: 0.0109 - mean_squared_error: 0.0109
Epoch 5/10
10/10 [==============================] - 2s 179ms/step - loss: 0.0106 - mean_squared_error: 0.0106
Epoch 6/10
10/10 [==============================] - 2s 180ms/step - loss: 0.0101 - mean_squared_error: 0.0101
Epoch 7/10
10/10 [==============================] - 2s 184ms/step - loss: 0.0097 - mean_squared_error: 0.0096
Epoch 8/10
10/10 [==============================] - 2s 179ms/step - loss: 0.0093 - mean_squared_error: 0.0094
Epoch 9/10
10/10 [==============================] - 2s 182ms/step - loss: 0.0088 - mean_squared_error: 0.0089
Epoch 10/10
10/10 [==============================] - 2s 181ms/step - loss: 0.0084 - mean_squared_error: 0.0084
```

```
20/20 [==============================] - 10s 131ms/step - loss: 0.0134 - mean_squared_error: 0.0129
Test loss: 0.013364320620894432
Test MSE: 0.012948649935424328
```



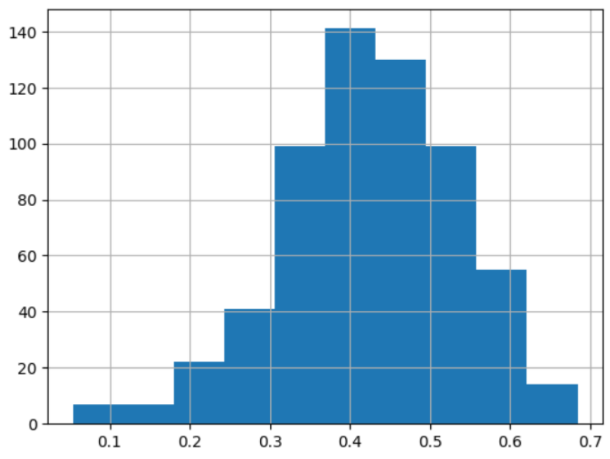
```
pd.Series(preds).hist()
```

```
<Axes: >
```



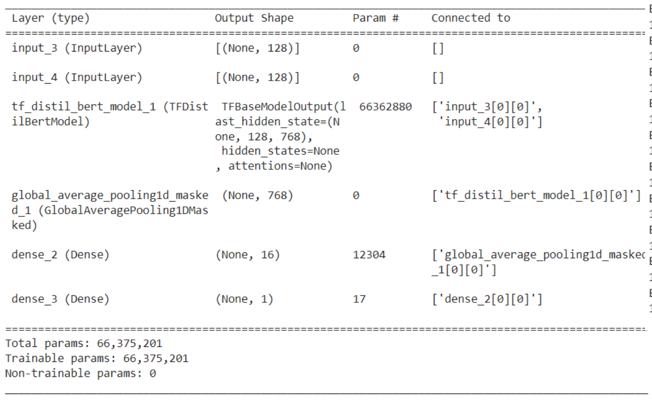
```
pd.Series(test_targets).hist()
```

```
<Axes: >
```

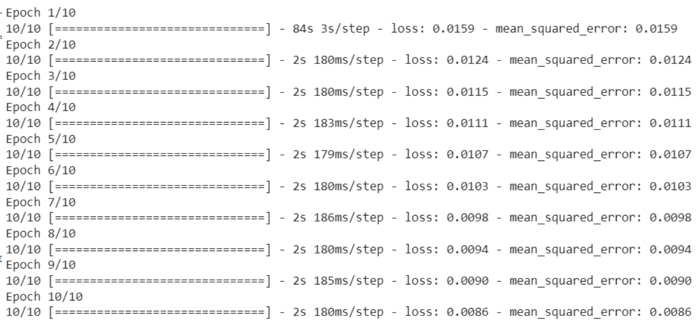


The range of the gold humour ratings is 0.63 and the range of the predicted humour ratings is 0.30617702. The predicted humour ratings are half the range of the gold humour ratings. This means that the spread of the predictions are much smaller than the spread of the gold humour ratings. The mean of the gold humour ratings is 0.423834146341463444 and the predicted mean is 0.46906206. This is an increase of over 10% over the gold mean. So, the predictions are on average 10% higher than they should be. So the model on average overpredicts the humour rating by 10% and the predictions are more centered around the mean (I.e., less spread) than the gold humour ratings.

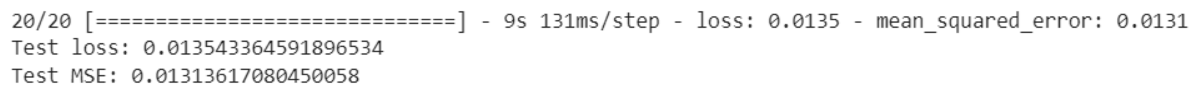
B. Model w preprocessing



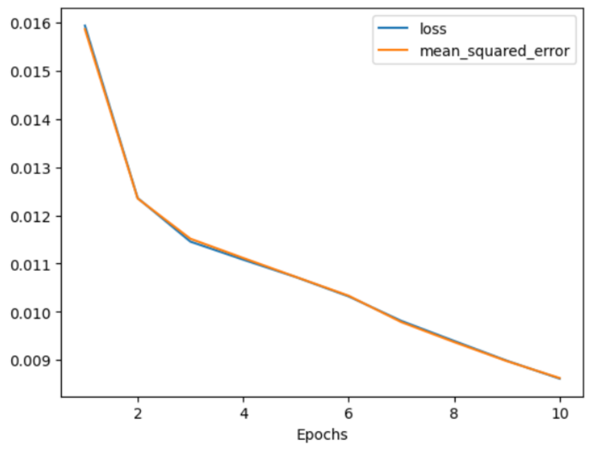
```
____________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
====================================================================================================
 input_3 (InputLayer)           [(None, 128)]        0           []

 input_4 (InputLayer)           [(None, 128)]        0           []

 tf_distil_bert_model_1 (TFDist  TFBaseModelOutput(l  66362880   ['input_3[0][0]',
 ilBertModel)                   ast_hidden_state=(N               'input_4[0][0]']
                                one, 128, 768),
                                 hidden_states=None
                                , attentions=None)

 global_average_pooling1d_maske  (None, 768)         0           ['tf_distil_bert_model_1[0][0]']
 d_1 (GlobalAveragePooling1DMas
 ked)

 dense_2 (Dense)                (None, 16)           12304       ['global_average_pooling1d_maskec
                                                                 _1[0][0]']

 dense_3 (Dense)                (None, 1)            17          ['dense_2[0][0]']

====================================================================================================
Total params: 66,375,201
Trainable params: 66,375,201
Non-trainable params: 0
____________________________________________________________________________________________________
```



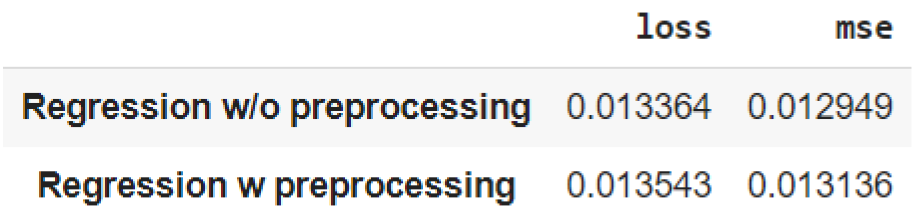
```
Epoch 1/10
10/10 [==============================] - 84s 3s/step - loss: 0.0159 - mean_squared_error: 0.0159
Epoch 2/10
10/10 [==============================] - 2s 180ms/step - loss: 0.0124 - mean_squared_error: 0.0124
Epoch 3/10
10/10 [==============================] - 2s 180ms/step - loss: 0.0115 - mean_squared_error: 0.0115
Epoch 4/10
10/10 [==============================] - 2s 183ms/step - loss: 0.0111 - mean_squared_error: 0.0111
Epoch 5/10
10/10 [==============================] - 2s 179ms/step - loss: 0.0107 - mean_squared_error: 0.0107
Epoch 6/10
10/10 [==============================] - 2s 180ms/step - loss: 0.0103 - mean_squared_error: 0.0103
Epoch 7/10
10/10 [==============================] - 2s 186ms/step - loss: 0.0098 - mean_squared_error: 0.0098
Epoch 8/10
10/10 [==============================] - 2s 180ms/step - loss: 0.0094 - mean_squared_error: 0.0094
Epoch 9/10
10/10 [==============================] - 2s 185ms/step - loss: 0.0090 - mean_squared_error: 0.0090
Epoch 10/10
10/10 [==============================] - 2s 180ms/step - loss: 0.0086 - mean_squared_error: 0.0086
```



```
20/20 [==============================] - 9s 131ms/step - loss: 0.0135 - mean_squared_error: 0.0131
Test loss: 0.013543364591896534
Test MSE: 0.01313617080450058
```



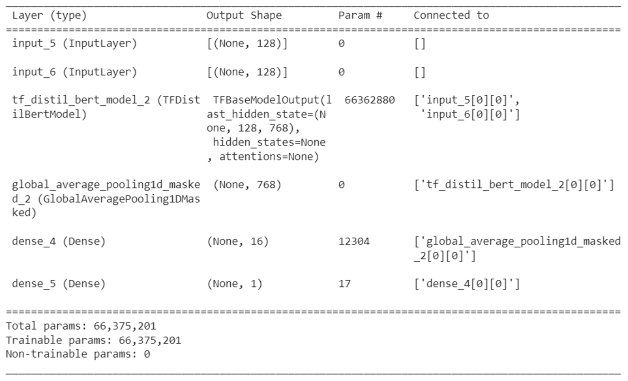
As the table below shows preprocessing doesn't make a significant impact, with it actually resulting in a slightly worse performance than without preprocessing. Preprocessing reduces the number of different types. As expected this results in worse performance due to the unpreprocessed text holding information that the BERT layer can pick out, as such these features that are processed and removed are clearly useful.



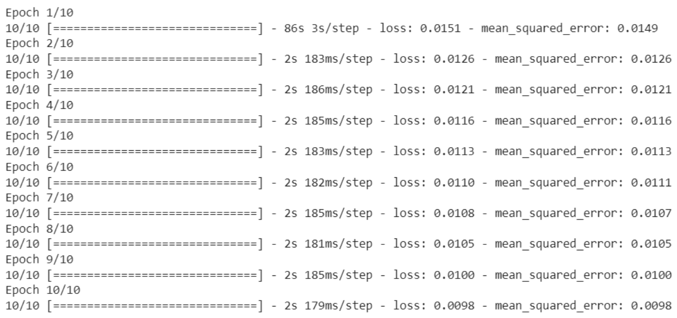
	loss	mse
Regression w/o preprocessing	0.013364	0.012949
Regression w preprocessing	0.013543	0.013136

2.

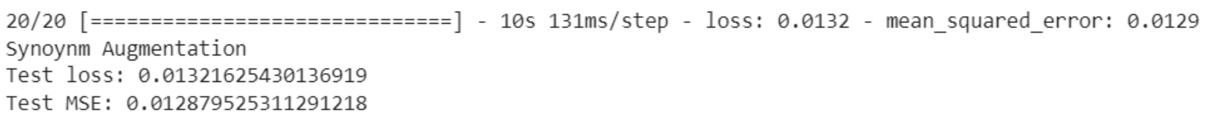
A. Synonym Aug



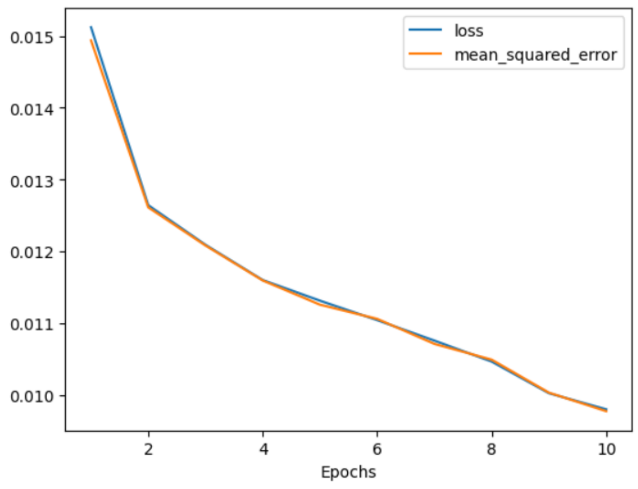
```
____________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
============================================================================================
 input_5 (InputLayer)           [(None, 128)]        0           []

 input_6 (InputLayer)           [(None, 128)]        0           []

 tf_distil_bert_model_2 (TFDist  TFBaseModelOutput(l  66362880    ['input_5[0][0]',
 ilBertModel)                   ast_hidden_state=(N                'input_6[0][0]']
                                one, 128, 768),
                                 hidden_states=None
                                , attentions=None)

 global_average_pooling1d_maske  (None, 768)         0           ['tf_distil_bert_model_2[0][0]']
 d_2 (GlobalAveragePooling1DMas
 ked)

 dense_4 (Dense)                (None, 16)           12304       ['global_average_pooling1d_masked
                                                                 _2[0][0]']

 dense_5 (Dense)                (None, 1)            17          ['dense_4[0][0]']

============================================================================================
Total params: 66,375,201
Trainable params: 66,375,201
Non-trainable params: 0
____________________________________________________________________________________________
```



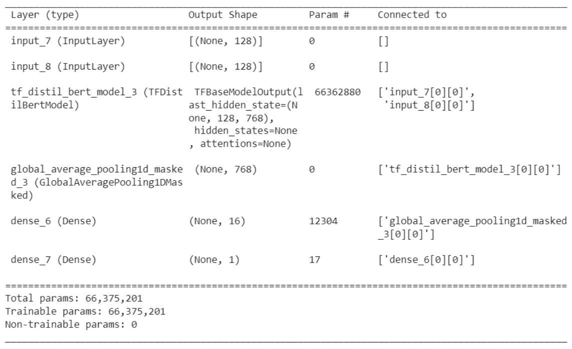
```
Epoch 1/10
10/10 [==============================] - 86s 3s/step - loss: 0.0151 - mean_squared_error: 0.0149
Epoch 2/10
10/10 [==============================] - 2s 183ms/step - loss: 0.0126 - mean_squared_error: 0.0126
Epoch 3/10
10/10 [==============================] - 2s 186ms/step - loss: 0.0121 - mean_squared_error: 0.0121
Epoch 4/10
10/10 [==============================] - 2s 185ms/step - loss: 0.0116 - mean_squared_error: 0.0116
Epoch 5/10
10/10 [==============================] - 2s 183ms/step - loss: 0.0113 - mean_squared_error: 0.0113
Epoch 6/10
10/10 [==============================] - 2s 182ms/step - loss: 0.0110 - mean_squared_error: 0.0111
Epoch 7/10
10/10 [==============================] - 2s 185ms/step - loss: 0.0108 - mean_squared_error: 0.0107
Epoch 8/10
10/10 [==============================] - 2s 181ms/step - loss: 0.0105 - mean_squared_error: 0.0105
Epoch 9/10
10/10 [==============================] - 2s 185ms/step - loss: 0.0100 - mean_squared_error: 0.0100
Epoch 10/10
10/10 [==============================] - 2s 179ms/step - loss: 0.0098 - mean_squared_error: 0.0098
```



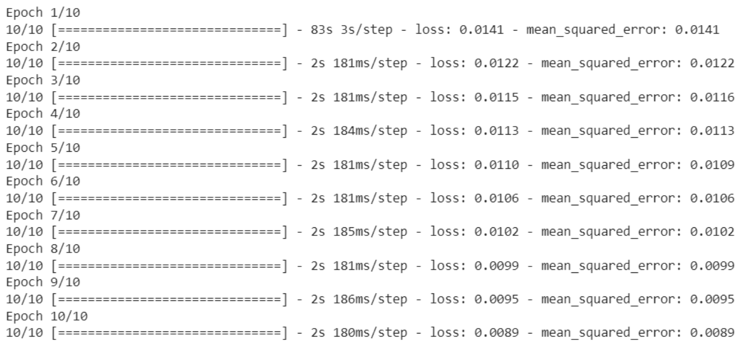
```
20/20 [==============================] - 10s 131ms/step - loss: 0.0132 - mean_squared_error: 0.0129
Synoynm Augmentation
Test loss: 0.01321625430136919
Test MSE: 0.012879525311291218
```



B. Random Aug

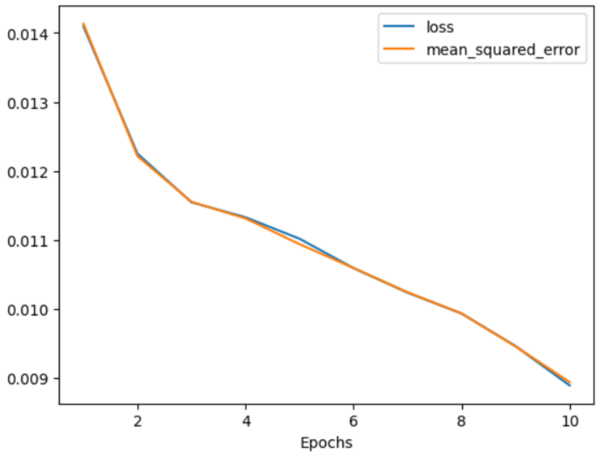


```
____________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
============================================================================================
 input_7 (InputLayer)           [(None, 128)]        0           []

 input_8 (InputLayer)           [(None, 128)]        0           []

 tf_distil_bert_model_3 (TFDist  TFBaseModelOutput(l  66362880    ['input_7[0][0]',
 ilBertModel)                   ast_hidden_state=(N                'input_8[0][0]']
                                one, 128, 768),
                                 hidden_states=None
                                , attentions=None)

 global_average_pooling1d_maske  (None, 768)         0           ['tf_distil_bert_model_3[0][0]']
 d_3 (GlobalAveragePooling1DMas
 ked)

 dense_6 (Dense)                (None, 16)           12304       ['global_average_pooling1d_masked
                                                                 _3[0][0]']

 dense_7 (Dense)                (None, 1)            17          ['dense_6[0][0]']

============================================================================================
Total params: 66,375,201
Trainable params: 66,375,201
Non-trainable params: 0
____________________________________________________________________________________________
```



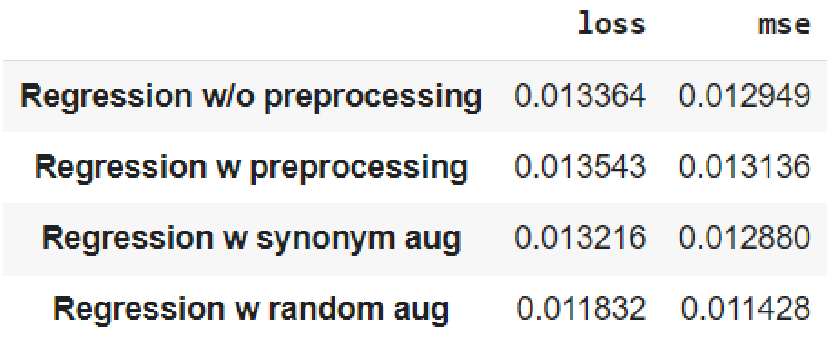
```
Epoch 1/10
10/10 [==============================] - 83s 3s/step - loss: 0.0141 - mean_squared_error: 0.0141
Epoch 2/10
10/10 [==============================] - 2s 181ms/step - loss: 0.0122 - mean_squared_error: 0.0122
Epoch 3/10
10/10 [==============================] - 2s 181ms/step - loss: 0.0115 - mean_squared_error: 0.0116
Epoch 4/10
10/10 [==============================] - 2s 184ms/step - loss: 0.0113 - mean_squared_error: 0.0113
Epoch 5/10
10/10 [==============================] - 2s 181ms/step - loss: 0.0110 - mean_squared_error: 0.0109
Epoch 6/10
10/10 [==============================] - 2s 181ms/step - loss: 0.0106 - mean_squared_error: 0.0106
Epoch 7/10
10/10 [==============================] - 2s 185ms/step - loss: 0.0102 - mean_squared_error: 0.0102
Epoch 8/10
10/10 [==============================] - 2s 181ms/step - loss: 0.0099 - mean_squared_error: 0.0099
Epoch 9/10
10/10 [==============================] - 2s 186ms/step - loss: 0.0095 - mean_squared_error: 0.0095
Epoch 10/10
10/10 [==============================] - 2s 180ms/step - loss: 0.0089 - mean_squared_error: 0.0089
```

```
20/20 [==============================] - 9s 132ms/step - loss: 0.0118 - mean_squared_error: 0.0114
Random Augmentation
Test loss: 0.011831956915557384
Test MSE: 0.011428473517298698
```



Random Aug is good and results in a smaller loss and mse.

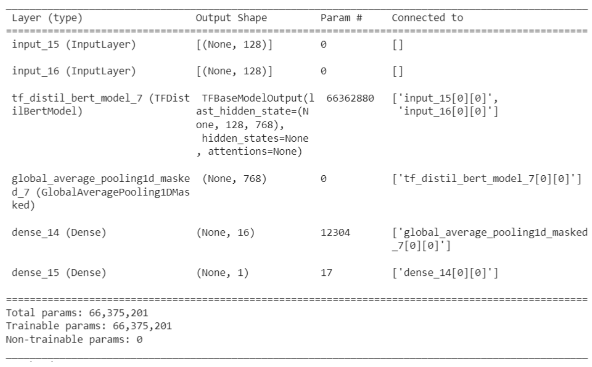
The table below shows the performance of the models trained above. I will look at the last 2, with augmenting. Random Aug produces better results than Synonym Aug. This improvement is likely due to the fact that Random Augmentation results in a larger training and more diversified training set. The Random augmentation is also less overfitting than Synonym Augmentation, so using random aug can help prevent overfitting.



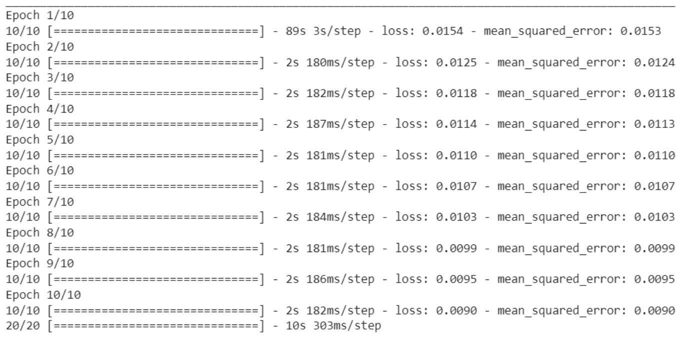
	loss	mse
Regression w/o preprocessing	0.013364	0.012949
Regression w preprocessing	0.013543	0.013136
Regression w synonym aug	0.013216	0.012880
Regression w random aug	0.011832	0.011428

3.

Model 1

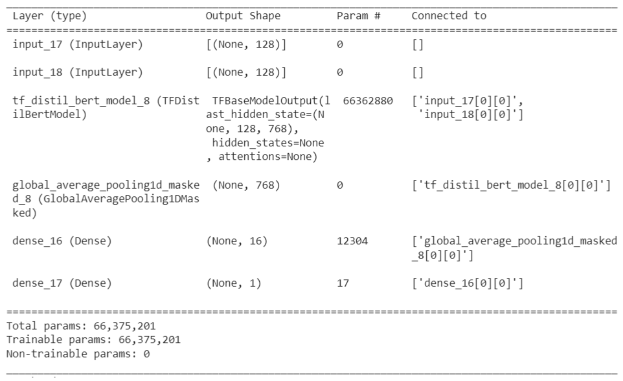


```
 Layer (type)                Output Shape              Param #     Connected to
==================================================================================================
 input_15 (InputLayer)       [(None, 128)]             0           []

 input_16 (InputLayer)       [(None, 128)]             0           []

 tf_distil_bert_model_7 (TFDist  TFBaseModelOutput(l  66362880    ['input_15[0][0]',
 ilBertModel)                ast_hidden_state=(N                   'input_16[0][0]']
                             one, 128, 768),
                              hidden_states=None
                             , attentions=None)

 global_average_pooling1d_maske  (None, 768)          0           ['tf_distil_bert_model_7[0][0]']
 d_7 (GlobalAveragePooling1DMas
 ked)

 dense_14 (Dense)            (None, 16)                12304       ['global_average_pooling1d_masked
                                                                   _7[0][0]']

 dense_15 (Dense)            (None, 1)                 17          ['dense_14[0][0]']

==================================================================================================
Total params: 66,375,201
Trainable params: 66,375,201
Non-trainable params: 0
```

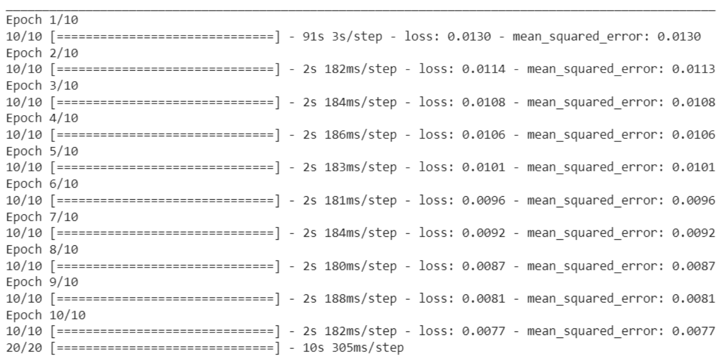


```
Epoch 1/10
10/10 [==============================] - 89s 3s/step - loss: 0.0154 - mean_squared_error: 0.0153
Epoch 2/10
10/10 [==============================] - 2s 180ms/step - loss: 0.0125 - mean_squared_error: 0.0124
Epoch 3/10
10/10 [==============================] - 2s 182ms/step - loss: 0.0118 - mean_squared_error: 0.0118
Epoch 4/10
10/10 [==============================] - 2s 187ms/step - loss: 0.0114 - mean_squared_error: 0.0113
Epoch 5/10
10/10 [==============================] - 2s 181ms/step - loss: 0.0110 - mean_squared_error: 0.0110
Epoch 6/10
10/10 [==============================] - 2s 181ms/step - loss: 0.0107 - mean_squared_error: 0.0107
Epoch 7/10
10/10 [==============================] - 2s 184ms/step - loss: 0.0103 - mean_squared_error: 0.0103
Epoch 8/10
10/10 [==============================] - 2s 181ms/step - loss: 0.0099 - mean_squared_error: 0.0099
Epoch 9/10
10/10 [==============================] - 2s 186ms/step - loss: 0.0095 - mean_squared_error: 0.0095
Epoch 10/10
10/10 [==============================] - 2s 182ms/step - loss: 0.0090 - mean_squared_error: 0.0090
20/20 [==============================] - 10s 303ms/step
```

Model 2

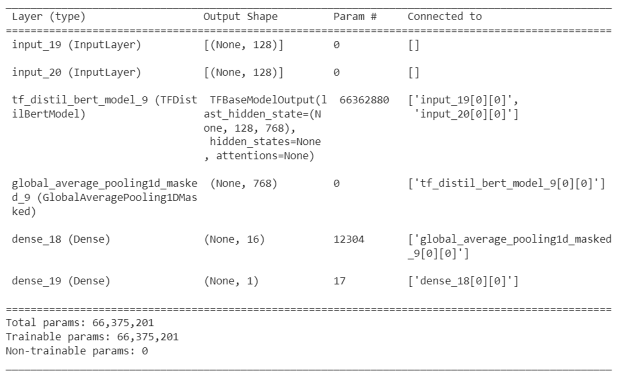


```
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_17 (InputLayer)          [(None, 128)]        0           []

 input_18 (InputLayer)          [(None, 128)]        0           []

 tf_distil_bert_model_8 (TFDist  TFBaseModelOutput(l  66362880    ['input_17[0][0]',
 ilBertModel)                   ast_hidden_state=(N               'input_18[0][0]']
                                one, 128, 768),
                                 hidden_states=None
                                , attentions=None)

 global_average_pooling1d_maske  (None, 768)         0           ['tf_distil_bert_model_8[0][0]']
 d_8 (GlobalAveragePooling1DMas
 ked)

 dense_16 (Dense)               (None, 16)           12304       ['global_average_pooling1d_masked
                                                                 _8[0][0]']

 dense_17 (Dense)               (None, 1)            17          ['dense_16[0][0]']

==================================================================================================
Total params: 66,375,201
Trainable params: 66,375,201
Non-trainable params: 0
__________________________________________________________________________________________________
```

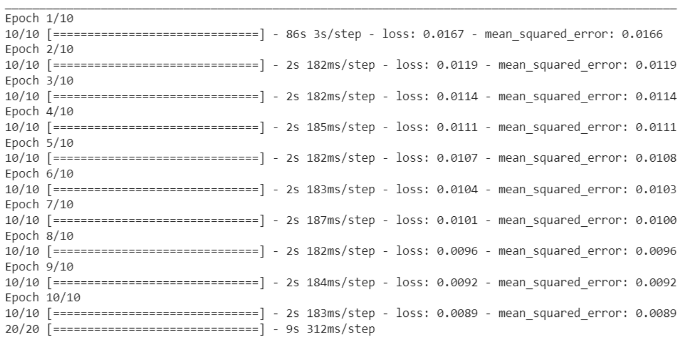


```
Epoch 1/10
10/10 [==============================] - 91s 3s/step - loss: 0.0130 - mean_squared_error: 0.0130
Epoch 2/10
10/10 [==============================] - 2s 182ms/step - loss: 0.0114 - mean_squared_error: 0.0113
Epoch 3/10
10/10 [==============================] - 2s 184ms/step - loss: 0.0108 - mean_squared_error: 0.0108
Epoch 4/10
10/10 [==============================] - 2s 186ms/step - loss: 0.0106 - mean_squared_error: 0.0106
Epoch 5/10
10/10 [==============================] - 2s 183ms/step - loss: 0.0101 - mean_squared_error: 0.0101
Epoch 6/10
10/10 [==============================] - 2s 181ms/step - loss: 0.0096 - mean_squared_error: 0.0096
Epoch 7/10
10/10 [==============================] - 2s 184ms/step - loss: 0.0092 - mean_squared_error: 0.0092
Epoch 8/10
10/10 [==============================] - 2s 180ms/step - loss: 0.0087 - mean_squared_error: 0.0087
Epoch 9/10
10/10 [==============================] - 2s 188ms/step - loss: 0.0081 - mean_squared_error: 0.0081
Epoch 10/10
10/10 [==============================] - 2s 182ms/step - loss: 0.0077 - mean_squared_error: 0.0077
20/20 [==============================] - 10s 305ms/step
```

Model 3

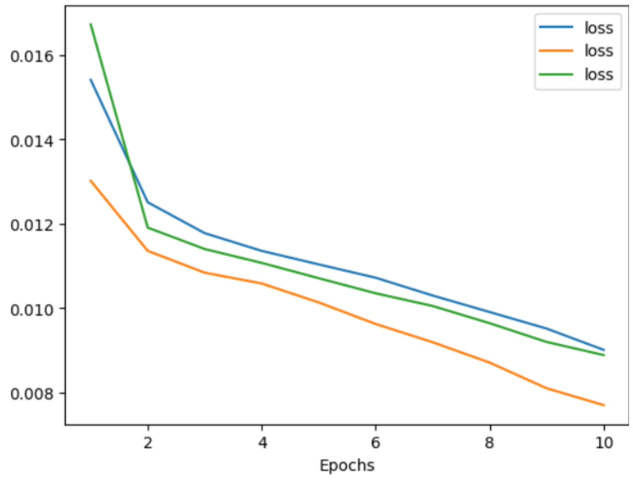


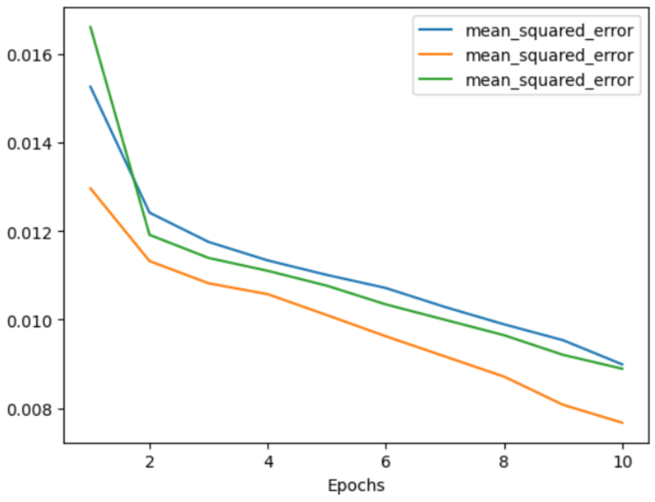
```
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_19 (InputLayer)          [(None, 128)]        0           []

 input_20 (InputLayer)          [(None, 128)]        0           []

 tf_distil_bert_model_9 (TFDist  TFBaseModelOutput(l  66362880    ['input_19[0][0]',
 ilBertModel)                   ast_hidden_state=(N               'input_20[0][0]']
                                one, 128, 768),
                                 hidden_states=None
                                , attentions=None)

 global_average_pooling1d_maske  (None, 768)         0           ['tf_distil_bert_model_9[0][0]']
 d_9 (GlobalAveragePooling1DMas
 ked)

 dense_18 (Dense)               (None, 16)           12304       ['global_average_pooling1d_masked
                                                                 _9[0][0]']

 dense_19 (Dense)               (None, 1)            17          ['dense_18[0][0]']

==================================================================================================
Total params: 66,375,201
Trainable params: 66,375,201
Non-trainable params: 0
__________________________________________________________________________________________________
```

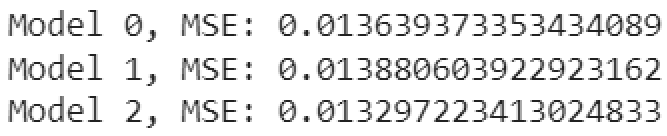


```
Epoch 1/10
10/10 [==============================] - 86s 3s/step - loss: 0.0167 - mean_squared_error: 0.0166
Epoch 2/10
10/10 [==============================] - 2s 182ms/step - loss: 0.0119 - mean_squared_error: 0.0119
Epoch 3/10
10/10 [==============================] - 2s 182ms/step - loss: 0.0114 - mean_squared_error: 0.0114
Epoch 4/10
10/10 [==============================] - 2s 185ms/step - loss: 0.0111 - mean_squared_error: 0.0111
Epoch 5/10
10/10 [==============================] - 2s 182ms/step - loss: 0.0107 - mean_squared_error: 0.0108
Epoch 6/10
10/10 [==============================] - 2s 183ms/step - loss: 0.0104 - mean_squared_error: 0.0103
Epoch 7/10
10/10 [==============================] - 2s 187ms/step - loss: 0.0101 - mean_squared_error: 0.0100
Epoch 8/10
10/10 [==============================] - 2s 182ms/step - loss: 0.0096 - mean_squared_error: 0.0096
Epoch 9/10
10/10 [==============================] - 2s 184ms/step - loss: 0.0092 - mean_squared_error: 0.0092
Epoch 10/10
10/10 [==============================] - 2s 183ms/step - loss: 0.0089 - mean_squared_error: 0.0089
20/20 [==============================] - 9s 312ms/step
```

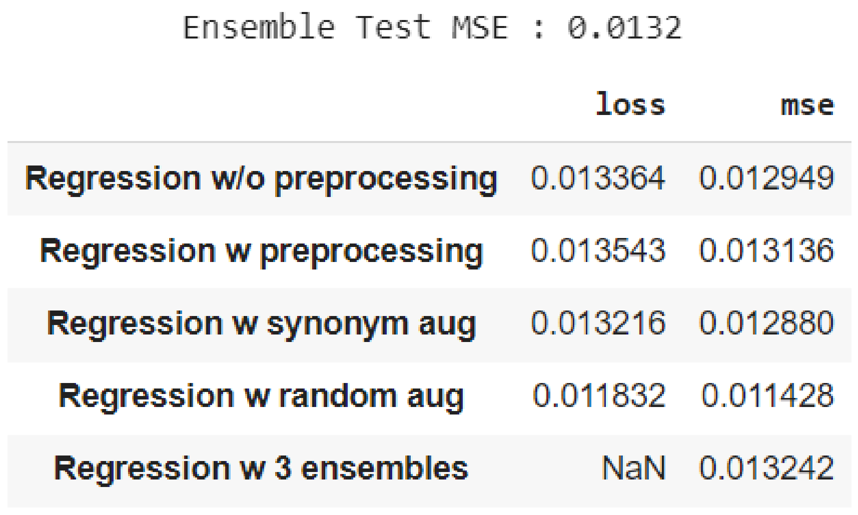
Ensemble of 3 models







```
Model 0, MSE: 0.013639373353434089
Model 1, MSE: 0.013880603922923162
Model 2, MSE: 0.013297223413024833
```



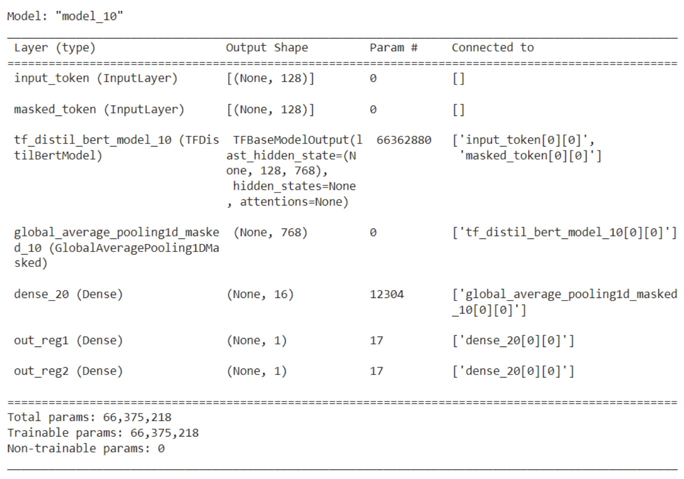
```
Ensemble Test MSE : 0.0132
```

	loss	mse
Regression w/o preprocessing	0.013364	0.012949
Regression w preprocessing	0.013543	0.013136
Regression w synonym aug	0.013216	0.012880
Regression w random aug	0.011832	0.011428
Regression w 3 ensembles	NaN	0.013242

Comparing the first and last rows, we see that ensembling doesn't actually result in an improved performance in this case. In general, I do not believe this to be the case. Ensembling regressors work by taking the mean of the outputs of the regressors. This means that if a single regressor performs badly then we can mitigate its effects by ensembling. It allows for a more stable and predictable result, so that a model stuck in a local optima doesn't produce bad results. Note that this model also takes longer to train in the currently used sequential fashion. This is evidently due to the fact we have to train multiple models.

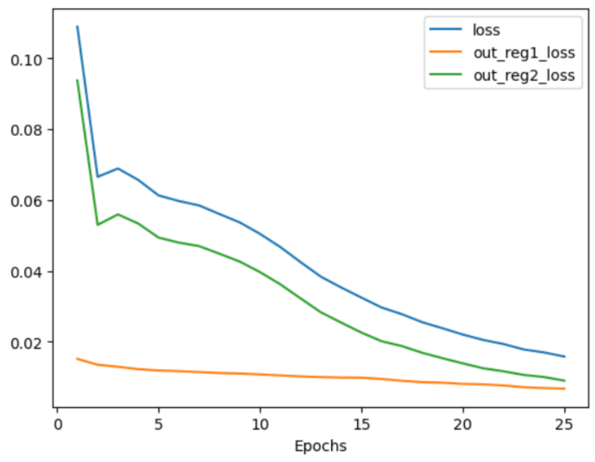
4.

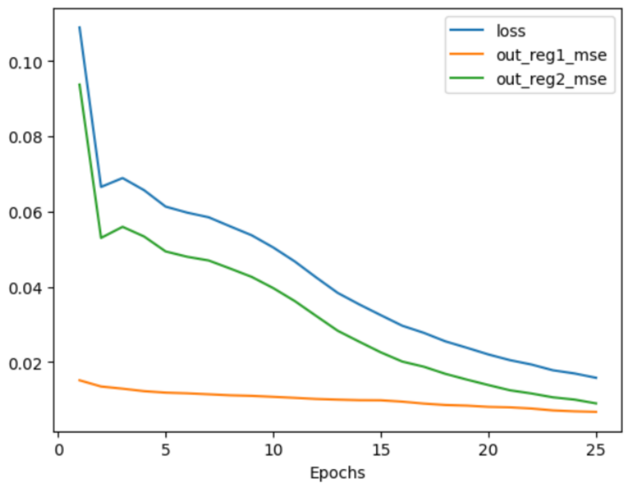
A.



```
Model: "model_10"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to                     
==================================================================================================
 input_token (InputLayer)       [(None, 128)]        0           []                               
                                                                                                  
 masked_token (InputLayer)      [(None, 128)]        0           []                               
                                                                                                  
 tf_distil_bert_model_10 (TFDis  TFBaseModelOutput(l  66362880   ['input_token[0][0]',            
 tilBertModel)                  ast_hidden_state=(N               'masked_token[0][0]']           
                                one, 128, 768),                                                   
                                 hidden_states=None                                               
                                , attentions=None)                                                
                                                                                                  
 global_average_pooling1d_maske  (None, 768)         0           ['tf_distil_bert_model_10[0][0]']
 d_10 (GlobalAveragePooling1DMa                                                                   
 sked)                                                                                            
                                                                                                  
 dense_20 (Dense)               (None, 16)           12304       ['global_average_pooling1d_masked
                                                                 _10[0][0]']                      
                                                                                                  
 out_reg1 (Dense)               (None, 1)            17          ['dense_20[0][0]']               
                                                                                                  
 out_reg2 (Dense)               (None, 1)            17          ['dense_20[0][0]']               
                                                                                                  
==================================================================================================
Total params: 66,375,218
Trainable params: 66,375,218
Non-trainable params: 0
__________________________________________________________________________________________________
```

```
Epoch 1/25
5/5 [==============================] - 95s 8s/step - loss: 0.1089 - out_reg1_loss: 0.0152 - out_reg2_loss: 0.0937 - out_reg1_mse: 0.0152 - out_reg2_mse: 0.0937
Epoch 2/25
5/5 [==============================] - 1s 193ms/step - loss: 0.0665 - out_reg1_loss: 0.0135 - out_reg2_loss: 0.0530 - out_reg1_mse: 0.0135 - out_reg2_mse: 0.0530
Epoch 3/25
5/5 [==============================] - 1s 191ms/step - loss: 0.0689 - out_reg1_loss: 0.0130 - out_reg2_loss: 0.0559 - out_reg1_mse: 0.0130 - out_reg2_mse: 0.0559
Epoch 4/25
5/5 [==============================] - 1s 194ms/step - loss: 0.0657 - out_reg1_loss: 0.0123 - out_reg2_loss: 0.0534 - out_reg1_mse: 0.0123 - out_reg2_mse: 0.0534
Epoch 5/25
5/5 [==============================] - 1s 191ms/step - loss: 0.0613 - out_reg1_loss: 0.0119 - out_reg2_loss: 0.0494 - out_reg1_mse: 0.0119 - out_reg2_mse: 0.0494
Epoch 6/25
5/5 [==============================] - 1s 194ms/step - loss: 0.0597 - out_reg1_loss: 0.0117 - out_reg2_loss: 0.0480 - out_reg1_mse: 0.0117 - out_reg2_mse: 0.0480
Epoch 7/25
5/5 [==============================] - 1s 192ms/step - loss: 0.0585 - out_reg1_loss: 0.0115 - out_reg2_loss: 0.0470 - out_reg1_mse: 0.0115 - out_reg2_mse: 0.0470
Epoch 8/25
5/5 [==============================] - 1s 193ms/step - loss: 0.0561 - out_reg1_loss: 0.0112 - out_reg2_loss: 0.0449 - out_reg1_mse: 0.0112 - out_reg2_mse: 0.0449
Epoch 9/25
5/5 [==============================] - 1s 197ms/step - loss: 0.0537 - out_reg1_loss: 0.0110 - out_reg2_loss: 0.0426 - out_reg1_mse: 0.0110 - out_reg2_mse: 0.0426
Epoch 10/25
5/5 [==============================] - 1s 207ms/step - loss: 0.0505 - out_reg1_loss: 0.0108 - out_reg2_loss: 0.0397 - out_reg1_mse: 0.0108 - out_reg2_mse: 0.0397
Epoch 11/25
5/5 [==============================] - 1s 194ms/step - loss: 0.0467 - out_reg1_loss: 0.0105 - out_reg2_loss: 0.0362 - out_reg1_mse: 0.0105 - out_reg2_mse: 0.0362
Epoch 12/25
5/5 [==============================] - 1s 197ms/step - loss: 0.0425 - out_reg1_loss: 0.0102 - out_reg2_loss: 0.0323 - out_reg1_mse: 0.0102 - out_reg2_mse: 0.0323
Epoch 13/25
5/5 [==============================] - 1s 190ms/step - loss: 0.0384 - out_reg1_loss: 0.0100 - out_reg2_loss: 0.0283 - out_reg1_mse: 0.0100 - out_reg2_mse: 0.0283
Epoch 14/25
5/5 [==============================] - 1s 202ms/step - loss: 0.0353 - out_reg1_loss: 0.0099 - out_reg2_loss: 0.0254 - out_reg1_mse: 0.0099 - out_reg2_mse: 0.0254
Epoch 15/25
5/5 [==============================] - 1s 189ms/step - loss: 0.0325 - out_reg1_loss: 0.0099 - out_reg2_loss: 0.0226 - out_reg1_mse: 0.0099 - out_reg2_mse: 0.0226
Epoch 16/25
5/5 [==============================] - 1s 194ms/step - loss: 0.0297 - out_reg1_loss: 0.0095 - out_reg2_loss: 0.0202 - out_reg1_mse: 0.0095 - out_reg2_mse: 0.0202
Epoch 17/25
5/5 [==============================] - 1s 190ms/step - loss: 0.0278 - out_reg1_loss: 0.0090 - out_reg2_loss: 0.0188 - out_reg1_mse: 0.0090 - out_reg2_mse: 0.0188
Epoch 18/25
5/5 [==============================] - 1s 192ms/step - loss: 0.0255 - out_reg1_loss: 0.0086 - out_reg2_loss: 0.0169 - out_reg1_mse: 0.0086 - out_reg2_mse: 0.0169
Epoch 19/25
5/5 [==============================] - 1s 200ms/step - loss: 0.0238 - out_reg1_loss: 0.0085 - out_reg2_loss: 0.0154 - out_reg1_mse: 0.0085 - out_reg2_mse: 0.0154
Epoch 20/25
5/5 [==============================] - 1s 192ms/step - loss: 0.0220 - out_reg1_loss: 0.0081 - out_reg2_loss: 0.0139 - out_reg1_mse: 0.0081 - out_reg2_mse: 0.0139
Epoch 21/25
5/5 [==============================] - 1s 192ms/step - loss: 0.0205 - out_reg1_loss: 0.0080 - out_reg2_loss: 0.0125 - out_reg1_mse: 0.0080 - out_reg2_mse: 0.0125
Epoch 22/25
5/5 [==============================] - 1s 248ms/step - loss: 0.0194 - out_reg1_loss: 0.0077 - out_reg2_loss: 0.0117 - out_reg1_mse: 0.0077 - out_reg2_mse: 0.0117
Epoch 23/25
5/5 [==============================] - 1s 200ms/step - loss: 0.0178 - out_reg1_loss: 0.0072 - out_reg2_loss: 0.0107 - out_reg1_mse: 0.0072 - out_reg2_mse: 0.0107
Epoch 24/25
5/5 [==============================] - 1s 195ms/step - loss: 0.0170 - out_reg1_loss: 0.0069 - out_reg2_loss: 0.0101 - out_reg1_mse: 0.0069 - out_reg2_mse: 0.0101
Epoch 25/25
5/5 [==============================] - 1s 194ms/step - loss: 0.0158 - out_reg1_loss: 0.0068 - out_reg2_loss: 0.0091 - out_reg1_mse: 0.0068 - out_reg2_mse: 0.0091
```



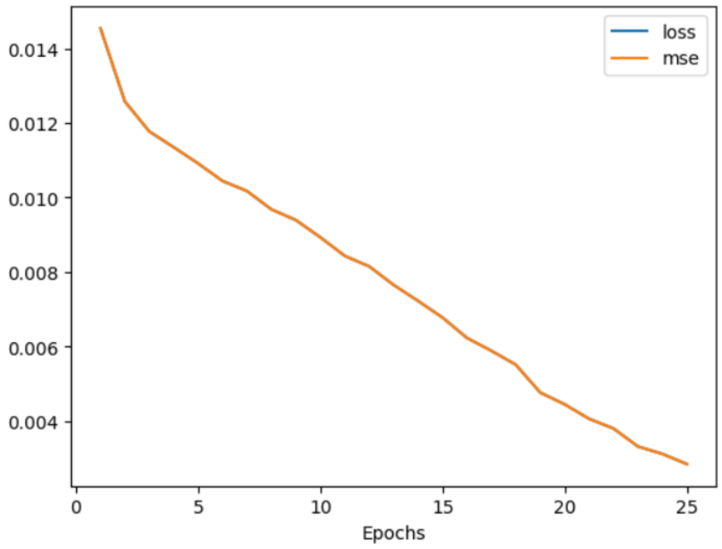


```
20/20 [==============================] - 2s 46ms/step - loss: 0.0321 - out_reg1_loss: 0.0137 - out_reg2_loss: 0.0184 - out_reg1_mse: 0.0137 - out_reg2_mse: 0.0184
Test loss: 0.03209678456187248
Test MSE: 0.013659887947142124
```

B.

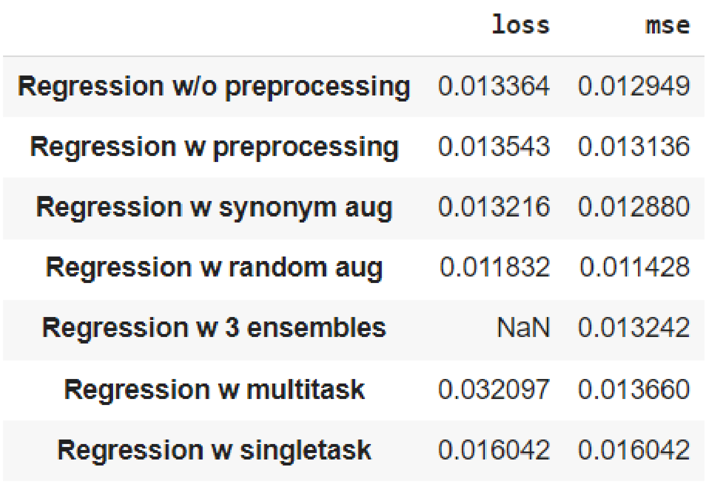
```
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_token (InputLayer)       [(None, 128)]        0           []

 masked_token (InputLayer)      [(None, 128)]        0           []

 tf_distil_bert_model_1 (TFDist  TFBaseModelOutput(l  66362880   ['input_token[0][0]',
 ilBertModel)                   ast_hidden_state=(N               'masked_token[0][0]']
                                one, 128, 768),
                                 hidden_states=None
                                , attentions=None)

 global_average_pooling1d_maske  (None, 768)         0           ['tf_distil_bert_model_1[0][0]']
 d_1 (GlobalAveragePooling1DMas
 ked)

 dense_2 (Dense)                (None, 16)           12304       ['global_average_pooling1d_masked
                                                                 _1[0][0]']

 out_reg1 (Dense)               (None, 1)            17          ['dense_2[0][0]']

==================================================================================================
Total params: 66,375,201
Trainable params: 66,375,201
Non-trainable params: 0
__________________________________________________________________________________________________
```

```
Epoch 1/25
5/5 [==============================] - 91s 8s/step - loss: 0.0145 - mse: 0.0145
Epoch 2/25
5/5 [==============================] - 1s 183ms/step - loss: 0.0126 - mse: 0.0126
Epoch 3/25
5/5 [==============================] - 1s 180ms/step - loss: 0.0118 - mse: 0.0118
Epoch 4/25
5/5 [==============================] - 1s 183ms/step - loss: 0.0114 - mse: 0.0114
Epoch 5/25
5/5 [==============================] - 1s 186ms/step - loss: 0.0109 - mse: 0.0109
Epoch 6/25
5/5 [==============================] - 1s 189ms/step - loss: 0.0104 - mse: 0.0104
Epoch 7/25
5/5 [==============================] - 1s 184ms/step - loss: 0.0102 - mse: 0.0102
Epoch 8/25
5/5 [==============================] - 1s 186ms/step - loss: 0.0097 - mse: 0.0097
Epoch 9/25
5/5 [==============================] - 1s 181ms/step - loss: 0.0094 - mse: 0.0094
Epoch 10/25
5/5 [==============================] - 1s 180ms/step - loss: 0.0089 - mse: 0.0089
Epoch 11/25
5/5 [==============================] - 1s 185ms/step - loss: 0.0084 - mse: 0.0084
Epoch 12/25
5/5 [==============================] - 1s 180ms/step - loss: 0.0081 - mse: 0.0081
Epoch 13/25
5/5 [==============================] - 1s 180ms/step - loss: 0.0077 - mse: 0.0077
Epoch 14/25
5/5 [==============================] - 1s 180ms/step - loss: 0.0072 - mse: 0.0072
Epoch 15/25
5/5 [==============================] - 1s 180ms/step - loss: 0.0068 - mse: 0.0068
Epoch 16/25
5/5 [==============================] - 1s 179ms/step - loss: 0.0062 - mse: 0.0062
Epoch 17/25
5/5 [==============================] - 1s 188ms/step - loss: 0.0059 - mse: 0.0059
Epoch 18/25
5/5 [==============================] - 1s 181ms/step - loss: 0.0055 - mse: 0.0055
Epoch 19/25
5/5 [==============================] - 1s 181ms/step - loss: 0.0048 - mse: 0.0048
Epoch 20/25
5/5 [==============================] - 1s 184ms/step - loss: 0.0044 - mse: 0.0044
Epoch 21/25
5/5 [==============================] - 1s 183ms/step - loss: 0.0041 - mse: 0.0041
Epoch 22/25
5/5 [==============================] - 1s 191ms/step - loss: 0.0038 - mse: 0.0038
Epoch 23/25
5/5 [==============================] - 1s 184ms/step - loss: 0.0033 - mse: 0.0033
Epoch 24/25
5/5 [==============================] - 1s 180ms/step - loss: 0.0031 - mse: 0.0031
Epoch 25/25
5/5 [==============================] - 1s 180ms/step - loss: 0.0028 - mse: 0.0028
```



```
20/20 [==============================] - 13s 148ms/step - loss: 0.0160 - mse: 0.0160
Test loss: 0.01604236476123333
Test MSE: 0.01604236476123333
```

The Multitask regressor firstly has 16 more trainable parameters, due to having 2 outputs. It also has a larger LOSS. This is due to the fact that the loss is the sum of the losses due to each output, and because it has 2 outputs, it performs worse in terms of loss compared to the single-task regressor. As such, this metric is non-comparable. So instead I will look at MSE. The multitask mse for humour detection is a lot lower than the single-task regressor trained with the same amount of data. This is likely due to the fact that the multi-task regressor has to learn generally which features are best for minimizing the loss, which in a multi-task regressor is the loss of at least 1 other output. And if the tasks are related, as they are in this case, the model should generalize much better. So, The multi-task regressor is much better at generalizing than the single-task regressor. This is evident by looking at the training mse. For the single-task regressor, the final mse is 0.0028 on the training set and for the multi-task regressor, the final mse (look at out_reg1_mse) is 0.0068 on the training set. But looking at the test-set performance, we see that the single-task regressor actually has a larger mse at 0.016042 than the multi-task regressor at 0.013660. This provides evidence to suggest that multi-task models are able to generalize better than their single-task counterparts. The multi-task model is slightly worse than the first regressor (From task 1), even though the multi-task model was trained on half the data. The first regressor was also trained for only 10 epochs so it is difficult to provide a conclusion regarding multi-task learning.

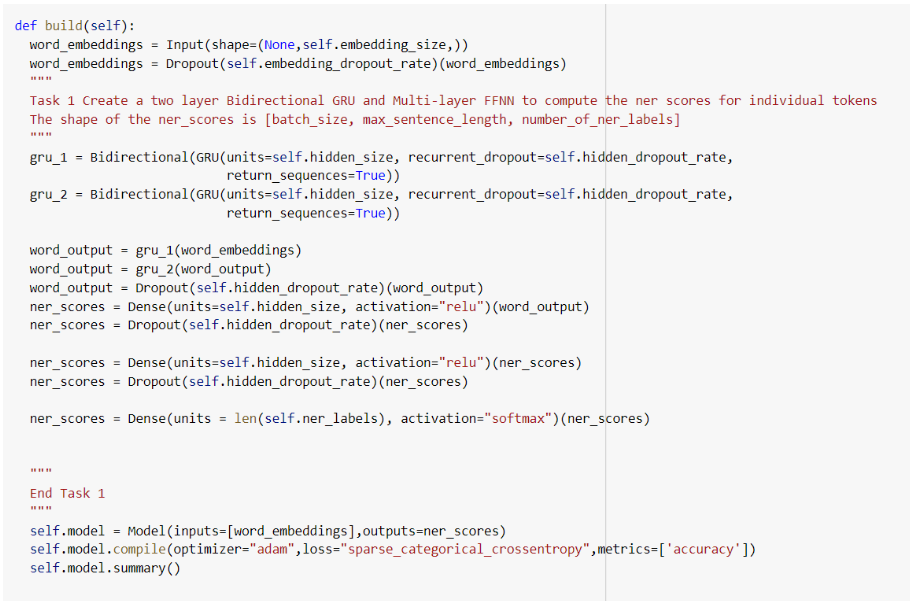


	loss	mse
Regression w/o preprocessing	0.013364	0.012949
Regression w preprocessing	0.013543	0.013136
Regression w synonym aug	0.013216	0.012880
Regression w random aug	0.011832	0.011428
Regression w 3 ensembles	NaN	0.013242
Regression w multitask	0.032097	0.013660
Regression w singletask	0.016042	0.016042

We see that the Random Aug model has the best performance w.r.t. The loss and the mse.

Part B: Information Extraction 1: Training a Named Entity Resolver

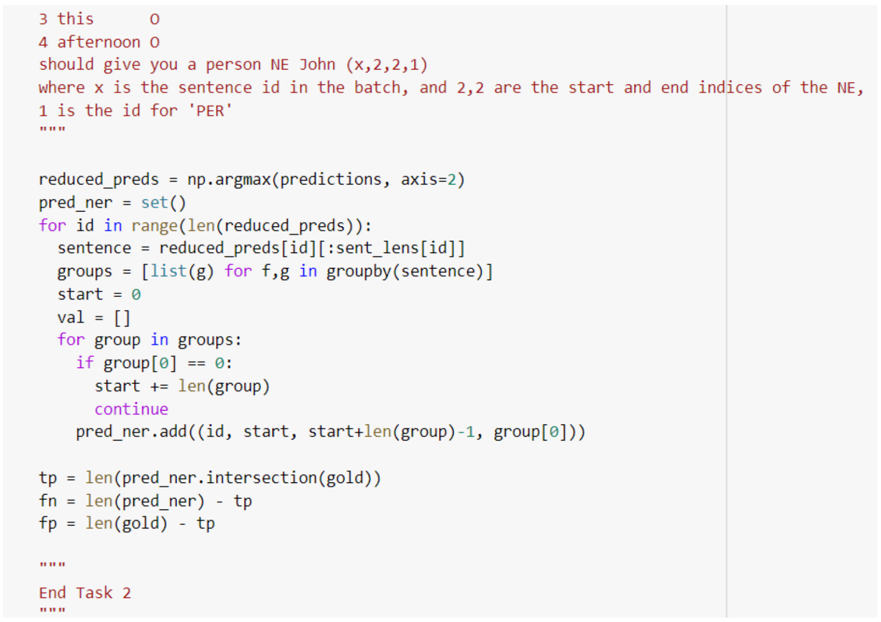
1.



```
def build(self):
  word_embeddings = Input(shape=(None,self.embedding_size,))
  word_embeddings = Dropout(self.embedding_dropout_rate)(word_embeddings)
  """
  Task 1 Create a two layer Bidirectional GRU and Multi-layer FFNN to compute the ner scores for individual tokens
  The shape of the ner_scores is [batch_size, max_sentence_length, number_of_ner_labels]
  """
  gru_1 = Bidirectional(GRU(units=self.hidden_size, recurrent_dropout=self.hidden_dropout_rate,
                            return_sequences=True))
  gru_2 = Bidirectional(GRU(units=self.hidden_size, recurrent_dropout=self.hidden_dropout_rate,
                            return_sequences=True))

  word_output = gru_1(word_embeddings)
  word_output = gru_2(word_output)
  word_output = Dropout(self.hidden_dropout_rate)(word_output)
  ner_scores = Dense(units=self.hidden_size, activation="relu")(word_output)
  ner_scores = Dropout(self.hidden_dropout_rate)(ner_scores)

  ner_scores = Dense(units=self.hidden_size, activation="relu")(ner_scores)
  ner_scores = Dropout(self.hidden_dropout_rate)(ner_scores)

  ner_scores = Dense(units = len(self.ner_labels), activation="softmax")(ner_scores)


  """
  End Task 1
  """
  self.model = Model(inputs=[word_embeddings],outputs=ner_scores)
  self.model.compile(optimizer="adam",loss="sparse_categorical_crossentropy",metrics=['accuracy'])
  self.model.summary()
```

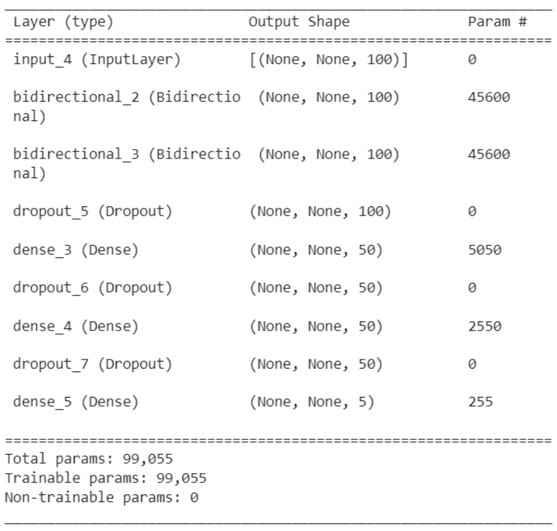
Gru_1 and Gru_2 are the bidirectional GRUs and there are 2 hidden dense layers with dropout applied as expected. The final layer is a Dense layer that outputs len(self.ner_labels) values and uses a softmax to turn the results into a probability distribution of the predicted ner label.

2.

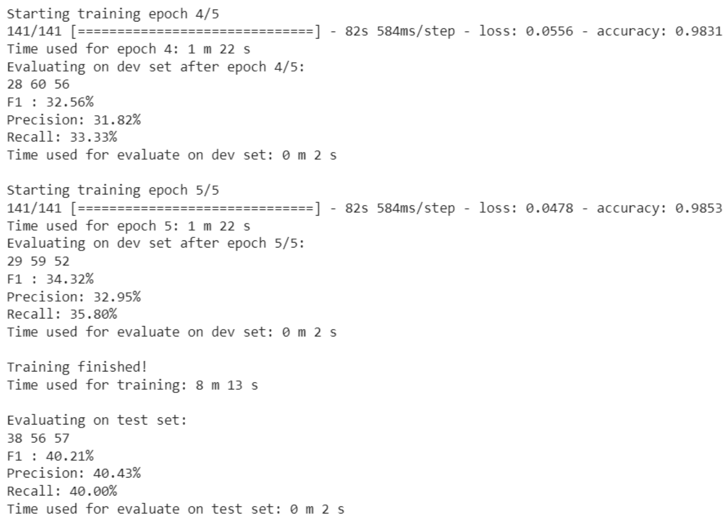


```
  3 this      O
  4 afternoon O
  should give you a person NE John (x,2,2,1)
  where x is the sentence id in the batch, and 2,2 are the start and end indices of the NE,
  1 is the id for 'PER'
  """

  reduced_preds = np.argmax(predictions, axis=2)
  pred_ner = set()
  for id in range(len(reduced_preds)):
    sentence = reduced_preds[id][:sent_lens[id]]
    groups = [list(g) for f,g in groupby(sentence)]
    start = 0
    val = []
    for group in groups:
      if group[0] == 0:
        start += len(group)
        continue
      pred_ner.add((id, start, start+len(group)-1, group[0]))

  tp = len(pred_ner.intersection(gold))
  fn = len(pred_ner) - tp
  fp = len(gold) - tp

  """
  End Task 2
  """
```

Reduced_preds takes the label with the largest probability for each word. Then, the predictions are added to a set in form (sent_id, start, end, ner_id). These are the predictions. Then, the true positives, tp, is simply the intersection of the predictions and the gold labels. False negative, fn, is the number of predicted ners minus the number of true positives. False positive, fp, is the number of golds minus the number of true positives.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_4 (InputLayer)        [(None, None, 100)]       0

 bidirectional_2 (Bidirectio  (None, None, 100)        45600
 nal)

 bidirectional_3 (Bidirectio  (None, None, 100)        45600
 nal)

 dropout_5 (Dropout)         (None, None, 100)         0

 dense_3 (Dense)             (None, None, 50)          5050

 dropout_6 (Dropout)         (None, None, 50)          0

 dense_4 (Dense)             (None, None, 50)          2550

 dropout_7 (Dropout)         (None, None, 50)          0

 dense_5 (Dense)             (None, None, 5)           255

=================================================================
Total params: 99,055
Trainable params: 99,055
Non-trainable params: 0
_________________________________________________________________
```

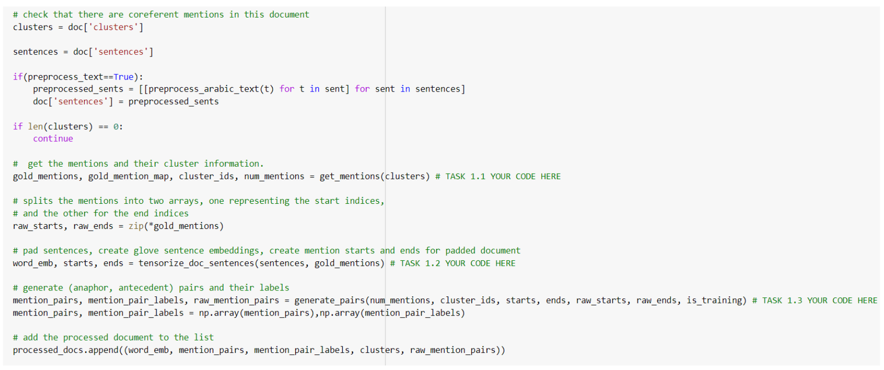


```
Starting training epoch 4/5
141/141 [==============================] - 82s 584ms/step - loss: 0.0556 - accuracy: 0.9831
Time used for epoch 4: 1 m 22 s
Evaluating on dev set after epoch 4/5:
28 60 56
F1 : 32.56%
Precision: 31.82%
Recall: 33.33%
Time used for evaluate on dev set: 0 m 2 s

Starting training epoch 5/5
141/141 [==============================] - 82s 584ms/step - loss: 0.0478 - accuracy: 0.9853
Time used for epoch 5: 1 m 22 s
Evaluating on dev set after epoch 5/5:
29 59 52
F1 : 34.32%
Precision: 32.95%
Recall: 35.80%
Time used for evaluate on dev set: 0 m 2 s

Training finished!
Time used for training: 8 m 13 s

Evaluating on test set:
38 56 57
F1 : 40.21%
Precision: 40.43%
Recall: 40.00%
Time used for evaluate on test set: 0 m 2 s
```

The final f1 score on the test set is 40.21%.

Part C: Information Extraction 2: A Coreference Resolver

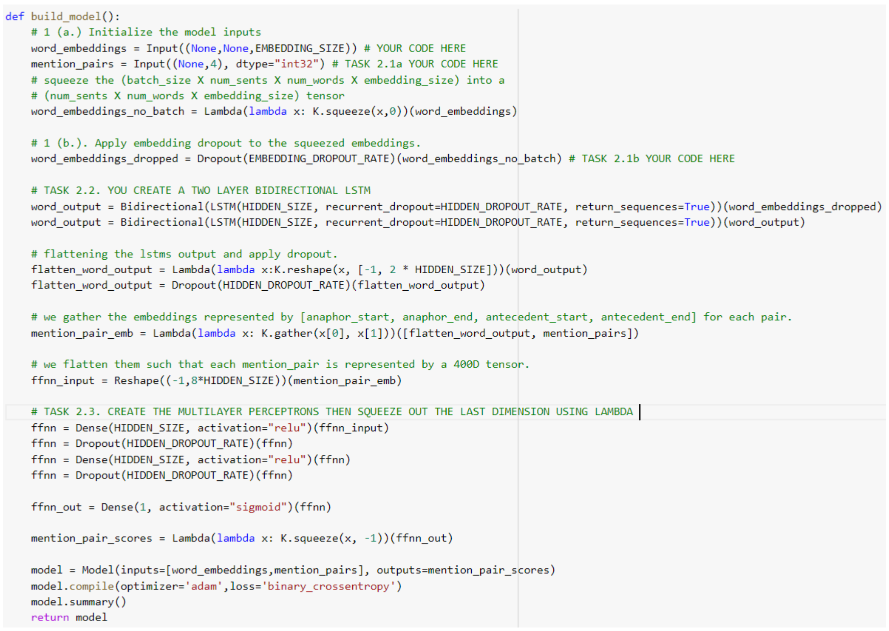
1.



```python
# check that there are coreferent mentions in this document
clusters = doc['clusters']

sentences = doc['sentences']

if(preprocess_text==True):
    preprocessed_sents = [[preprocess_arabic_text(t) for t in sent] for sent in sentences]
    doc['sentences'] = preprocessed_sents

if len(clusters) == 0:
    continue

#  get the mentions and their cluster information.
gold_mentions, gold_mention_map, cluster_ids, num_mentions = get_mentions(clusters) # TASK 1.1 YOUR CODE HERE

# splits the mentions into two arrays, one representing the start indices,
# and the other for the end indices
raw_starts, raw_ends = zip(*gold_mentions)

# pad sentences, create glove sentence embeddings, create mention starts and ends for padded document
word_emb, starts, ends = tensorize_doc_sentences(sentences, gold_mentions) # TASK 1.2 YOUR CODE HERE

# generate (anaphor, antecedent) pairs and their labels
mention_pairs, mention_pair_labels, raw_mention_pairs = generate_pairs(num_mentions, cluster_ids, starts, ends, raw_starts, raw_ends, is_training) # TASK 1.3 YOUR CODE HERE
mention_pairs, mention_pair_labels = np.array(mention_pairs),np.array(mention_pair_labels)

# add the processed document to the list
processed_docs.append((word_emb, mention_pairs, mention_pair_labels, clusters, raw_mention_pairs))
```

1.1 destructures the output of get_mentions(clusters) to the 4 variables as shown above. 1.2 passes sentences and gold_mentions to tensorize_doc_sentences. This creates a padded document embedding (sentences) and it returns embedded sentences with the mention starts and ends are adjusted for the padding. 1.3 generates the (anaphor, antecedent) pairs and their labels, by calling the generate_pairs function with the information from 1.1 and 1.2.

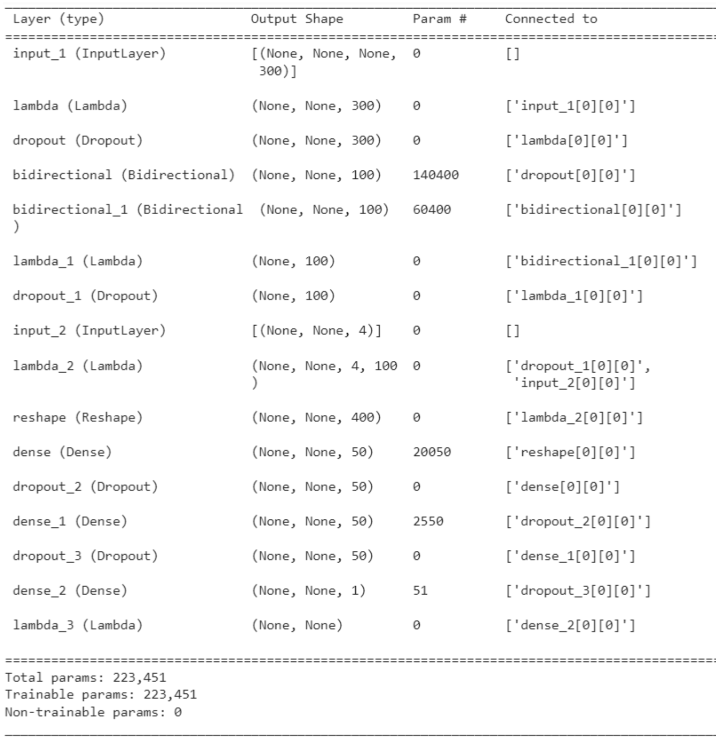
2.



```python
def build_model():
    # 1 (a.) Initialize the model inputs
    word_embeddings = Input((None,None,EMBEDDING_SIZE)) # YOUR CODE HERE
    mention_pairs = Input((None,4), dtype="int32") # TASK 2.1a YOUR CODE HERE
    # squeeze the (batch_size X num_sents X num_words X embedding_size) into a
    # (num_sents X num_words X embedding_size) tensor
    word_embeddings_no_batch = Lambda(lambda x: K.squeeze(x,0))(word_embeddings)

    # 1 (b.). Apply embedding dropout to the squeezed embeddings.
    word_embeddings_dropped = Dropout(EMBEDDING_DROPOUT_RATE)(word_embeddings_no_batch) # TASK 2.1b YOUR CODE HERE

    # TASK 2.2. YOU CREATE A TWO LAYER BIDIRECTIONAL LSTM
    word_output = Bidirectional(LSTM(HIDDEN_SIZE, recurrent_dropout=HIDDEN_DROPOUT_RATE, return_sequences=True))(word_embeddings_dropped)
    word_output = Bidirectional(LSTM(HIDDEN_SIZE, recurrent_dropout=HIDDEN_DROPOUT_RATE, return_sequences=True))(word_output)

    # flattening the lstms output and apply dropout.
    flatten_word_output = Lambda(lambda x:K.reshape(x, [-1, 2 * HIDDEN_SIZE]))(word_output)
    flatten_word_output = Dropout(HIDDEN_DROPOUT_RATE)(flatten_word_output)

    # we gather the embeddings represented by [anaphor_start, anaphor_end, antecedent_start, antecedent_end] for each pair.
    mention_pair_emb = Lambda(lambda x: K.gather(x[0], x[1]))([flatten_word_output, mention_pairs])

    # we flatten them such that each mention_pair is represented by a 400D tensor.
    ffnn_input = Reshape((-1,8*HIDDEN_SIZE))(mention_pair_emb)

    # TASK 2.3. CREATE THE MULTILAYER PERCEPTRONS THEN SQUEEZE OUT THE LAST DIMENSION USING LAMBDA
    ffnn = Dense(HIDDEN_SIZE, activation="relu")(ffnn_input)
    ffnn = Dropout(HIDDEN_DROPOUT_RATE)(ffnn)
    ffnn = Dense(HIDDEN_SIZE, activation="relu")(ffnn)
    ffnn = Dropout(HIDDEN_DROPOUT_RATE)(ffnn)

    ffnn_out = Dense(1, activation="sigmoid")(ffnn)

    mention_pair_scores = Lambda(lambda x: K.squeeze(x, -1))(ffnn_out)

    model = Model(inputs=[word_embeddings,mention_pairs], outputs=mention_pair_scores)
    model.compile(optimizer='adam',loss='binary_crossentropy')
    model.summary()
    return model
```

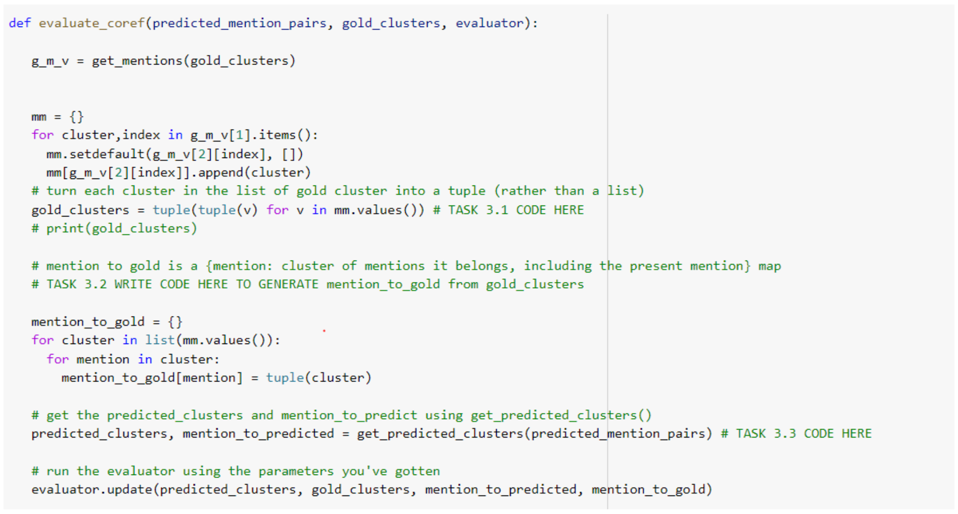
First initialize the model input, (I.e., take as input, the pre-trained embedding, and the mention pairs). Then, apply dropout to the word embedding before passing to a Bidirectional LSTM. Which then passes its outputs to another Bidirectional LSTM (passing the output at each node rather than just the final).

In 2.3., simply create 2 feedforward layers with dropout, of which the output is then passed to a single dense layer with a single output. The output is then squeezed into the desired format for the mention pair scores.



```
_____________________________________________________________________________________________
 Layer (type)                Output Shape         Param #     Connected to
=============================================================================================
 input_1 (InputLayer)        [(None, None, None,  0           []
                              300)]

 lambda (Lambda)             (None, None, 300)    0           ['input_1[0][0]']

 dropout (Dropout)           (None, None, 300)    0           ['lambda[0][0]']

 bidirectional (Bidirectional)  (None, None, 100)  140400     ['dropout[0][0]']

 bidirectional_1 (Bidirectional  (None, None, 100)  60400     ['bidirectional[0][0]']
 )

 lambda_1 (Lambda)           (None, 100)          0           ['bidirectional_1[0][0]']

 dropout_1 (Dropout)         (None, 100)          0           ['lambda_1[0][0]']

 input_2 (InputLayer)        [(None, None, 4)]    0           []

 lambda_2 (Lambda)           (None, None, 4, 100  0           ['dropout_1[0][0]',
                             )                                 'input_2[0][0]']

 reshape (Reshape)           (None, None, 400)    0           ['lambda_2[0][0]']

 dense (Dense)               (None, None, 50)     20050       ['reshape[0][0]']

 dropout_2 (Dropout)         (None, None, 50)     0           ['dense[0][0]']

 dense_1 (Dense)             (None, None, 50)     2550        ['dropout_2[0][0]']

 dropout_3 (Dropout)         (None, None, 50)     0           ['dense_1[0][0]']

 dense_2 (Dense)             (None, None, 1)      51          ['dropout_3[0][0]']

 lambda_3 (Lambda)           (None, None)         0           ['dense_2[0][0]']

=============================================================================================
Total params: 223,451
Trainable params: 223,451
Non-trainable params: 0
_____________________________________________________________________________________________
```

3.



```python
def evaluate_coref(predicted_mention_pairs, gold_clusters, evaluator):

  g_m_v = get_mentions(gold_clusters)


  mm = {}
  for cluster,index in g_m_v[1].items():
    mm.setdefault(g_m_v[2][index], [])
    mm[g_m_v[2][index]].append(cluster)
  # turn each cluster in the list of gold cluster into a tuple (rather than a list)
  gold_clusters = tuple(tuple(v) for v in mm.values()) # TASK 3.1 CODE HERE
  # print(gold_clusters)

  # mention to gold is a {mention: cluster of mentions it belongs, including the present mention} map
  # TASK 3.2 WRITE CODE HERE TO GENERATE mention_to_gold from gold_clusters

  mention_to_gold = {}
  for cluster in list(mm.values()):
    for mention in cluster:
      mention_to_gold[mention] = tuple(cluster)

  # get the predicted_clusters and mention_to_predict using get_predicted_clusters()
  predicted_clusters, mention_to_predicted = get_predicted_clusters(predicted_mention_pairs) # TASK 3.3 CODE HERE

  # run the evaluator using the parameters you've gotten
  evaluator.update(predicted_clusters, gold_clusters, mention_to_predicted, mention_to_gold)
```

First create g_m_v which gets the mentions from the gold_clusters.

g_m_v[0] is the gold mentions (list of mentions)
g_m_v[1] is the gold mention map (dict of form {mention:id})
g_m_v[2] is the cluster ids (list where the id'th index contains the cluster id for mention with the id, id.
g_m_v[3] is just the number of mentions.

Then, loop over g_m_v[1].items() ((mention, id) of gold mention map).
Then, for each cluster id (g_m_v[2][index]), set the default value to an empty list if it hasn't been set yet.

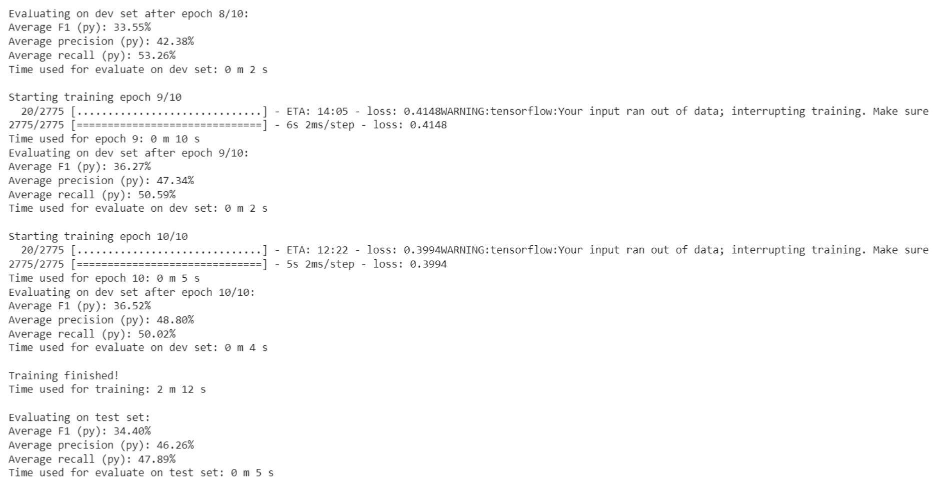
Then, append each cluster to that particular cluster id. So the final dict is of the form {cluster_id : cluster_of_mentions}.

So, gold_clusters is simply a tuple of the values of mm.

3.2. Mention_to_gold is a dict of the form, {mention: cluster_where_that_mention_belongs}. This is done for every mention.

3.3. Simply get the predicted clusters and mention_to_predicted by using the get_predicted_clusters function.

Then pass these to the evaluator to get an evaluation.

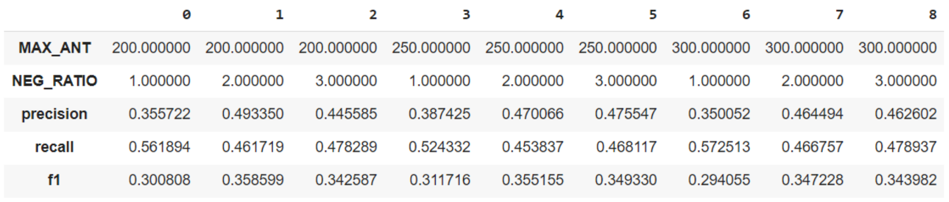


```
Evaluating on dev set after epoch 8/10:
Average F1 (py): 33.55%
Average precision (py): 42.38%
Average recall (py): 53.26%
Time used for evaluate on dev set: 0 m 2 s

Starting training epoch 9/10
  20/2775 [..............................] - ETA: 14:05 - loss: 0.4148WARNING:tensorflow:Your input ran out of data; interrupting training. Make sure
2775/2775 [==============================] - 6s 2ms/step - loss: 0.4148
Time used for epoch 9: 0 m 10 s
Evaluating on dev set after epoch 9/10:
Average F1 (py): 36.27%
Average precision (py): 47.34%
Average recall (py): 50.59%
Time used for evaluate on dev set: 0 m 2 s

Starting training epoch 10/10
  20/2775 [..............................] - ETA: 12:22 - loss: 0.3994WARNING:tensorflow:Your input ran out of data; interrupting training. Make sure
2775/2775 [==============================] - 5s 2ms/step - loss: 0.3994
Time used for epoch 10: 0 m 5 s
Evaluating on dev set after epoch 10/10:
Average F1 (py): 36.52%
Average precision (py): 48.80%
Average recall (py): 50.02%
Time used for evaluate on dev set: 0 m 4 s

Training finished!
Time used for training: 2 m 12 s

Evaluating on test set:
Average F1 (py): 34.40%
Average precision (py): 46.26%
Average recall (py): 47.89%
Time used for evaluate on test set: 0 m 5 s
```

4.

1. Would the performance decrease if we do not preprocess the text? If yes (or no), then why?
I think the performance would decrease if we do not preprocess the text. I believe this to be because of a reduction in vocabulary size and thus reducing noise. For example, setting all inputs to lowercase means that we can use an embedding without considering uppercase, which reduces the vocabulary size greatly, and as such less data is needed.

2. Experiment with different values for max antecedent (MAX_ANT) and negative ratio (NEG_RATIO), what do you observe?



	0	1	2	3	4	5	6	7	8
MAX_ANT	200.000000	200.000000	200.000000	250.000000	250.000000	250.000000	300.000000	300.000000	300.000000
NEG_RATIO	1.000000	2.000000	3.000000	1.000000	2.000000	3.000000	1.000000	2.000000	3.000000
precision	0.355722	0.493350	0.445585	0.387425	0.470066	0.475547	0.350052	0.464494	0.462602
recall	0.561894	0.461719	0.478289	0.524332	0.453837	0.468117	0.572513	0.466757	0.478937
f1	0.300808	0.358599	0.342587	0.311716	0.355155	0.349330	0.294055	0.347228	0.343982

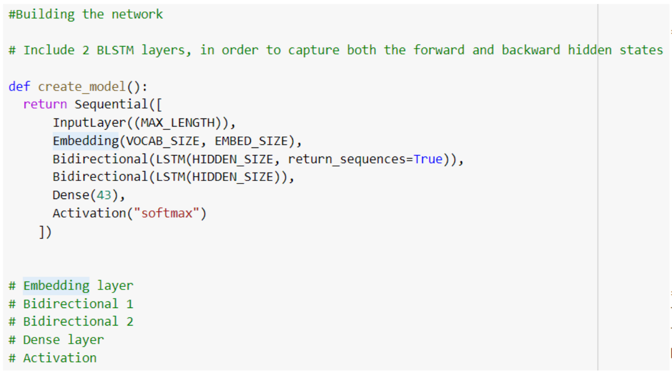
From my experimentation (results in the table above), I saw that a MAX_ANT of 200 and a NEG_RATIO of 2 had the best results with an average F1 score that was just slightly above the default of MAX_ANT, 250 and NEG_RATIO, 2. Setting MAX_ANT as a constant and looking at changing the NEG_RATIO, there is a clearly observable pattern. The highest to lowest average F1 is always in the order of NEG_RATIO: 2, 3, 1. Now, keeping NEG_RATIO constant and looking at changing MAX_ANT, gets a bit more interesting as there is no clear pattern as to which one is better. NEG_RATIO: 1, MAX_ANT order is: 250, 200, 300. NEG_RATIO: 2, MAX_ANT order is: 200, 250, 300. NEG_RATIO: 3, MAX_ANT order is: 250, 300, 200. Overall, 250 ranks the best overall. But, NEG_RATIO: 2, MAX_ANT: 200, has the best F1. So, It seems that having a NEG_RATIO of around 2 is the best and having a MAX_ANT of around 200 is also the best.

3. How would you improve the accuracy?
One possibility could be to use more complex and larger Embeddings. For example, using the BERT embeddings which has a token embedding as well as a segment embedding. This provides much more information for the resolver to work with. Could try different preprocessing methods to reduce the noise in the data even further.

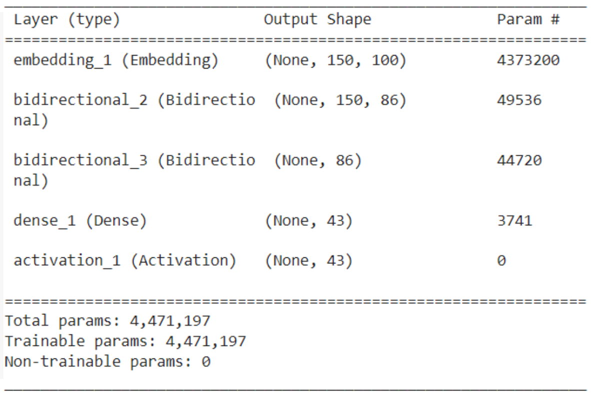
Part D: Dialogue 1: Dialogue Act Tagging

Please Note that for the evaluation of individual classes, I will look at accuracy, precision, and recall in terms of a binary classifier (I.e., One class is that class and the other class is all other classes). Due to the nature of this, accuracy isn't the best metric as for minority classes, this metric is always higher, so I will look more into precision and recall. I will compare accuracies across models but only for the same class.

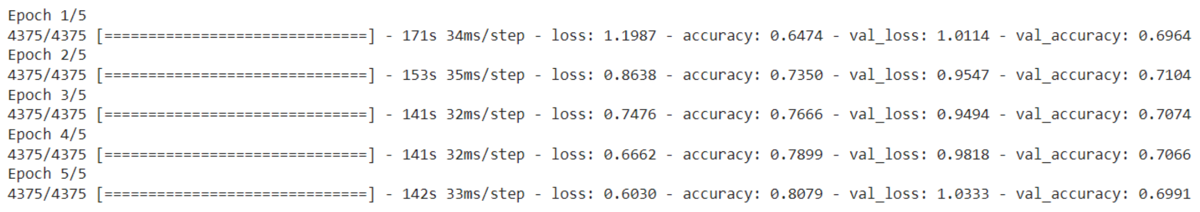
1.



```
#Building the network

# Include 2 BLSTM layers, in order to capture both the forward and backward hidden states

def create_model():
  return Sequential([
      InputLayer((MAX_LENGTH)),
      Embedding(VOCAB_SIZE, EMBED_SIZE),
      Bidirectional(LSTM(HIDDEN_SIZE, return_sequences=True)),
      Bidirectional(LSTM(HIDDEN_SIZE)),
      Dense(43),
      Activation("softmax")
    ])


# Embedding layer
# Bidirectional 1
# Bidirectional 2
# Dense layer
# Activation
```



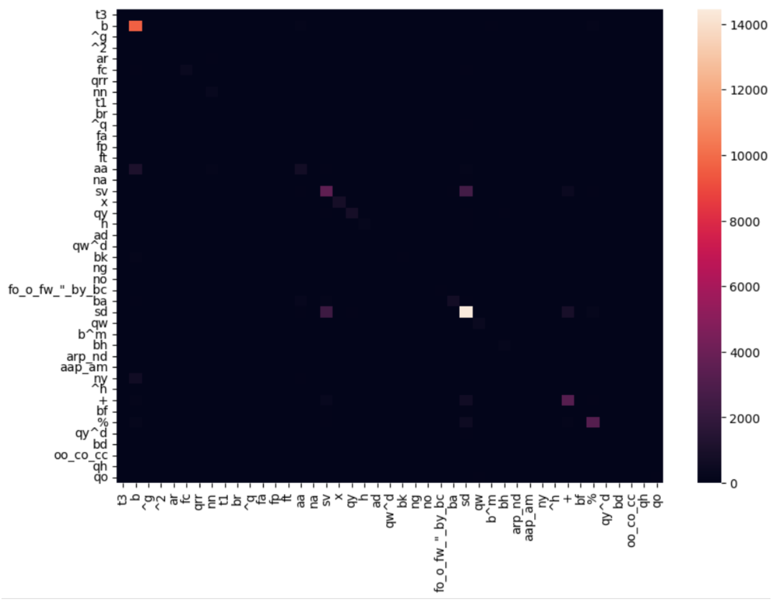
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 150, 100)          4373200

bidirectional_2 (Bidirectio  (None, 150, 86)           49536
nal)

bidirectional_3 (Bidirectio  (None, 86)                44720
nal)

dense_1 (Dense)              (None, 43)                3741

activation_1 (Activation)    (None, 43)                0

=================================================================
Total params: 4,471,197
Trainable params: 4,471,197
Non-trainable params: 0
_________________________________________________________________
```



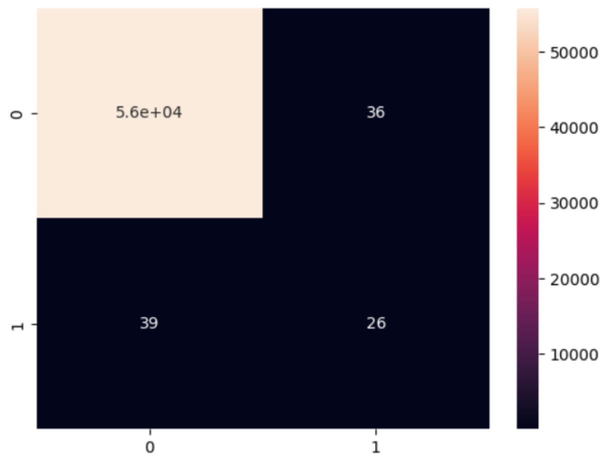
```
Epoch 1/5
4375/4375 [==============================] - 171s 34ms/step - loss: 1.1987 - accuracy: 0.6474 - val_loss: 1.0114 - val_accuracy: 0.6964
Epoch 2/5
4375/4375 [==============================] - 153s 35ms/step - loss: 0.8638 - accuracy: 0.7350 - val_loss: 0.9547 - val_accuracy: 0.7104
Epoch 3/5
4375/4375 [==============================] - 141s 32ms/step - loss: 0.7476 - accuracy: 0.7666 - val_loss: 0.9494 - val_accuracy: 0.7074
Epoch 4/5
4375/4375 [==============================] - 141s 32ms/step - loss: 0.6662 - accuracy: 0.7899 - val_loss: 0.9818 - val_accuracy: 0.7066
Epoch 5/5
4375/4375 [==============================] - 142s 33ms/step - loss: 0.6030 - accuracy: 0.8079 - val_loss: 1.0333 - val_accuracy: 0.6991
```

```
560/560 [==============================] - 16s 26ms/step - loss: 1.0378 - accuracy: 0.6989
```

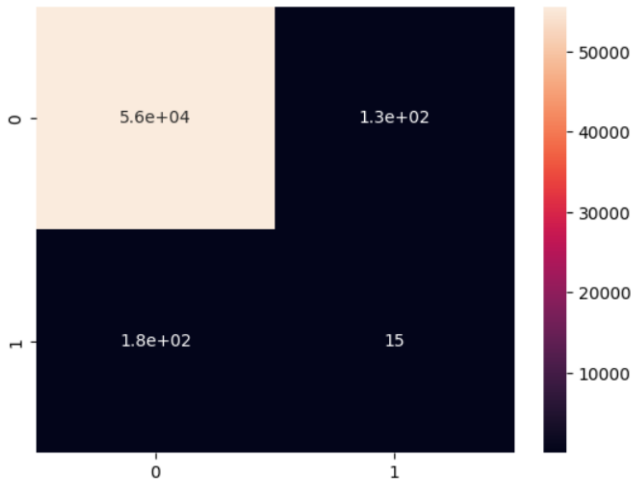
```
Overall Accuracy: 69.89374160766602
```



```
br_conf = confusion_matrix_calculator("br", test_predicted_classes)
```



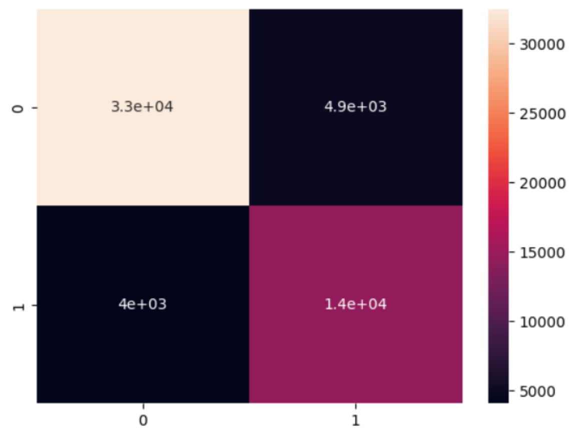
```
bf_conf = confusion_matrix_calculator("bf", test_predicted_classes)
```



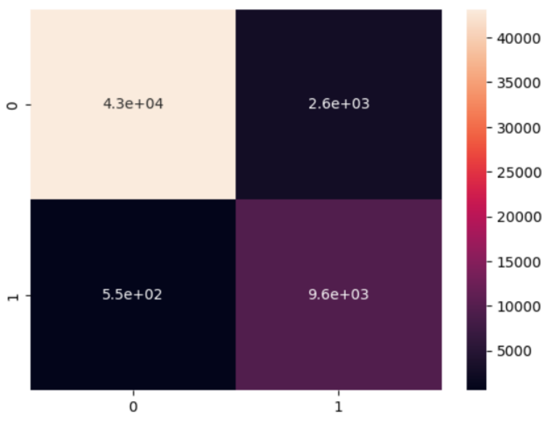
```
br accuracy: 99.87%
br precision: 41.94%
br recall: 40.00%

bf accuracy: 99.44%
bf precision: 10.20%
bf recall: 7.65%
```

Evidently, the Tagger doesn't perform too well on minority classes like the above. Let us now look at the majority classes:

```
sd accuracy: 84.02%
sd precision: 74.68%
sd recall: 78.16%
```



```
b accuracy: 94.42%
b precision: 78.98%
b recall: 94.59%
```



We see that evidently, the model performs much better on the more prevalent classes and struggles with the minority classes like "br" and "bf". These classes have a low precision and low recall compared to the more dominant classes like "sd" and "b".

2.

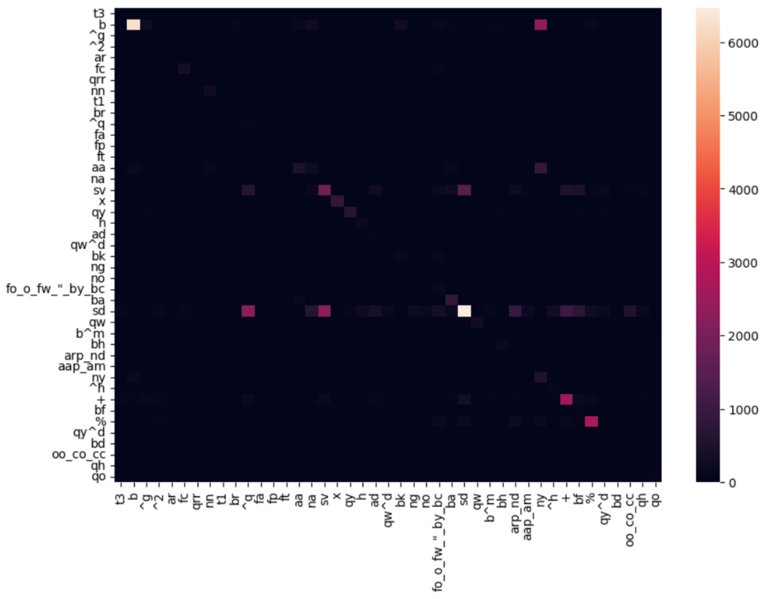
The more balanced model:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 embedding_2 (Embedding)     (None, 150, 100)          4373200   
                                                                 
 bidirectional_4 (Bidirectio  (None, 150, 86)          49536     
 nal)                                                            
                                                                 
 bidirectional_5 (Bidirectio  (None, 86)               44720     
 nal)                                                            
                                                                 
 dense_2 (Dense)             (None, 43)                3741      
                                                                 
 activation_2 (Activation)   (None, 43)                0         
                                                                 
=================================================================
Total params: 4,471,197
Trainable params: 4,471,197
Non-trainable params: 0
_________________________________________________________________
```

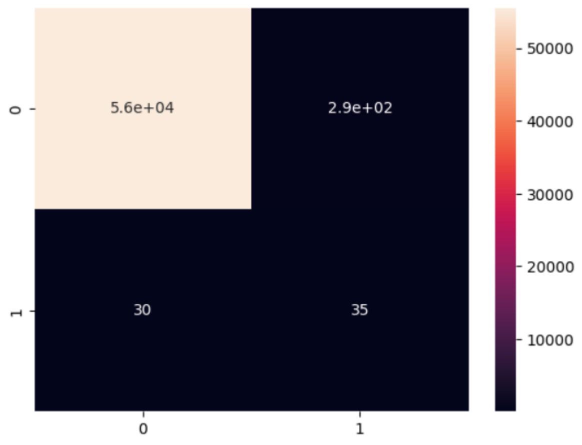
```
Epoch 1/5
4375/4375 [==============================] - 175s 37ms/step - loss: 2.7508 - accuracy: 0.3202 - val_loss: 2.2900 - val_accuracy: 0.3583
Epoch 2/5
4375/4375 [==============================] - 139s 32ms/step - loss: 1.9034 - accuracy: 0.4087 - val_loss: 1.9305 - val_accuracy: 0.4542
Epoch 3/5
4375/4375 [==============================] - 139s 32ms/step - loss: 1.5299 - accuracy: 0.4531 - val_loss: 1.9603 - val_accuracy: 0.4093
Epoch 4/5
4375/4375 [==============================] - 141s 32ms/step - loss: 1.2923 - accuracy: 0.4870 - val_loss: 1.8724 - val_accuracy: 0.4417
Epoch 5/5
4375/4375 [==============================] - 141s 32ms/step - loss: 1.0999 - accuracy: 0.5288 - val_loss: 1.8250 - val_accuracy: 0.4671
```

```
560/560 [==============================] - 17s 28ms/step - loss: 1.8920 - accuracy: 0.4542
```

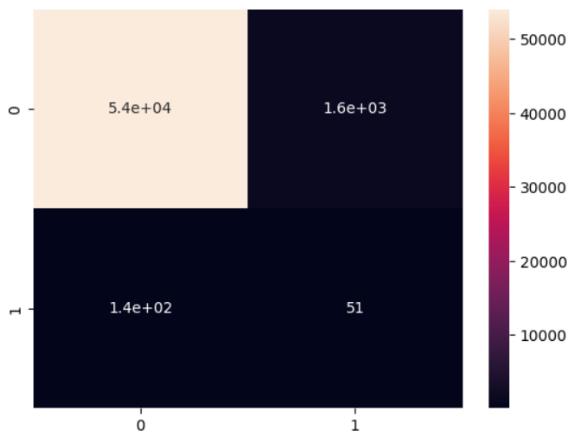
```
Overall Accuracy: 45.42234539985657
```



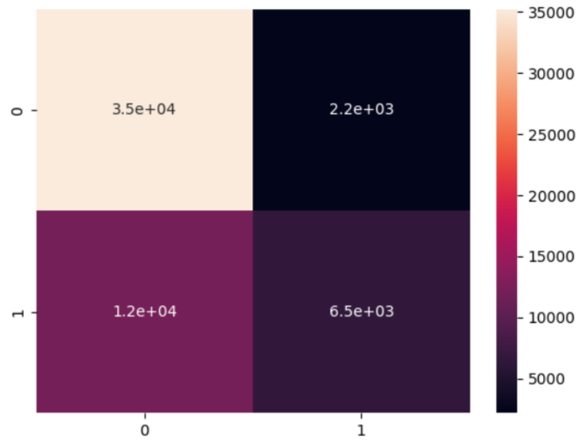
```
br accuracy: 99.42%
br precision: 10.70%
br recall: 53.85%
```



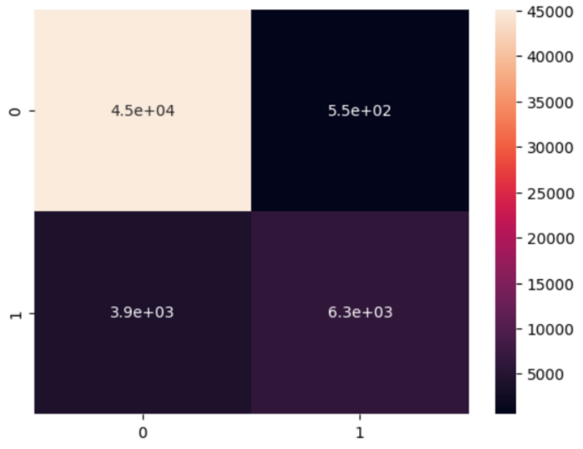
```
bf accuracy: 96.80%
bf precision: 3.01%
bf recall: 26.02%
```



```
sd accuracy: 74.61%
sd precision: 74.84%
sd recall: 34.96%
```



```
b accuracy: 92.00%
b precision: 91.91%
b recall: 61.56%
```

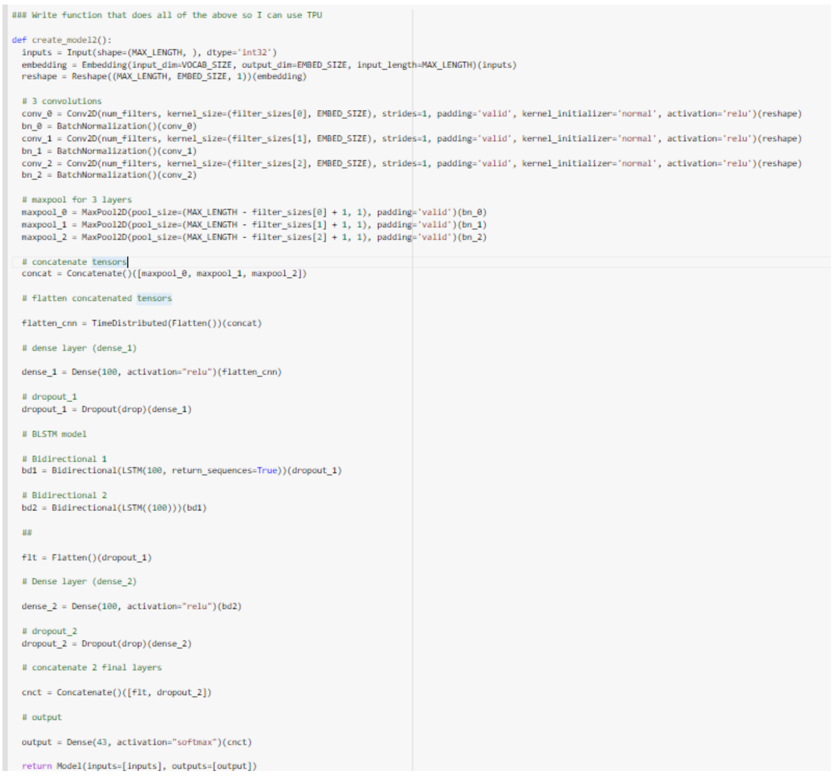


Comparing to model 1:
“Br” - precision goes down quite a lot but the recall increases by 10%. Accuracy goes down a bit.
“Bf” - Again, precision goes down but the recall increases. Accuracy goes down by about 2.5%.
“Sd” - Precision stays around the same but recall halves. Accuracy goes down by 10%.
“B” - Precision goes up by about 13% but recall goes down by about 1-third. Accuracy goes down by about 2%.

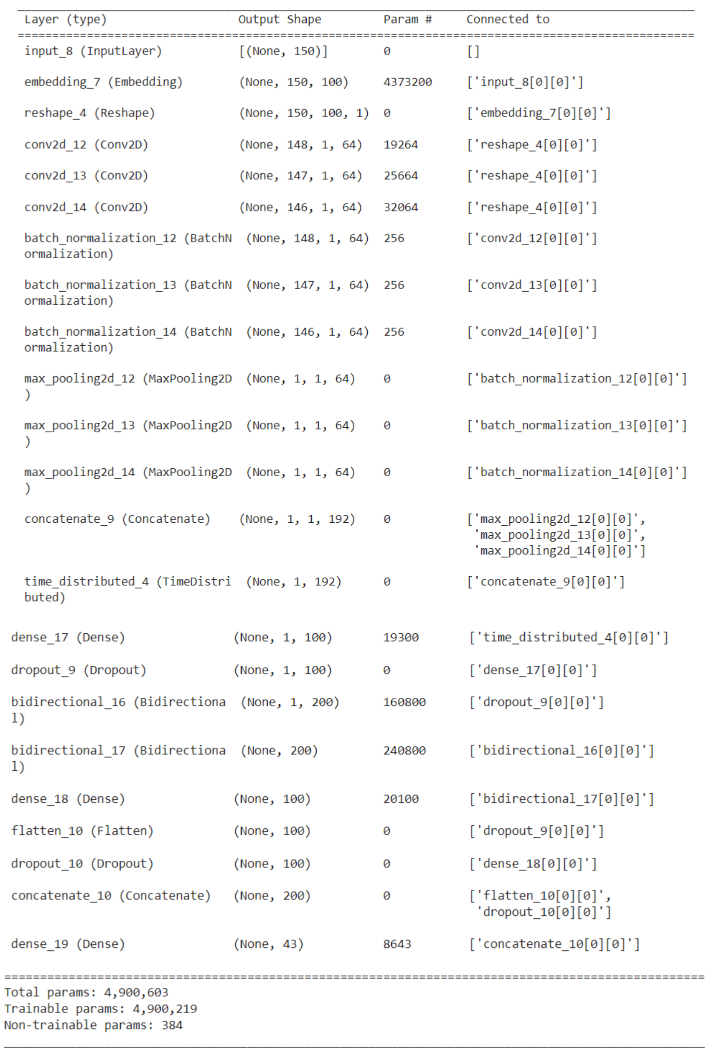
Overall, this model increases the recall of the less frequent classes and as such decreases the recall of the more frequent classes.

1.

Context Model.



```
### Write function that does all of the above so I can use TPU

def create_model2():
  inputs = Input(shape=(MAX_LENGTH, ), dtype='int32')
  embedding = Embedding(input_dim=VOCAB_SIZE, output_dim=EMBED_SIZE, input_length=MAX_LENGTH)(inputs)
  reshape = Reshape((MAX_LENGTH, EMBED_SIZE, 1))(embedding)

  # 3 convolutions
  conv_0 = Conv2D(num_filters, kernel_size=(filter_sizes[0], EMBED_SIZE), strides=1, padding='valid', kernel_initializer='normal', activation='relu')(reshape)
  bn_0 = BatchNormalization()(conv_0)
  conv_1 = Conv2D(num_filters, kernel_size=(filter_sizes[1], EMBED_SIZE), strides=1, padding='valid', kernel_initializer='normal', activation='relu')(reshape)
  bn_1 = BatchNormalization()(conv_1)
  conv_2 = Conv2D(num_filters, kernel_size=(filter_sizes[2], EMBED_SIZE), strides=1, padding='valid', kernel_initializer='normal', activation='relu')(reshape)
  bn_2 = BatchNormalization()(conv_2)

  # maxpool for 3 layers
  maxpool_0 = MaxPool2D(pool_size=(MAX_LENGTH - filter_sizes[0] + 1, 1), padding='valid')(bn_0)
  maxpool_1 = MaxPool2D(pool_size=(MAX_LENGTH - filter_sizes[1] + 1, 1), padding='valid')(bn_1)
  maxpool_2 = MaxPool2D(pool_size=(MAX_LENGTH - filter_sizes[2] + 1, 1), padding='valid')(bn_2)

  # concatenate tensors
  concat = Concatenate()([maxpool_0, maxpool_1, maxpool_2])

  # flatten concatenated tensors

  flatten_cnn = TimeDistributed(Flatten())(concat)

  # dense layer (dense_1)

  dense_1 = Dense(100, activation="relu")(flatten_cnn)

  # dropout_1
  dropout_1 = Dropout(drop)(dense_1)

  # BLSTM model

  # Bidirectional 1
  bd1 = Bidirectional(LSTM(100, return_sequences=True))(dropout_1)

  # Bidirectional 2
  bd2 = Bidirectional(LSTM((100)))(bd1)

  ##

  flt = Flatten()(dropout_1)

  # Dense layer (dense_2)

  dense_2 = Dense(100, activation="relu")(bd2)

  # dropout_2
  dropout_2 = Dropout(drop)(dense_2)

  # concatenate 2 final layers

  cnct = Concatenate()([flt, dropout_2])

  # output

  output = Dense(43, activation="softmax")(cnct)

  return Model(inputs=[inputs], outputs=[output])
```

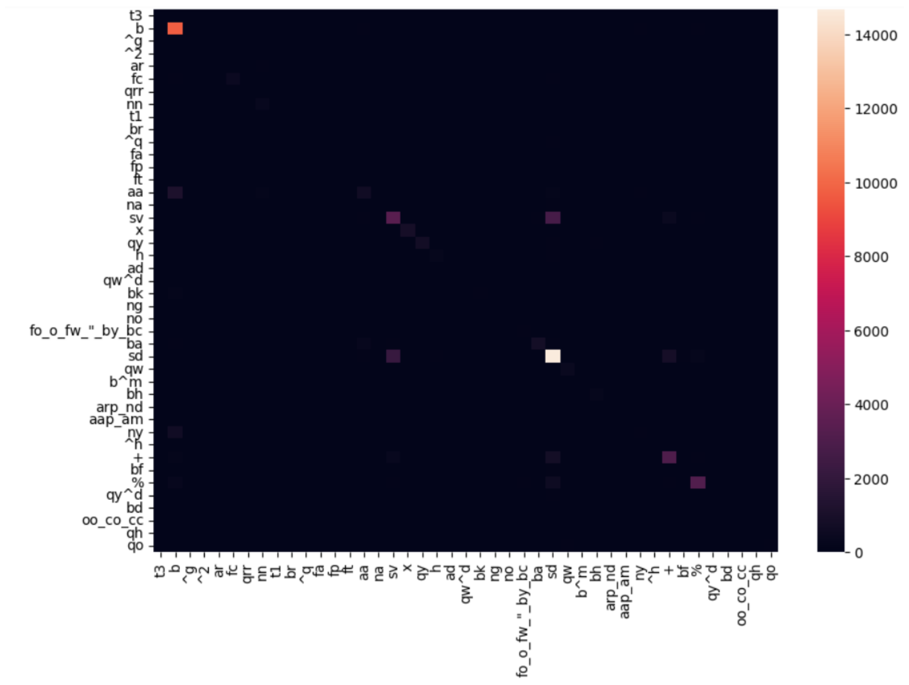


```
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_8 (InputLayer)           [(None, 150)]        0           []

 embedding_7 (Embedding)        (None, 150, 100)     4373200     ['input_8[0][0]']

 reshape_4 (Reshape)            (None, 150, 100, 1)  0           ['embedding_7[0][0]']

 conv2d_12 (Conv2D)             (None, 148, 1, 64)   19264       ['reshape_4[0][0]']

 conv2d_13 (Conv2D)             (None, 147, 1, 64)   25664       ['reshape_4[0][0]']

 conv2d_14 (Conv2D)             (None, 146, 1, 64)   32064       ['reshape_4[0][0]']

 batch_normalization_12 (BatchN  (None, 148, 1, 64)  256         ['conv2d_12[0][0]']
 ormalization)

 batch_normalization_13 (BatchN  (None, 147, 1, 64)  256         ['conv2d_13[0][0]']
 ormalization)

 batch_normalization_14 (BatchN  (None, 146, 1, 64)  256         ['conv2d_14[0][0]']
 ormalization)

 max_pooling2d_12 (MaxPooling2D  (None, 1, 1, 64)    0           ['batch_normalization_12[0][0]']
 )

 max_pooling2d_13 (MaxPooling2D  (None, 1, 1, 64)    0           ['batch_normalization_13[0][0]']
 )

 max_pooling2d_14 (MaxPooling2D  (None, 1, 1, 64)    0           ['batch_normalization_14[0][0]']
 )

 concatenate_9 (Concatenate)    (None, 1, 1, 192)    0           ['max_pooling2d_12[0][0]',
                                                                  'max_pooling2d_13[0][0]',
                                                                  'max_pooling2d_14[0][0]']

 time_distributed_4 (TimeDistri  (None, 1, 192)      0           ['concatenate_9[0][0]']
 buted)

 dense_17 (Dense)               (None, 1, 100)       19300       ['time_distributed_4[0][0]']

 dropout_9 (Dropout)            (None, 1, 100)       0           ['dense_17[0][0]']

 bidirectional_16 (Bidirectiona  (None, 1, 200)      160800      ['dropout_9[0][0]']
 l)

 bidirectional_17 (Bidirectiona  (None, 200)         240800      ['bidirectional_16[0][0]']
 l)

 dense_18 (Dense)               (None, 100)          20100       ['bidirectional_17[0][0]']

 flatten_10 (Flatten)           (None, 100)          0           ['dropout_9[0][0]']

 dropout_10 (Dropout)           (None, 100)          0           ['dense_18[0][0]']

 concatenate_10 (Concatenate)   (None, 200)          0           ['flatten_10[0][0]',
                                                                  'dropout_10[0][0]']

 dense_19 (Dense)               (None, 43)           8643        ['concatenate_10[0][0]']

==================================================================================================
Total params: 4,900,603
Trainable params: 4,900,219
Non-trainable params: 384
__________________________________________________________________________________________________
```

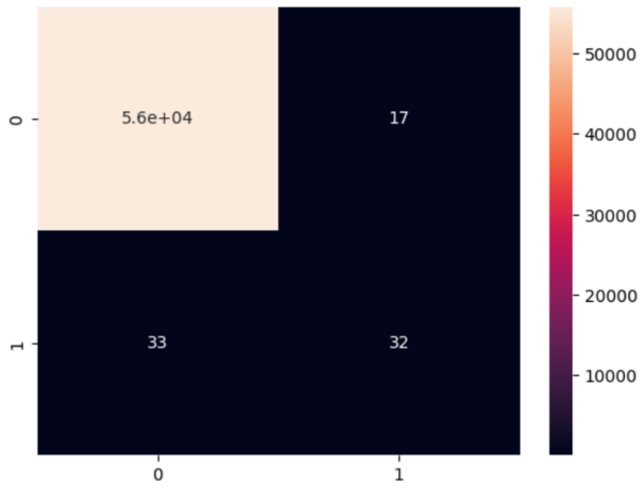
```
Epoch 1/5
1400/1400 [==============================] - 58s 29ms/step - loss: 1.0784 - accuracy: 0.6770 - val_loss: 0.9493 - val_accuracy: 0.7038
Epoch 2/5
1400/1400 [==============================] - 36s 26ms/step - loss: 0.8034 - accuracy: 0.7459 - val_loss: 0.9663 - val_accuracy: 0.7077
Epoch 3/5
1400/1400 [==============================] - 35s 25ms/step - loss: 0.6619 - accuracy: 0.7865 - val_loss: 1.0329 - val_accuracy: 0.7003
Epoch 4/5
1400/1400 [==============================] - 36s 26ms/step - loss: 0.5586 - accuracy: 0.8177 - val_loss: 1.1322 - val_accuracy: 0.6939
Epoch 5/5
1400/1400 [==============================] - 35s 25ms/step - loss: 0.4923 - accuracy: 0.8372 - val_loss: 1.2119 - val_accuracy: 0.6917
```

```
560/560 [==============================] - 10s 17ms/step - loss: 1.2072 - accuracy: 0.6965
```

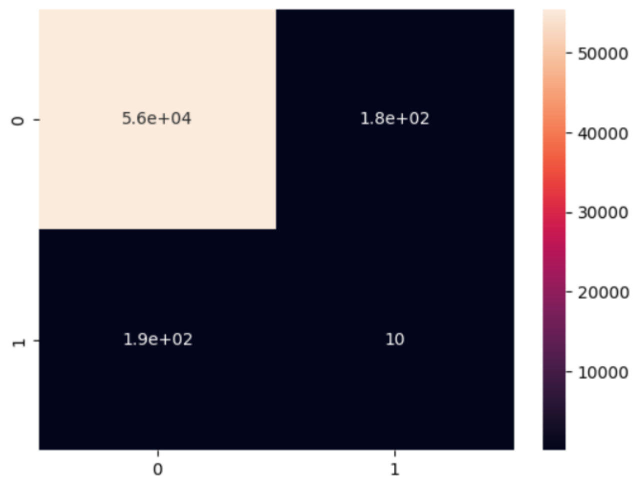
```
Overall Accuracy: 69.64688301086426
```



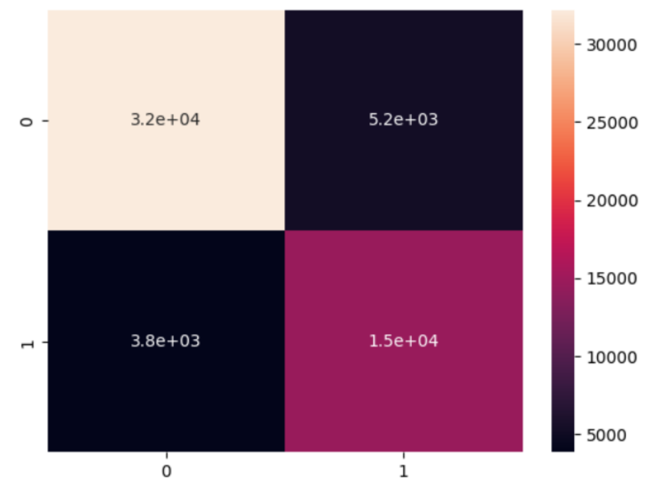
```
br accuracy: 99.91%
br precision: 65.31%
br recall: 49.23%
```



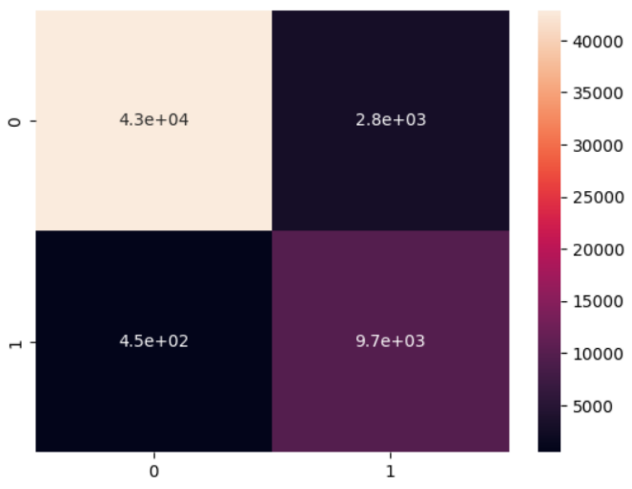
```
bf accuracy: 99.35%
bf precision: 5.35%
bf recall: 5.10%
```



```
sd accuracy: 83.84%
sd precision: 73.76%
sd recall: 79.36%
```



```
b accuracy: 94.15%
b precision: 77.56%
b recall: 95.56%
```

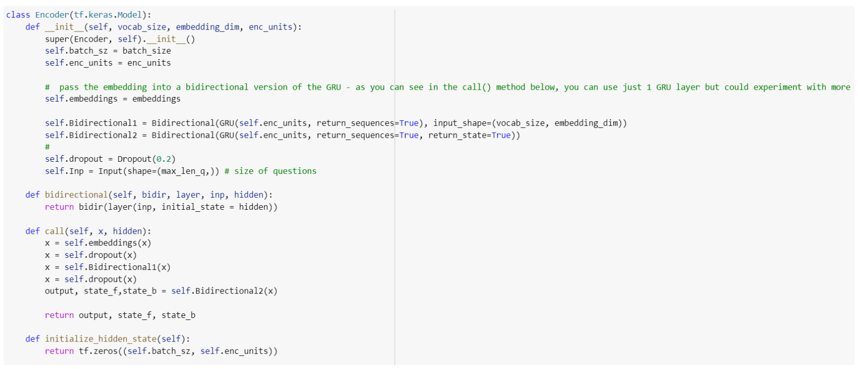


Compared to model 1:
“Br” - All the recorded metrics increase. Precision by about 23% and recall around 9%.
“Bf” - All metrics decrease and model 3 performs worse for this class.
“Sd” - The metrics for this class are about the same with small differences either way by about 1%.
“B” - The metrics for this class are about the same with small differences either way by about 1%.

So clearly, this new model taking context into account performs better for minority classes whilst preserving the performance on majority classes. A frequent error is that the class “Bf” always performs badly. This could be due to the fact that instances of “Bf” might be similar in structure to instances of a more dominant class, so almost all instances of “Bf” are mistaken for that more dominant class. The best way to improve this would be to introduce more instances of “Bf” during training.

Part E: Dialogue 2: A Conversational Dialogue System

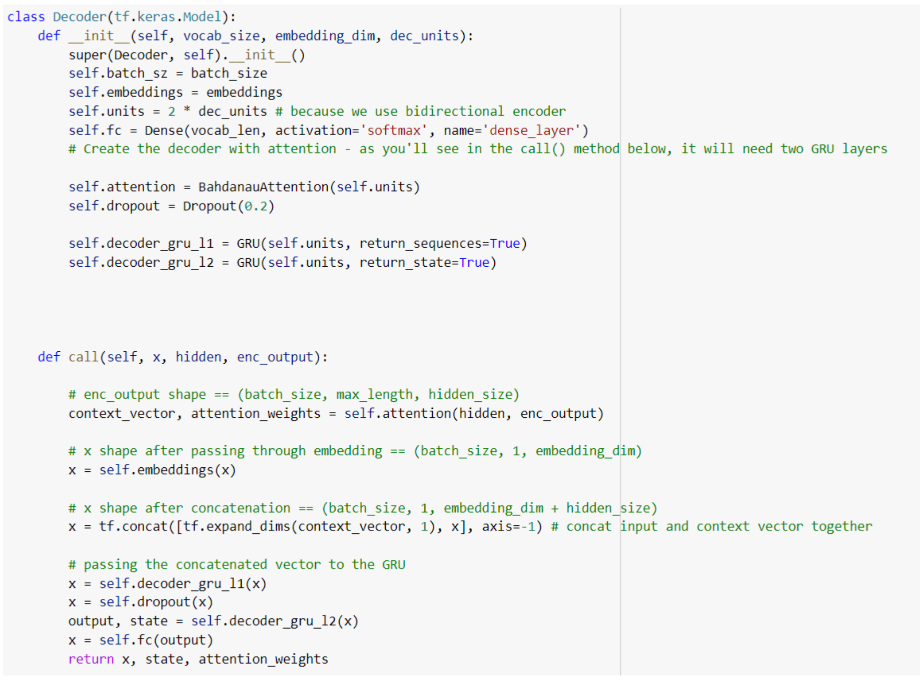
1.



```python
class Encoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, enc_units):
        super(Encoder, self).__init__()
        self.batch_sz = batch_size
        self.enc_units = enc_units

        #  pass the embedding into a bidirectional version of the GRU - as you can see in the call() method below, you can use just 1 GRU layer but could experiment with more
        self.embeddings = embeddings

        self.Bidirectional1 = Bidirectional(GRU(self.enc_units, return_sequences=True), input_shape=(vocab_size, embedding_dim))
        self.Bidirectional2 = Bidirectional(GRU(self.enc_units, return_sequences=True, return_state=True))
        #
        self.dropout = Dropout(0.2)
        self.Inp = Input(shape=(max_len_q,)) # size of questions

    def bidirectional(self, bidir, layer, inp, hidden):
        return bidir(layer(inp, initial_state = hidden))

    def call(self, x, hidden):
        x = self.embeddings(x)
        x = self.dropout(x)
        x = self.Bidirectional1(x)
        x = self.dropout(x)
        output, state_f,state_b = self.Bidirectional2(x)

        return output, state_f, state_b

    def initialize_hidden_state(self):
        return tf.zeros((self.batch_sz, self.enc_units))
```

Encoder simply uses an embedding with 2 bidirectional GRUs. The outputs of the last GRU are returned to be passed to the next layer (the Decoder).

2.

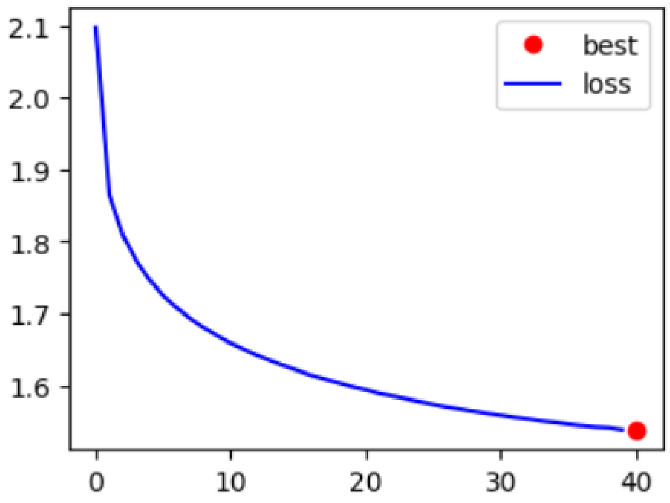


```python
class Decoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, dec_units):
        super(Decoder, self).__init__()
        self.batch_sz = batch_size
        self.embeddings = embeddings
        self.units = 2 * dec_units # because we use bidirectional encoder
        self.fc = Dense(vocab_len, activation='softmax', name='dense_layer')
        # Create the decoder with attention - as you'll see in the call() method below, it will need two GRU layers

        self.attention = BahdanauAttention(self.units)
        self.dropout = Dropout(0.2)

        self.decoder_gru_l1 = GRU(self.units, return_sequences=True)
        self.decoder_gru_l2 = GRU(self.units, return_state=True)



    def call(self, x, hidden, enc_output):

        # enc_output shape == (batch_size, max_length, hidden_size)
        context_vector, attention_weights = self.attention(hidden, enc_output)

        # x shape after passing through embedding == (batch_size, 1, embedding_dim)
        x = self.embeddings(x)

        # x shape after concatenation == (batch_size, 1, embedding_dim + hidden_size)
        x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1) # concat input and context vector together

        # passing the concatenated vector to the GRU
        x = self.decoder_gru_l1(x)
        x = self.dropout(x)
        output, state = self.decoder_gru_l2(x)
        x = self.fc(output)
        return x, state, attention_weights
```

Takes the inputs from the Encoder and applies the embedding. Then, the context computed from the attention is concatenated to the output of the embedding. Then, the output is passed through 2 GRUs and then finally a Dense output with softmax activation.

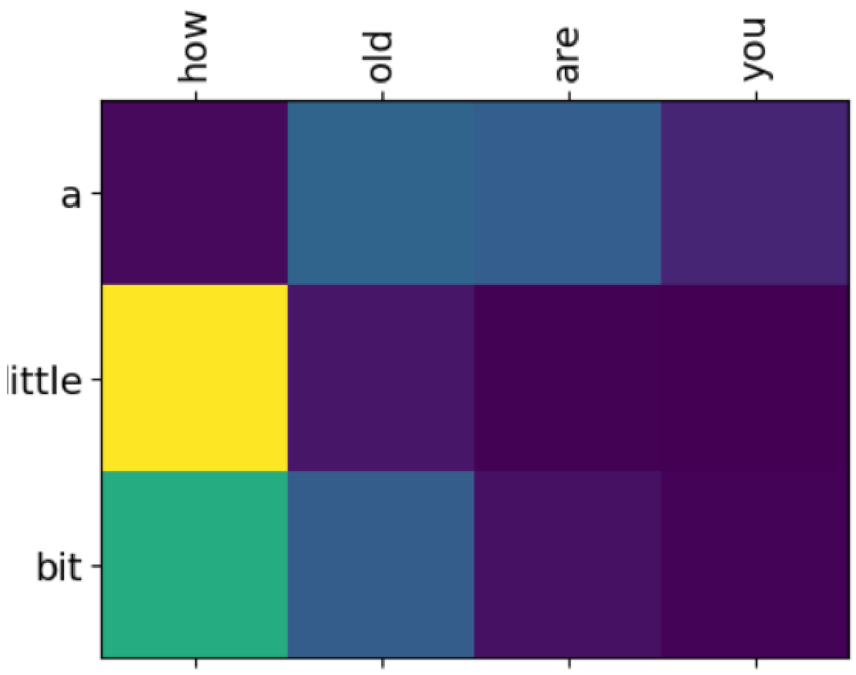
3.

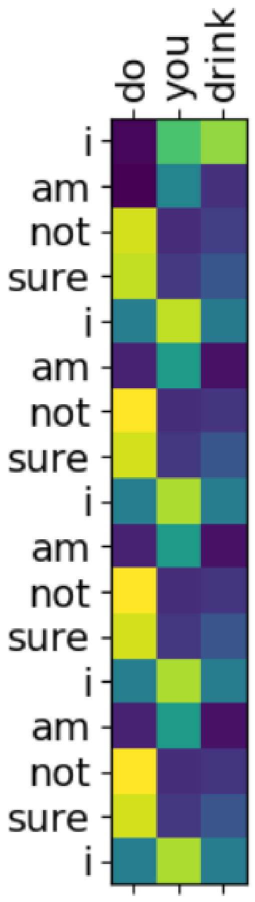
NOTE: I was only able to run this section for 40 Epochs due to limitations on colab GPU.



```
Best epoch so far:  40
Time  199.083 sec

*** Epoch 40 Loss 1.5387 ***

####################
Greedy| Q: Hello   A: hello
%
Greedy| Q: How are you ?  A: i am a dentist
%
Greedy| Q: What are you doing ?  A: i am not going to do
%
Greedy| Q: What is your favorite restaurant ?  A: i am not going to be a little
%
Greedy| Q: Do you want to go out ?  A: i am sorry
####################
check point saved!
```

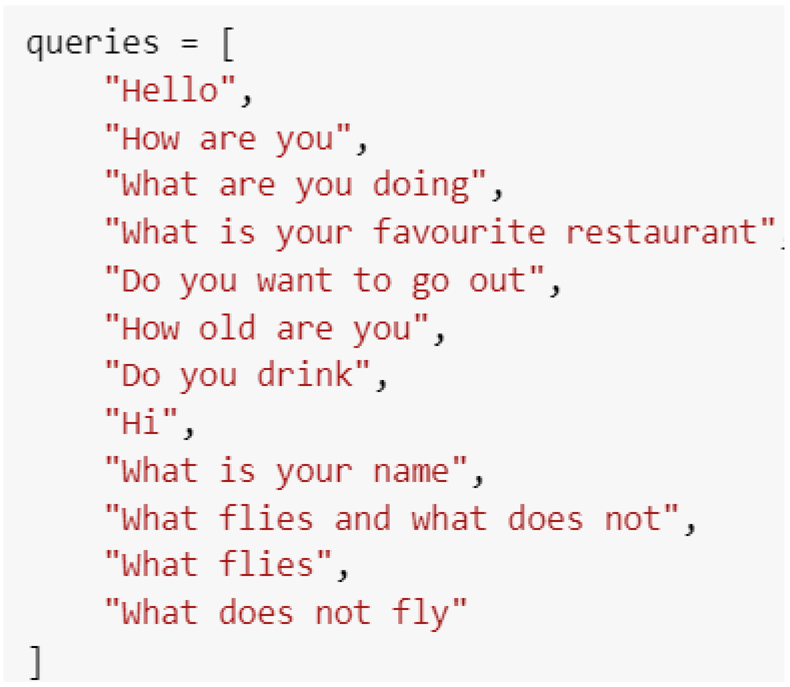
The following 2 Attention Weght heatmaps are for Epoch 40.





To evaluate the performance of my model, I will infer answers to the queries stated below for epochs 5, 30, and 40.

Note due to my report getting lengthy, I will only show the Attention Weight Heatmaps most relevant to the question.



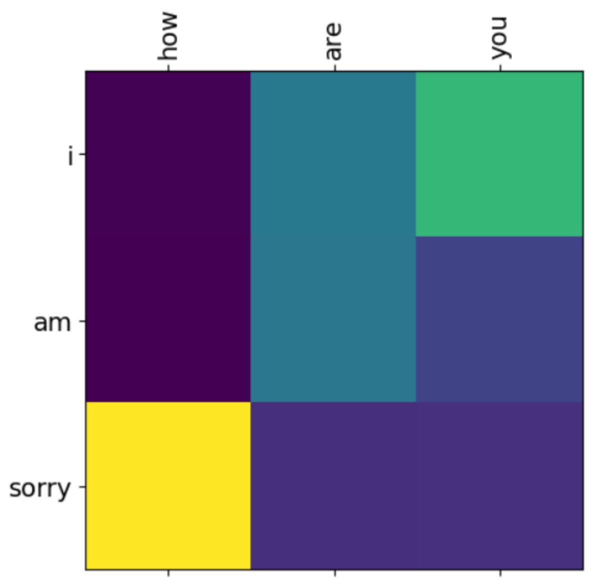
```
queries = [
    "Hello",
    "How are you",
    "What are you doing",
    "What is your favourite restaurant",
    "Do you want to go out",
    "How old are you",
    "Do you drink",
    "Hi",
    "What is your name",
    "What flies and what does not",
    "What flies",
    "What does not fly"
]
```

1. Did the models learn to track local relations between words?

Yes, the model was very good at this. For example, if the query contains “you” (As expected when asking someone a question) the answer tends to begin with “I”. It learns this relation very early on and it is evidenced by the Attention Weights Heatmap. It was also very good at learning that when referring to itself, I.e., beginning the sentence with “I”, the next word should be “am”.

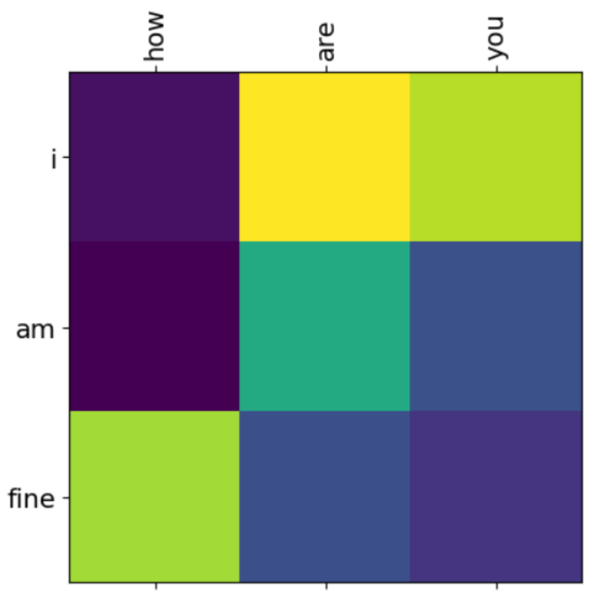
Epoch 5:

```
Input: how are you
Predicted answer: i am sorry
```



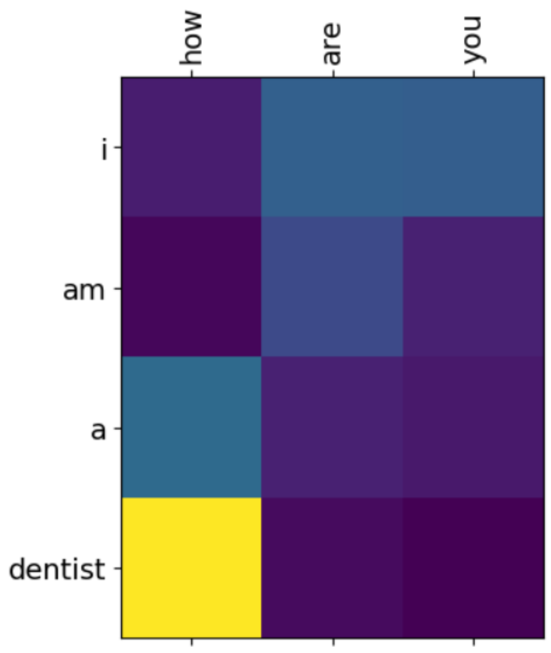
Epoch 30:

```
Input: how are you
Predicted answer: i am fine
```



Epoch 40:

```
Input: how are you
Predicted answer: i am a dentist
```

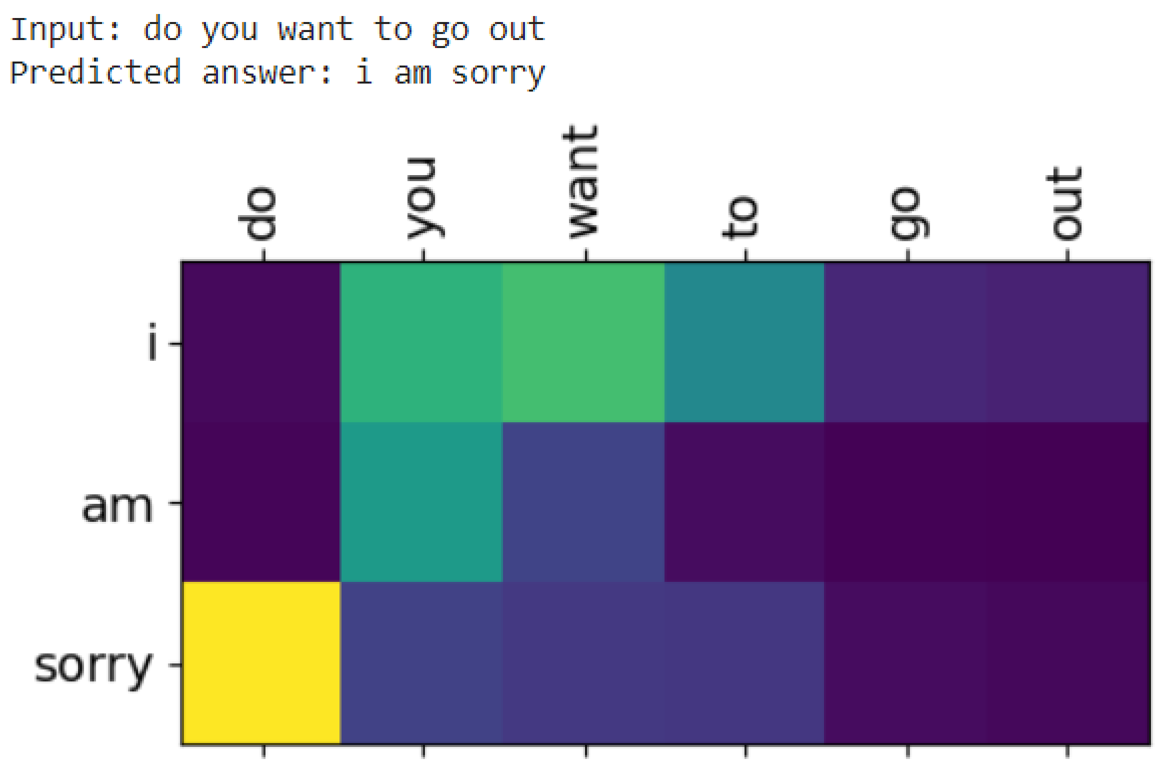


2. Did the models attend to the least frequent tokens in the utterance? Can you see signs of overfitting in models that hang on to the least frequent words?

For a lot of the Queries, the model has "sorry" in the answer. The model learns this quite early on and struggles to overcome it. For example, looking at Epoch 30, the answer contains "sorry" even though it isn't relevant. This is still prevalent in Epoch 40. And the Attention Heatmap. Looking at Query "Do you want to go out", the answer is "I am sorry". This is likely due to the fact that the model has learnt that if the query contains "you", the best answer will start with "I". But this may not always be the best answer. The model is clearly overfitting and as such is struggling to find tokens to finish the sentence. On Epoch 5, the answer for the same query is "I am not". This supports my theory that the model is overfitting and will almost always respond starting with "I" if the query contains "you".

```
*** Epoch 30 Loss 1.5613 ***

####################
Greedy| Q: Hello   A: hello
%
Greedy| Q: How are you ?  A: i am fine
%
Greedy| Q: What are you doing ?  A: i am sorry
%
Greedy| Q: What is your favorite restaurant ?  A: i am not going to be here
%
Greedy| Q: Do you want to go out ?  A: i am sorry
####################
check point saved!
```

Epoch 40:



```
Input: do you want to go out
Predicted answer: i am sorry
```

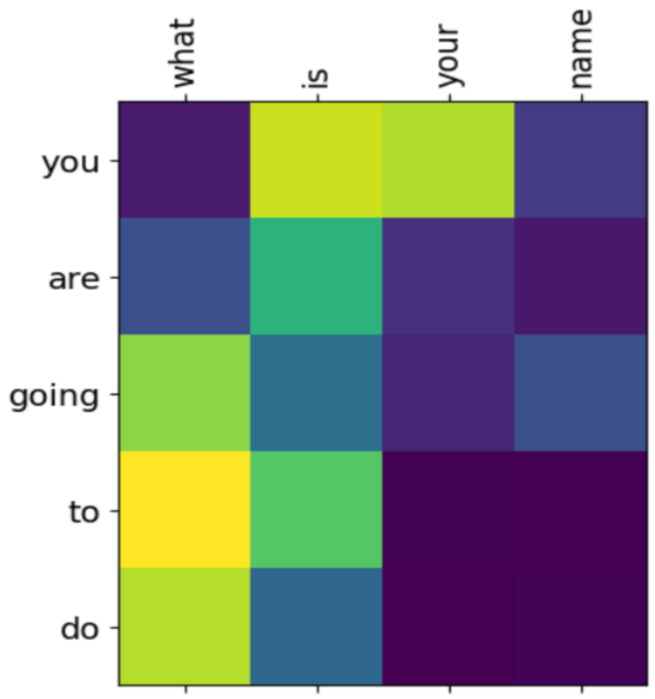
3. Did the models learn to track some major syntactic relations in the utterance (e.g. subject-verb, verb-object)?

Yes, the model performs quite well with regard to this. The answers makes sense syntactically. The model responds to the query “How are you” on epoch 40, with the answer “I am a dentist”. Even though this answer isn’t relevant to the question, the answer still makes syntactic sense. The model is easily able to form an answer that has the correct syntactic structure.

4. Do they learn to encode some other linguistic features? Do they capture part-of-speech tags (POS tags)?

The model struggles a lot with semantics. Semantic meaning is a lot harder to capture and reproduce. This is evidenced by the Attention Weight Heatmaps on question 1 above. The answers make sense syntactically but struggle in making sense. Below shows a more extreme example where the syntax is perfect but the semantics are completely incorrect:
Epoch 40:

```
Input: what is your name
Predicted answer: you are going to do
```



With regards to POS tagging, the answers from the model are quite unambiguous and as such don't really stretch the model in terms of pos tagging so it is difficult to answer. But I believe that a dialogue system does capture the POS tag information as this is very useful for helping it deconstruct and construct sentences.

5. What is the effect of more training on the length of responses?

It varies depending on the query. For example, on Epoch 1, the answer to the query "Hello" is "i am sorry" but on the 4th epoch the model learns to respond with "hi" and then a few epochs later with "hello". And even by epoch 40, the answer is "hello". Here the model has decreased the length of its answer. But looking at the query "do you drink", the answer on epoch 5 is "I am not", on epoch 30 is "I am not going to be" and finally on epoch 40 is "I am not sure i am not sure i am not sure i am not sure i am". The model's answer here is getting longer as the training increases. I think for some queries, like "hello", the model is very confident in its answer and as such only responds with a short and succinct answer, like "hello". But in some cases, once the model learns more about the relationship between the word and the structure of the utterances, it generates longer answers as the model has a better understanding of the language and patterns.

6. In some instances, by the time the decoder has to generate the beginning of a response, it may already forget the most relevant early query tokens. Can you suggest ways to change the training pipeline to make it easier for the model to remember the beginning of the query when it starts to generate the response?

Could try experimenting with different attention mechanisms and different forms of it. For example, using Luong attention instead. Or also using self-attention to help carry relevant information forward in the encoder/decoder. Could also try muliheaded attention to help capture a range of information that may help the model capture relevant early query tokens. Could increase the embedding size as this allows for more information to be encoded and as such a greater potential for relevant information. Could increase the number of neurons in the bidirectional GRUs. Could try different positional encodings as this may help the model extract the relevant information.