Capstone Project

Automatic Ticket Assignment

GitHub Link to the code

Index

1. Summary of Problem statement , Data and Findings

1.1. Problem Statement

The IT industry today has the policy of keeping the lights on, which is to provide support round the clock for the tools and technologies so the work can go unhindered. This is done by resolving the issues as soon as possible as they arise.
When a problem/issue arises the user provides it in the form of a ticket. These issues are sorted out based on certain paradigm so it can be allocated to a certain group who handles that issue. This whole process can be very time consuming.
Our project is to automate the process of the allocation of the tickets to their respective groups to avoid the issues of wrong allocation and fasten the process

1.2. Data and findings

The Dataset has 4 columns. The ***short description*** is like the headline/subject of the problem. The ***description*** column is the detailed explanation of the problem. The ***caller*** tells us the user who filed the issue. The caller column is masked. The final column is the ***assignment group*** which is the group to which the ticket has been assigned. This gives us the information about which support team the ticket has been allocated to and thus this column will be the target column that we have to predict. We have 74 labels i.e 74 groups in this column and we have to predict which group the ticket must be assigned to. The data set has 8500 records in total i.e 8500 tickets.

2. Overview of EDA and Preprocessing

2.1. Handling missing Values

First we checked for the presence of missing values. There were some missing values in the description and short description column. On further analysis we found that the missing values were present on either of the columns but not on both. We thus decide to not drop any column. As long as we have either description or short description we are good to go. So instead of dropping columns we replaced the NULL values with empty strings and filled the empty rows in short description column with corresponding description row and empty data in description row with corresponding short description row. Luckily we had only 9 such rows where either short description or description was missing.

2.2. Dropping the columns

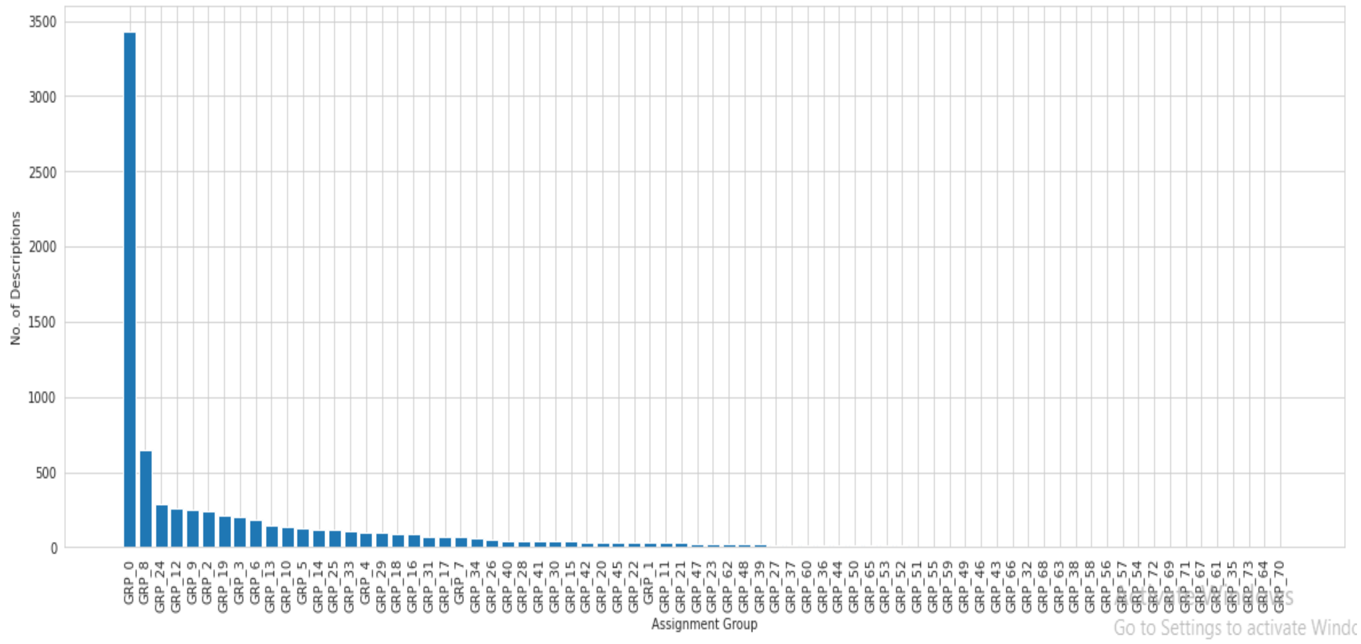
The caller column was removed. It is the name of the caller who raised the ticket. We choose to remove this column since it has no significance whatsoever in predicting which group the ticket must be assigned to.

2.3. Duplicate entries

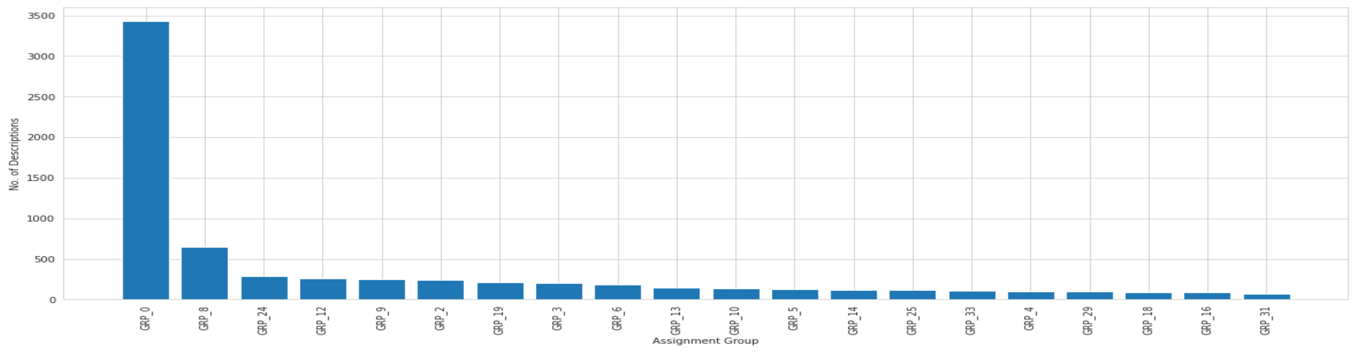
Next step we did was to check for duplicates. There were 591 duplicates found in the data set. So we removed the duplicates but kept the first instances on every entry.

2.4. Analysis of Target Variable

```
GRP_0    0.433557
GRP_8    0.081553
Name: Assignment group, dtype: float64
```

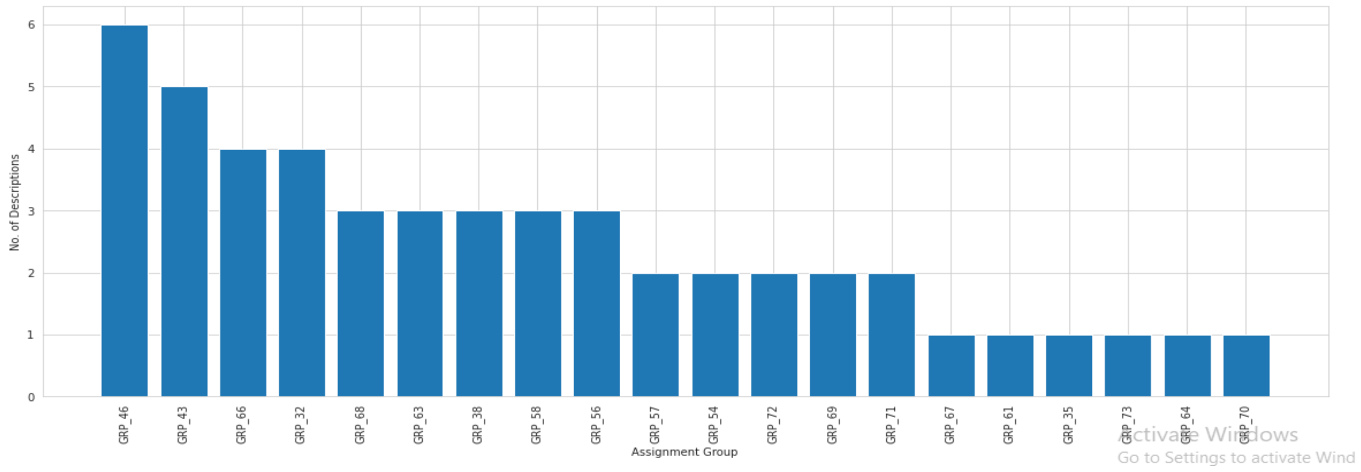


From the above plot we can see the distribution of Target Variable i.e. Assignment Group column. Highly skewed distribution with 43% of the tickets belonging to group 0. The next closest group which has the most number of tickets is group 8 with 8% of the tickets assigned to it.



Above plot is the Top 20 groups to which the ticket has been assigned

Below plot shows the least 20 groups



At this point we would like to bring your attention to the last 6 groups in this plot as they have just 1 ticket assigned to them. We will explain why we mentioned this as we move ahead.

2.5. Language Detection

We used the python's langdetect library to detect languages other than English. Surprisingly we have other languages. This tells us that we may have some unicode characters in the dataset which we need to fix. Therefore using the ftfy python package we fixed the unicode characters.

2.6. Cleaning Data

The descriptions and short descriptions had punctuations, links, email addresses, different date and time formats, IP addresses and other alpha numeric words and jargons. We created a make tokens function which will remove all the punctuations, lowercase the word, escape html entities, remove email ids and corresponding jargons, removes date and time, replaces contractions and special characters, removes stopwords and lemmatizes the text.

For IP address and hyperlinks we replaced the **actual IP addresses** with the **word "IP address"** and **actual hyperlinks** with the **word "hyperlink"**.

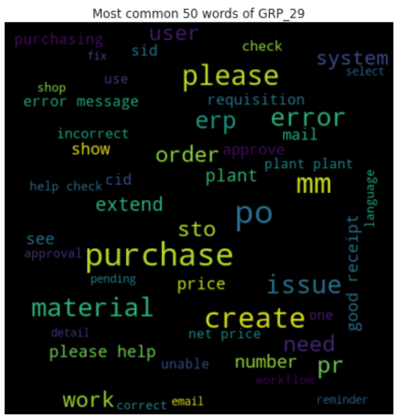
To summarize the cleaning process we cleaned specific patterns like emails, hyperlinks, IP addresses, different date and time formats, special characters, html entities, contractions etc. but retained alpha numeric jargons and words since they are specific to the corpus.

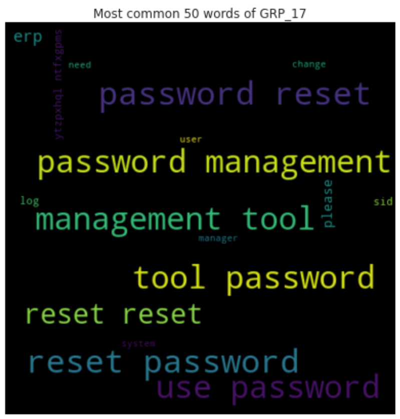
After cleaning both the columns, we checked for missing rows again. As before, if either of the two columns had text we retained them but the ones where both description and short description were empty, we dropped them. It turned out that there were perhaps 32 such rows where both description and short description were empty. This was because after cleaning the rows that **only** had stopwords, emails or unicode characters were cleaned completely, hence we dropped these 32 rows.

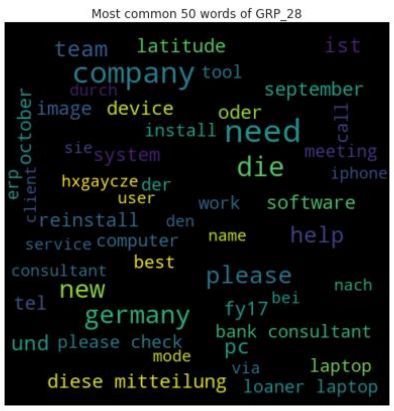
Note: We saved the cleaned text in cleaned_short_dec and cleaned_desc.

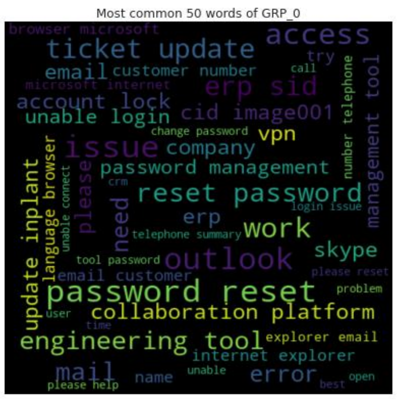
2.7. Creating word clouds

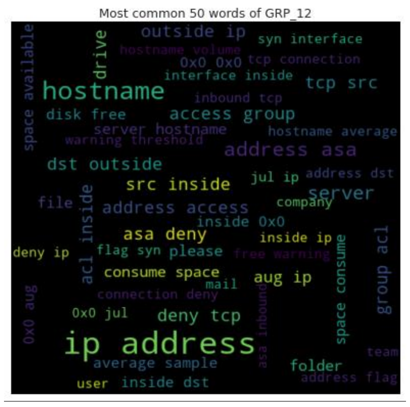
Finally we created the word clouds using the python's word cloud library. We created word clouds for every target label i.e. total 74 word cloud plots. Plotting 6 of them below:

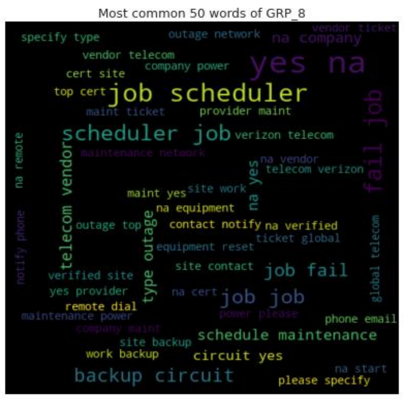












3. Deciding Models and Model Building

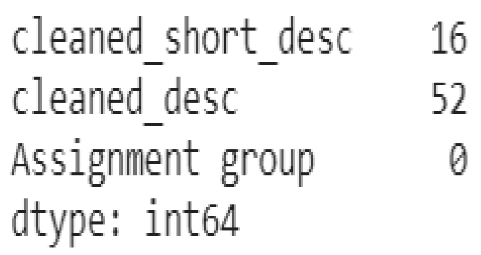
Here we end the pre-processing of Data. **Next up is to decide the modelling approach.** As a part of this milestone we will train base models and gauge which models may or may not work, based of on which we will decide a final model for performance tuning in milestone 2. **We have two base approaches for this milestone:**

In **base model approach 1** we will convert our corpus into Tf-Idf and BagOfWords matrices and train traditional Machine Learning models on it.

In **base model approach 2** we will go for Deep Learning models.

Both these approaches will require respective pre-processing and data transformation which we will see

3.1. STARTING WITH BASE APPROACH – 1

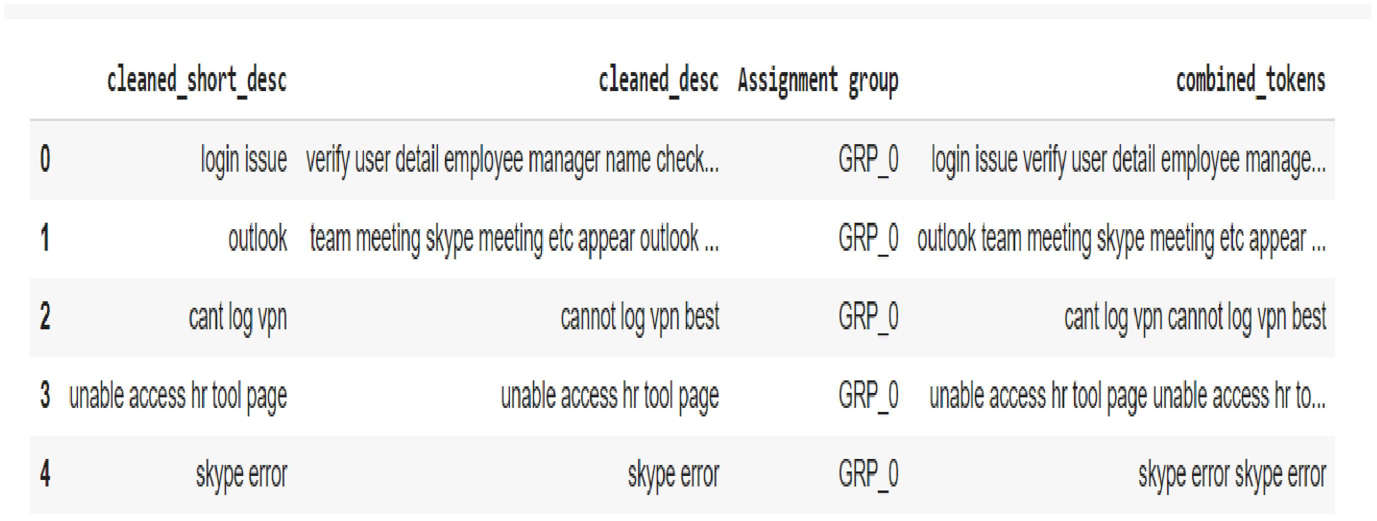


```
cleaned_short_desc    16
cleaned_desc          52
Assignment group       0
dtype: int64
```

After loading the pre-processed dataset, we first check for missing values again. As expected we have some missing values but remember that in pre-processing we removed only those records that had missing values in both description and short description columns. If either of the two had text, we retained it. These missing values were earlier saved with an empty string in the dataframe during pre-processing, but when we converted it to csv and loaded it again, the empty strings were displayed with NaN. We will simply replace these NaNs with empty strings again.

3.1.1. Preparing Data for transformation

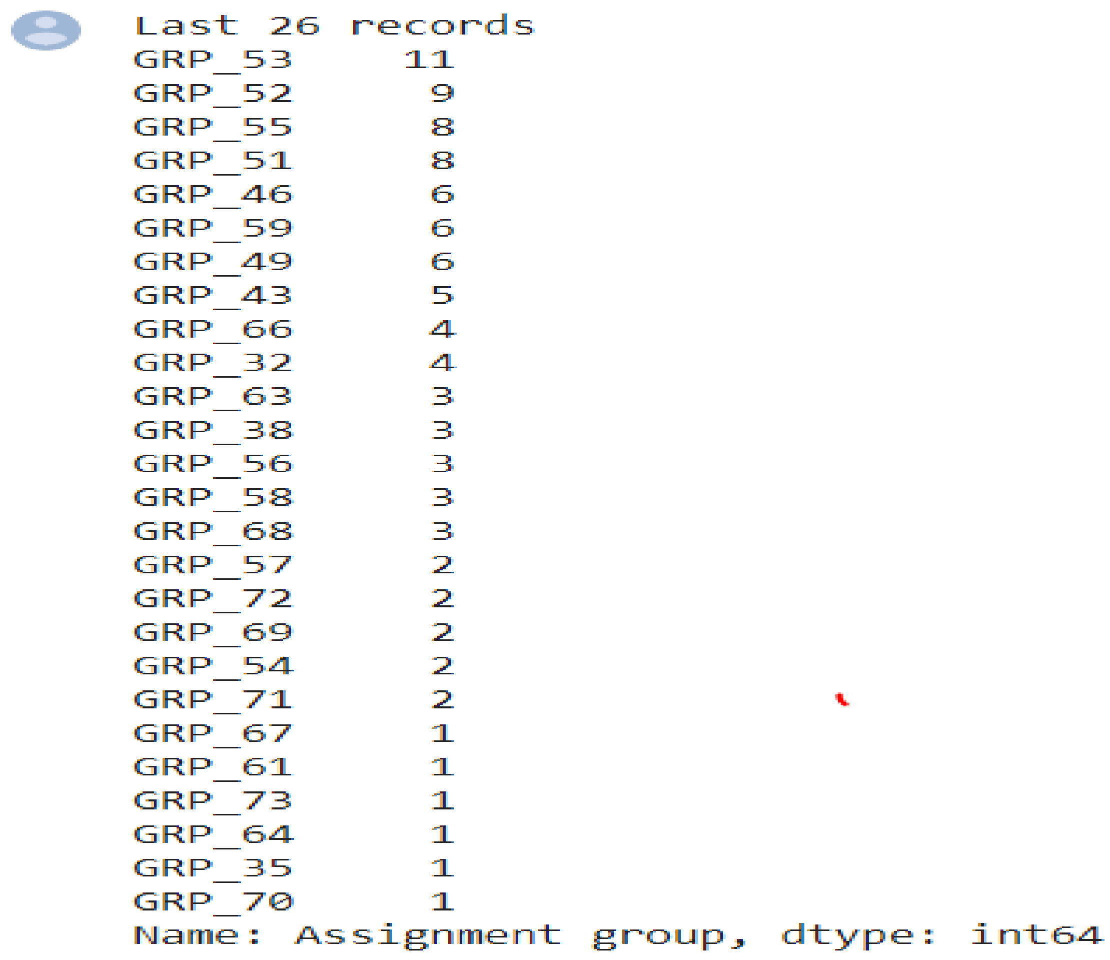
The first method that we'll be taking up is transforming the corpus into **tf-idf** and **bag of words** matrices; hence **we will combine the cleaned_desc and cleaned_short_desc column into one column named "combined_tokens**". This was the reason we retained all the rows that had either description or short description



	cleaned_short_desc	cleaned_desc	Assignment group	combined_tokens
0	login issue	verify user detail employee manager name check...	GRP_0	login issue verify user detail employee manage...
1	outlook	team meeting skype meeting etc appear outlook ...	GRP_0	outlook team meeting skype meeting etc appear ...
2	cant log vpn	cannot log vpn best	GRP_0	cant log vpn cannot log vpn best
3	unable access hr tool page	unable access hr tool page	GRP_0	unable access hr tool page unable access hr to...
4	skype error	skype error	GRP_0	skype error skype error

3.1.2. Preparation for stratified split

Before moving ahead, in analysis and pre-processing we saw that 6 target classes had only 1 record. Showing last 26 records:



```
Last 26 records
GRP_53    11
GRP_52     9
GRP_55     8
GRP_51     8
GRP_46     6
GRP_59     6
GRP_49     6
GRP_43     5
GRP_66     4
GRP_32     4
GRP_63     3
GRP_38     3
GRP_56     3
GRP_58     3
GRP_68     3
GRP_57     2
GRP_72     2
GRP_69     2
GRP_54     2
GRP_71     2
GRP_67     1
GRP_61     1
GRP_73     1
GRP_64     1
GRP_35     1
GRP_70     1
Name: Assignment group, dtype: int64
```

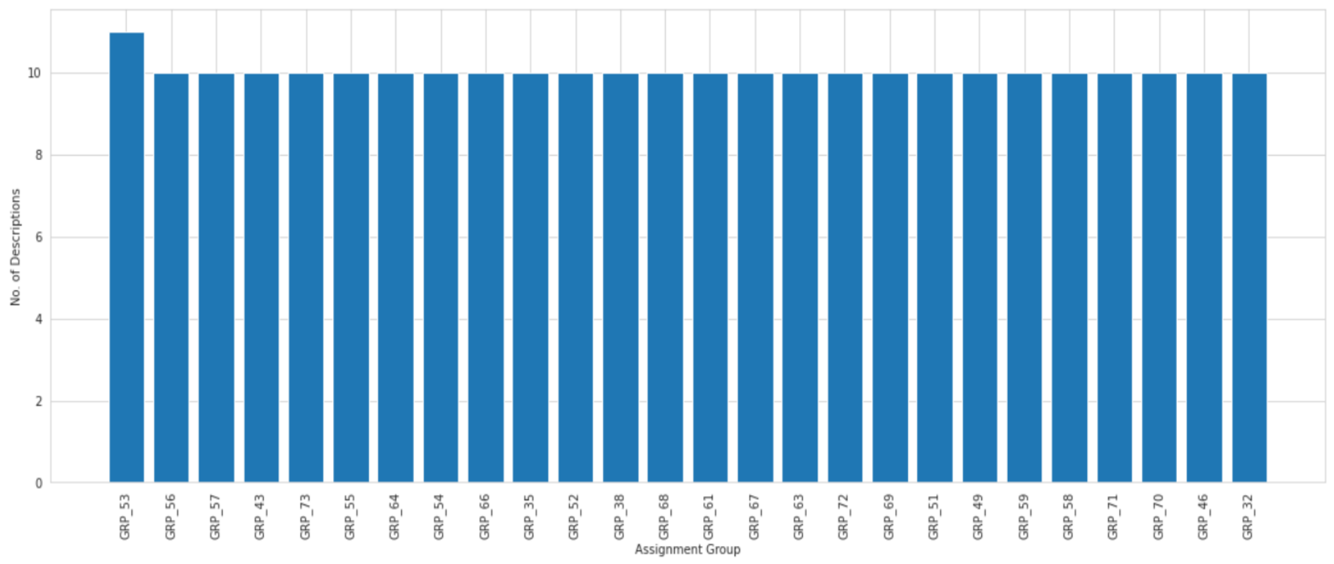
In the last 6 records, we have only one ticket assigned to each of them. As a result of this, we will not be able to split in a stratified fashion. A stratified split means to split in train and test split such that the target class distribution is retained. Eg: If the target class had 3 labels: A(75%), B(20%) and C(5%), then the train and test sets, both will have the same proportion of A, B and C as in the original data.

Now to make a stratified split every target label class should have at least 2 records, so that it can be sent to both train and test set. We see that 6 of our labels have only 1 record.

WHY DO WE HAVE TO HAVE A STRATIFIED SPLIT?

We'll have a problem while calculating the ROC AUC of the multiclass labels. We need to have a stratified split to ensure that both the sets (train and test) have all the target classes. E.g. Let's say after splitting without stratify, our train set has data points belonging to these 5 classes; A, B, C, D and E and test set has data points belonging to only A, C, D and E. Since we'll train on the train set and predict on

the test set; it is very likely that at least one of the test data points will be predicted as B, but there is no B in the test labels. This will cause error while calculating the ROC AUC or other evaluation metrics.
Now to overcome this issue we will slightly upsample some of our minority classes. If you'll look at the last 26 records in the above graph, group 53 has 11 tickets assigned and except that all other groups have less than 10 tickets assigned to it. Hence we will **upsample** all the groups having less than 10 tickets to 10. With this upsampling, each group will be **upsampled to 10** simply by duplicating the data points in that group.



gle.com/drive/search?q=owner%3Ame %28type%3Aapplication%2Fvnd.goo...

We see in above figure that all groups having less than 10 records have been sampled to 10. Now we can make a stratified split. After label encoding the target column we made a 80:20 train test split.

3.1.3. Converting to BagofWords and TF-IDF

We then convert the train and test sets into **BagOfWords and Tf-Idf matrix using sklearns count vectorizer and Tf-Idf library**. We made use of two parameters for this: **maxdf** and **max_feat**. maxdf ignores terms/words that have a document frequency higher than 'maxdf' threshold. In other words, it ignores terms that appear in more than 'maxdf' number of documents/records. We use **maxdf = 0.8**. max_feat on the other hand builds a vocabulary considering only top "max_features" ordered by term frequency across the corpus. We set **max_feat = 2000.** This is the shape of our 4 matrices:

Tf-Idf and BOW train set (6432, 2000)
Tf-Idf and BOW test set (1608, 2000)

3.1.4. Trying Base models

LinearSVC, Logistic Regression and Random Forest will be the first set of base models of our choice.

- Although we are using **Random Forest** model but it is expected that it won't be as good as our other two choices because theoretically tree based models flunk in higher dimensions and we have a massive dimensionality of 2000 after Tf-Idf and BOW data transformation
- **k-nearest neighbors** may or may not work but given the size of the dataset and the dimensions training KNN's would be painfully slow. Hence training a KNN model is not computationally viable.
- Naive Bayes can also be ignored in this use case even though it performs in higher dimensions because we have a highly skewed target class. High skewness especially affects Naive Bayes because prior probability of minority class tends to be zero

Note that we are using LinearSVC instead of classical SVC with different kernals since they will be computationally expensive to train as well. Some key differences that make LinearSVC faster is as follows:

- For multi-class classification problem SVC fits N * (N - 1) / 2 models where N is the amount of classes. LinearSVC, by contrast, simply fits N models.
- The underlying estimator for **LinearSVC is liblinear** while SVC uses libsvm estimator. liblinear estimators are optimized for a linear case and thus converge faster on big amounts of data than libsvm. That is why LinearSVC takes less time to solve the problem.
- In practice the complexity of the SVC algorithm (that works both for kernel and linear SVC) as implemented in libsvm is O(n^2) or O(n^3) whereas liblinear is O(n) but it does not support kernel SVCs.

Also we will be calculating ROC_AUC for the models which requires us to predict class probabilities. Logistic Regression by default calculates probabilities which can be obtained simply by predict_proba() but SVM's do not. **SVC can calculate the probabilities if we set probability = True but LinearSVC does not have support for this. Hence to obtain probabilities for LinearSVM we will be wrapping it inside a CalibratedClassifier**

How does it calibrate?
First it finds probabilities with which LinearSVC classifies (The same way regular SVC does when we set probability = True). This is done by calculating p(j|x) i.e given a sample x, knowing its distance from the separating hyperplane and the class to which it belongs, it finds the probability with which x belongs to class j
Then the CalibratedClassifier further calibrates these probabilities by fitting a regressor that maps the output of the classifier (LinearSVC) to the calibrated probability between 0 and 1. In simpler words, it tries to predict p(yi=1|x) for a sample x

Before moving ahead lets have a look at our evaluation metric. We will print f1-score, accuracy as well as ROC-AUC score. Ideally, ROC-AUC is for binary classification but it can be modified for multi-class as well. We will use sklearn's metric library for the same. We created a ModelTuningAnd Evaluation module that consists of evaluation_metric function that prints all these 3 metrics. This function takes two special arguments; **'multiclass'** and **'avg'.**

- **multiclass argument** can take 2 string values: 'ovr' which implements OneVsRest and 'ovo' which implements OneVsOne. 'ovr' can be implemented with models that natively support multi-class classification like logistic regression, neural networks etc. while 'ovo' can be used with others as well. Especially 'ovo' is preferred for SVM's and other kernel methods because these methods do not scale in proportion to the size of the training dataset and take a large amount of time to train when the dataset is huge. Hence for these methods we train dataset in subsets which may affect the evaluation metrics. **We will use 'OVR' for this case.**

- **avg argument** can take 3 string values**: "micro", "macro" and "weighted**". We will focus on micro and macro mainly:
  - **Micro:** let's look at micro averaging with an example. Say we are calculating precision for a multiclass problem. Precision is calculated by TP/TP+FP. Now since we have a multi class problem we will have TP's and FP's for every class. Hence the average precision with 'micro' is given by (sum of all TP's)/(sum of all TP's) + (sum of all FP's) i.e (TP1+TP2+...+TPn)/(TP1+TP2+...+TPn) + (FP1+FP2+...+FPn)
  - **Macro:** macro averaging on the other hand is pretty straight forward. Again taking the example of precision, it calculates precision individually for each class and averages it. Eg: Let's say we have 'n' classes. Therefore we will have 'n' precision scores (p1, p2, p3...., pn). . Now the average precision score should be (p1+p2+….pn)/n.

When we have balanced classes micro and macro metrics will be almost similar, however when we have class imbalance, macro averaging will always be less than micro; since the metric calculated for each class is individually averaged. So let's say if a particular class has very low recall (or precision) compared to other classes, the mean will be affected because of that class and will decrease

Also according to the sklearn's documentation... when micro averaging, micro F1-Score = micro precision = micro Recall = Accuracy in case of multi-class classification where a data point belongs to exactly 1 class. This behaviour can be explained with a simple example:
Precision = tp/tp+fp and Recall = tp/tp+fn. So numerator is same. In denominator we have, fp and fn. Now, fn with respect to one class is same as fp with respect to another class, hence fp=fn, and so now denominator is also same. Therefore precision = Recall and thus harmonic mean between the two i.e. f1-score will also be same in case of micro-averaging. Now if we use micro-average , say for n classes , Formula for micro precision is [Sum of all True positive/(Sum of all True positive + Sum of all False positive)].

Since fp=fn; the denominator becomes all positives + all negatives i.e. total number of samples and numerator is all positives (tp) i.e. all correctly classified samples. **So all** correctly classified/total no. of samples = accuracy**.** For minor averaging, f1 score = accuracy = recall= precision.

In a broad sense, 'micro' gives the overall performance of the model despite having class imbalance while 'macro' takes into account the effects of minority classes and gives the performance accordingly.

Hence we will monitor macro metrics i.e macro ROC-AUC and macro F1-Score ----and---- accuracy since it will give the overall performance of the model and is equal to micro metrics"

NOTE THAT:

- ROC-AUC only support macro averaging; while f1-score supports both macro and micro
- F1-score does not have multiclass parameter hence it does not take 'ovo' or 'ovr' into account but ROC-AUC supports multiclass parameter hence we'll be using 'ovo'
- Accuracy is independent of these two parameters. They are not applicable to accuracy. But since accuracy will give the micro performance (i.e. overall performance of the model despite class imbalance), we will monitor it too

After training the 3 base models we have the following performance. Note that this is base model performance with default parameters. We thus conclude the base approach 1 here. In 2nd approach we will train base deep learning model; LSTM which widely benefits sequential modelling.

	BOW					
	Accuracy		**Macro F1-Score**		**Macro ROC_AUC**	
Model	**Train**	**Test**	**Train**	**Test**	**train**	**test**
LinearSVC	0.93657	0.63308	0.93285	0.48332	0.99946	0.89622
Logistic Regression	0.91216	0.67724	0.90373	0.51452	0.99895	0.94223
Random Forest	0.95756	0.64677	0.95321	0.45274	0.99942	0.91447

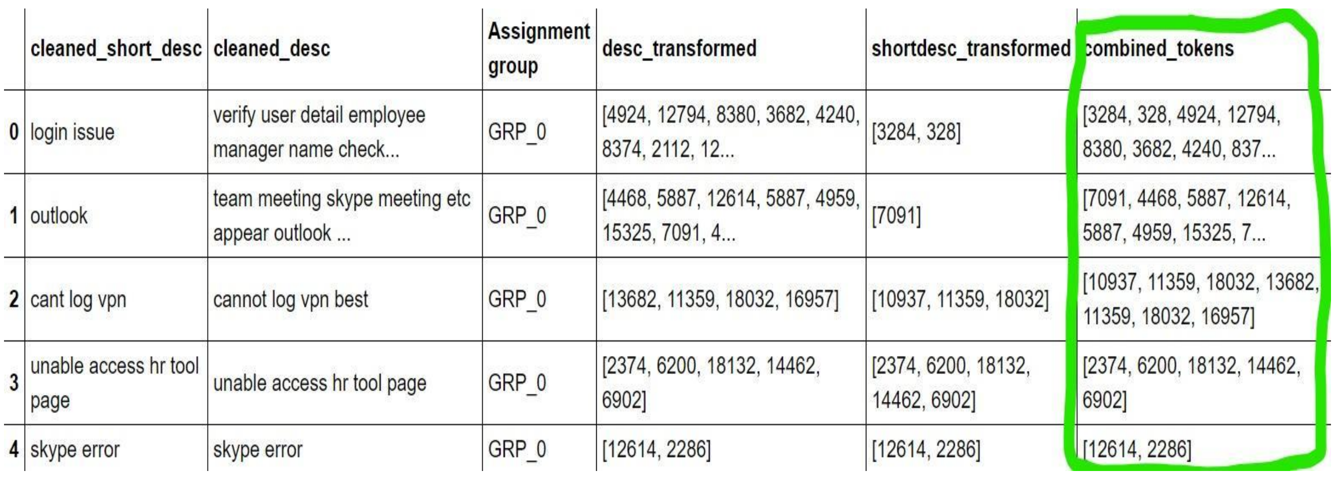
	Tf-Idf					
	Accuracy		**Macro F1-Score**		**Macro ROC_AUC**	
Model	**Train**	**Test**	**Train**	**Test**	**train**	**test**
LinearSVC	0. 90329	0.67351	0.9056	0.52115	0.99885	0.93253
Logistic Regression	0.6915	0.6306	0.31357	0.22749	0.97658	0.9145
Random Forest	0.9572	0.6592	0.95263	0.46348	0.99944	0.92269

3.2. STARTING WITH BASE APPROACH – 2

We load the upsampled data from the previous approach. This upsampled data already has the 'combined tokens' column which we created in approach 1, however we will drop this column and combine again in a slightly different way. Earlier we simply merged the short description and description column since we had to make BOW and Tf-Idf matrices but now since we will be making using of **"sequences"** we can't have repetitive sequences one after another. We have a lot of records where short description and description are exactly same, hence if they are exactly same we don't want to club them and unnecessarily increase the length of our sequences. Therefore if description and short description are **exactly the same we don’t club them, otherwise we do.**

3.2.1. Creating Vocabulary

We create a list under the name of ***vocabulary*** and store every unique word from the corpus that we have. Next we ***convert the vocabulary to numerical values*** and create two dictionaries; mapping the words to the numbers and numbers back to the words. Using these dictionaries we transform both the description and short description columns to numerical values and save them in ***desc_transformned*** and ***short_desc_transformned*** columns.
Finally as explained above we can combine the columns. For this we create a function that returns only desc_transformed if both desc_transformed and short _desc_transformed are same else returns the clubbed tokens. We apply this function to each row thereby creating a new revised combine_tokens.



	cleaned_short_desc	cleaned_desc	Assignment group	desc_transformed	shortdesc_transformed	combined_tokens
0	login issue	verify user detail employee manager name check...	GRP_0	[4924, 12794, 8380, 3682, 4240, 8374, 2112, 12...	[3284, 328]	[3284, 328, 4924, 12794, 8380, 3682, 4240, 837...
1	outlook	team meeting skype meeting etc appear outlook ...	GRP_0	[4468, 5887, 12614, 5887, 4959, 15325, 7091, 4...	[7091]	[7091, 4468, 5887, 12614, 5887, 4959, 15325, 7...
2	cant log vpn	cannot log vpn best	GRP_0	[13682, 11359, 18032, 16957]	[10937, 11359, 18032]	[10937, 11359, 18032, 13682, 11359, 18032, 16957]
3	unable access hr tool page	unable access hr tool page	GRP_0	[2374, 6200, 18132, 14462, 6902]	[2374, 6200, 18132, 14462, 6902]	[2374, 6200, 18132, 14462, 6902]
4	skype error	skype error	GRP_0	[12614, 2286]	[12614, 2286]	[12614, 2286]

3.2.2. Transforming target variable

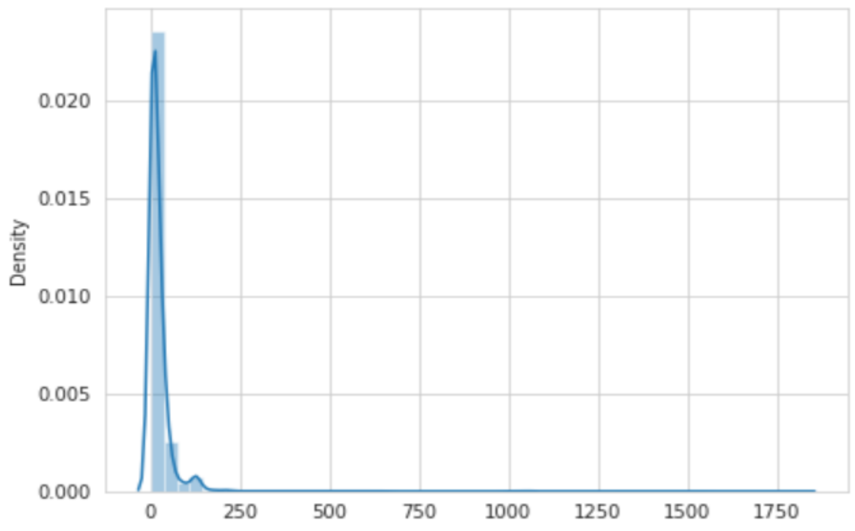
We converted the target variable (assignment group column) into one-hot binarizer form i.e. for every record we have 1 at the group it belongs to otherwise 0. In other words we have all 0's except one 1 (hot) at the group (class) that the particular record belongs to. After transforming the target variable the array will be of shape (8040,74) since we have 75 total classes in the target variable.

3.2.3. Transforming tokens to sequences

Finally we converted the combined_tokens column to sequences. Note that we have different lengths of sequences which cannot be used in the model. The model inputs are always of same length hence we are required to pad the sequences. On some investigation we found that:
95% of the sequence have 76.05 or below 76.05 words whilst 90% of the sequences are 46 and below 46 words. Hence 76 seems to be a fair choice for padding.

Padding sequences to 76 means that sequences having greater than 76 words will be trimmed down to 76 and sequences having less than 76 words will be padded to be 76**,** thereby making all the sequences of the same size.

```
Max no. of tokens in short description (sequence):  1820
Avg. no. of tokens in short description (sequence):  25.32
90% of the sequences are <= 46.0
95% of the sequences are <= 76.05
```



While making the vocabulary we added a ‘DUMMYWORD’ in the vocabulary list which is encoded by the numerical value 19234. We will pad with this value. This means the words that are padded to a sequence will be ‘DUMMYWORD’

Also we will do pre-padding. This is because LSTMs are temporal i.e. it has timestamps. Hence words coming at the later timestamps are retained more in the 'memory' as compared to the words at the beginning of the sequence. Hence we want the dummy words in the beginning of the sequence instead at the end. After padding the sequence array will be of shape (8040,76)

3.2.4. Test and Train split

Again the same 80:20 split which resulted in x_train and y_train of shape **(**6432, 76) and (6432, 74) respectively and x_test and y_test of shape (1608, 76) and (1608, 74) respectively.

3.2.5. Trying base LSTM

Before moving on to training, we first created a pre-trained GloVe embedding matrix. For base model, we tried both **pre-trained glove embeddings** and **tensorflow's default embedding training** that tensorflow trains during the training process.

As for the GloVe embedding we have used pre-trained 200 dimensional embedding. The embedding matrix that we created will be used as the weights of the embedding layer in the LSTM model.

3.2.6. Training

As we said above that we tried a simple architecture of the LSTM model in this base approach. We trained two models; **one with default tensorflow embedding layer** that will be trained from scratch in the training process only and **other with pre-trained GloVe embedding layer with trainable = True** in order to customize the pre-trained embedding for our corpus

The model architecture is as follows for both the base models (except that 1 model has pre-trained embedding and other has default embedding training):

- input layer of size 76
- embedding layer with input_dim = size of the vocabulary, output_dim = 200, input length = size of input layer i.e. 76, weights = GloVeEmbedding_matrix and trainable=True **(applicable when using pre-trained embedding)**
- LSTM layer with 100 units and return sequence = False
- Output dense layer with units = # labels in the target

Being a base model approach, we just trained the model for 10 epochs. We didn't use any regularization yet which, as expected resulted in overfitting.

3.2.7. Performance

MODEL1A – Tensorflow Embedding trained from scratch					
Accuracy		**Macro F1-Score**		**Macro ROC_AUC**	
Train	**Test**	**Train**	**Test**	**train**	**test**
0.9704	0.63805	0.9426	0.4634	0.9993	0.8938

MODEL1B – GloVe Embedding (trainable)					
Accuracy		**Macro F1-Score**		**Macro ROC_AUC**	
Train	**Test**	**Train**	**Test**	**train**	**test**
0.9777	0.6666	0.9625	0.5134	0.9998	0.9247

3.3. CONCLUDING BOTH APPROACHES

Comparing models from both the approaches (1 and 2), we came to a conclusion that LSTMs standout. Even with a very basic architecture they gave almost similar results to Base approach 1. With LSTMs we will have more freedom in choosing the architecture and tuning the performance as opposed to traditional models from base approach 1. A lot can still be done in LSTM model as opposed to traditional ML models in approach 1

Within LSTMs also, pre-trained GloVe embedding outshined the tensorflow's default embedding that was trained from scratch during the training process.

4. How to improve model performance

We will move on to the next part and improve upon this existing architecture of the lstm model and tune the hyper-parameters. We choose LSTM with pre-trained GloVe embedding as the model of our choice which we will carry on with from here.

We will extend the architecture of the model:

- Add more LSTM units
- Maybe use 2 LSTM layers instead of 1
- Introduce regularization such as dropouts as well as recurrent dropouts
- Use return sequences
- Introduce Bi-Directional layer

*****We will explain all these concepts in depth while implementing them.***

We will train the model for more epochs:

- Implement callbacks such as Early Stopping, ReduceLrOnPlateau and Model Checkpoints to save the best model and solve for overfitting
- Train with cross-validation and try to find a better learning rate and optimizer

5. Model Evaluation and Tuning

This is our plan for tuning. **First we will improve upon the architecture (part 1):**

- turn on return sequences
- add some regularization such as dropouts
- introduce bi-directional layer
- add more LSTM units

After deciding the architecture we will tune for better hyper-parameters (part 2):

- try a different optimizer
- try to hit the right learning rate
- implement callbacks in order to train for more epochs and account for overfitting

Improve upon the short comings (part 3):

- We will figure out where and what we fall short of in an attempt to improve the score and generalize the model

5.1. Part 1 Tuning:

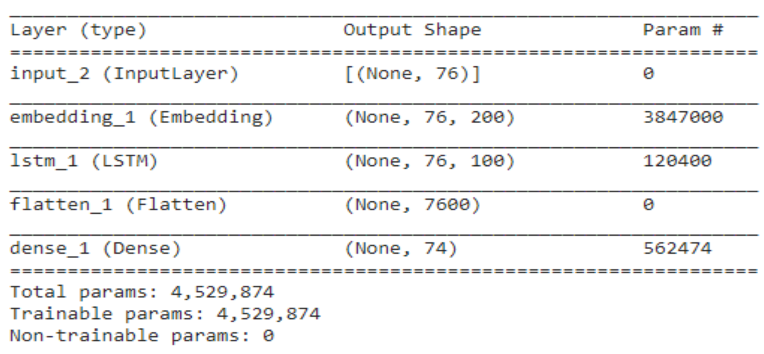
We begin the tuning of model with different phases on it. The training is done for 15 epochs so we can monitor any improvements on it. In Part 1, we are going to find an optimum model architecture finalise it for the part 2 tuning.

5.1.1. Phase 1

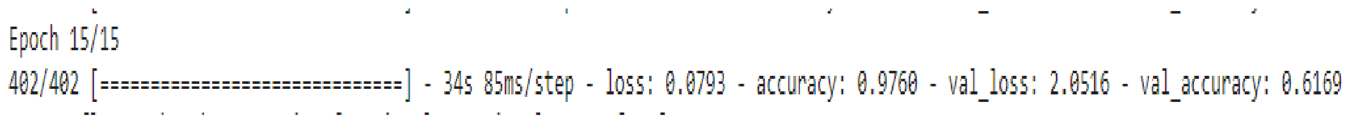
Turning on return_sequences. When we configure return_sequences = True, the model returns the lstm's hidden state at every timestamp (at every word of the sequence). We have 76 words in every sequence so the Lstm layer output shape would be (None,76,100) as opposed to (None,100) when return_sequences = False. To maintain the shape integrity for the output layer we will need to flatten the input coming from lstm layer before passing to the output layer. **This can be done in two ways:** Either we can use the **dense layer** or a **pool layer** to flatten it; whichever gives the best result. **Theoretically, with return_sequences = true, the output of the hidden state is used as an input to the next LSTM layer at every timestamp**

This will be our model 2 which is an improvement over model 1 from base approach 2. First we will flatten with dense layer:

Model 2:



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 76)]              0
_________________________________________________________________
embedding_1 (Embedding)      (None, 76, 200)           3847000
_________________________________________________________________
lstm_1 (LSTM)                (None, 76, 100)           120400
_________________________________________________________________
flatten_1 (Flatten)          (None, 7600)              0
_________________________________________________________________
dense_1 (Dense)              (None, 74)                562474
=================================================================
Total params: 4,529,874
Trainable params: 4,529,874
Non-trainable params: 0
```



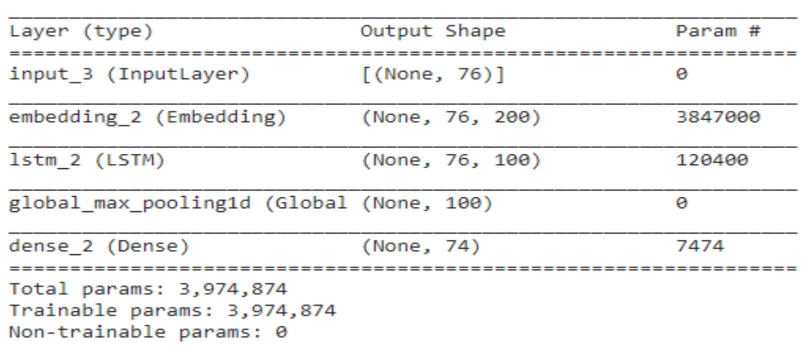
```
Epoch 15/15
402/402 [==============================] - 34s 85ms/step - loss: 0.0793 - accuracy: 0.9760 - val_loss: 2.0516 - val_accuracy: 0.6169
```

Remarks: After training the model with this set of architecture we notice that there has been a considerable improvement in the validation accuracy (from .59 to .61) with the **dense layer**.

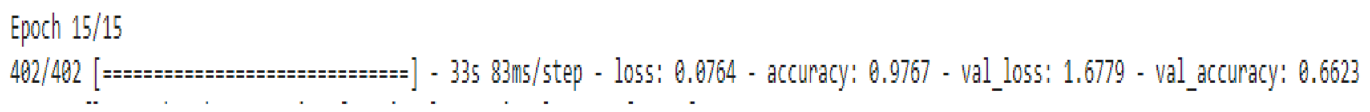
5.1.2. Phase 2

Flattening with **globalMaxPooling layer**.

Model 3:



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 76)]              0
_________________________________________________________________
embedding_2 (Embedding)      (None, 76, 200)           3847000
_________________________________________________________________
lstm_2 (LSTM)                (None, 76, 100)           120400
_________________________________________________________________
global_max_pooling1d (Global (None, 100)               0
_________________________________________________________________
dense_2 (Dense)              (None, 74)                7474
=================================================================
Total params: 3,974,874
Trainable params: 3,974,874
Non-trainable params: 0
```



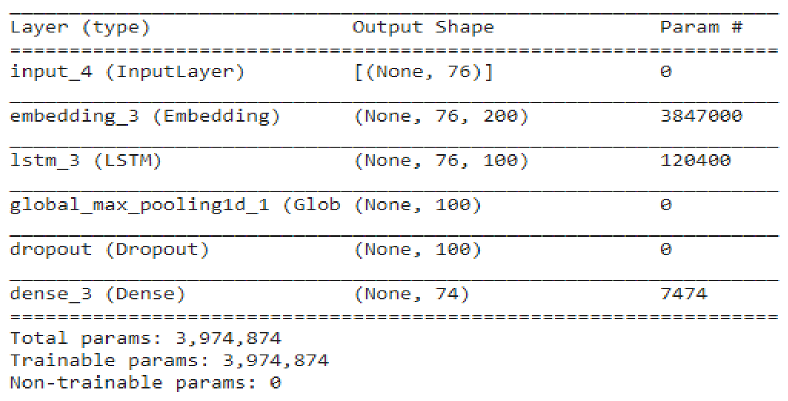
```
Epoch 15/15
402/402 [==============================] - 33s 83ms/step - loss: 0.0764 - accuracy: 0.9767 - val_loss: 1.6779 - val_accuracy: 0.6623
```

Remarks: From the above image we can notice the validation accuracy has increased much higher than the previous model with the dense pooling layer. **GlobaMaxpooling** gives us much higher accuracy and it is much faster than dense layer. So we are moving ahead with the pooling layer.

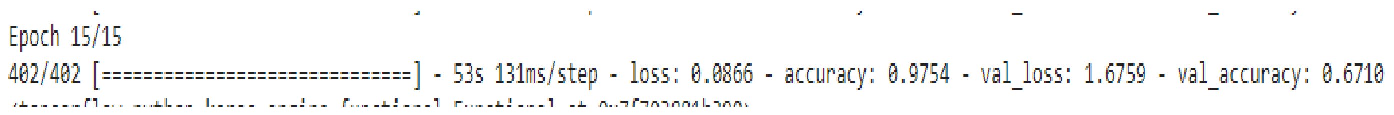
5.1.3. Phase 3

Next we go ahead with ***Regularization***; we are adding a ***dropout layer*** after pooling with the initial ***10%***

Model 4:



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_4 (InputLayer)         [(None, 76)]              0         
_________________________________________________________________
embedding_3 (Embedding)      (None, 76, 200)           3847000   
_________________________________________________________________
lstm_3 (LSTM)                (None, 76, 100)           120400    
_________________________________________________________________
global_max_pooling1d_1 (Glob (None, 100)               0         
_________________________________________________________________
dropout (Dropout)            (None, 100)               0         
_________________________________________________________________
dense_3 (Dense)              (None, 74)                7474      
=================================================================
Total params: 3,974,874
Trainable params: 3,974,874
Non-trainable params: 0
```

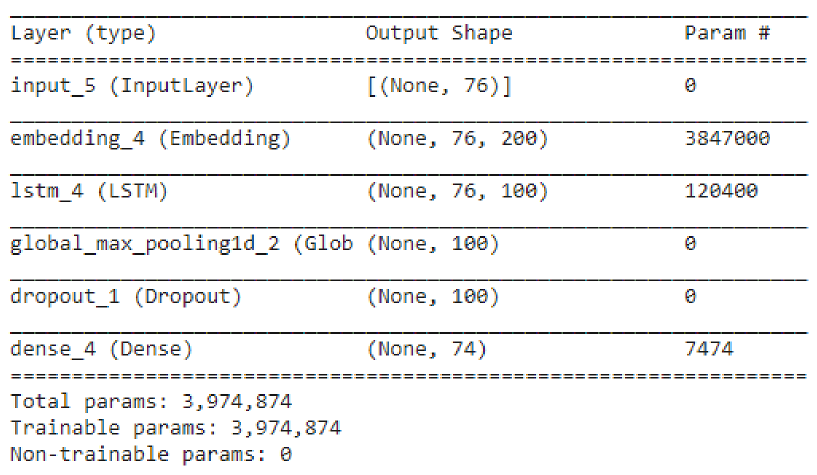


```
Epoch 15/15
402/402 [==============================] - 53s 131ms/step - loss: 0.0866 - accuracy: 0.9754 - val_loss: 1.6759 - val_accuracy: 0.6710
```

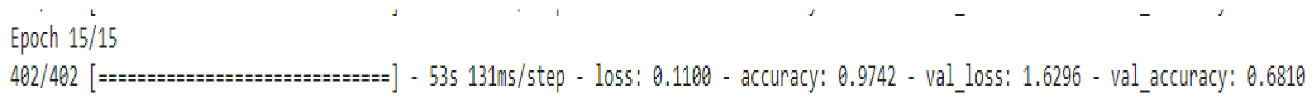
5.1.4. Phase 4:

From the above image we can realize adding a dropout layer increases the validation accuracy, so we increase the ***dropout rate to 20%*** and check for the outcome.

Model 5:



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_5 (InputLayer)         [(None, 76)]              0         
_________________________________________________________________
embedding_4 (Embedding)      (None, 76, 200)           3847000   
_________________________________________________________________
lstm_4 (LSTM)                (None, 76, 100)           120400    
_________________________________________________________________
global_max_pooling1d_2 (Glob (None, 100)               0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 100)               0         
_________________________________________________________________
dense_4 (Dense)              (None, 74)                7474      
=================================================================
Total params: 3,974,874
Trainable params: 3,974,874
Non-trainable params: 0
```



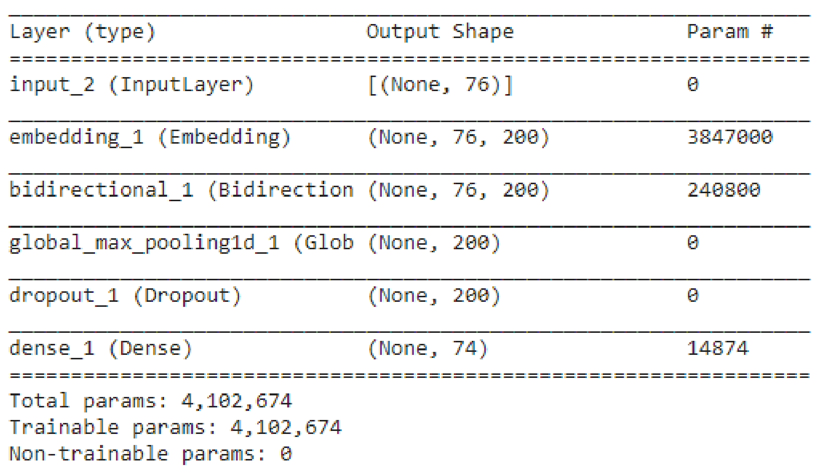
```
Epoch 15/15
402/402 [==============================] - 53s 131ms/step - loss: 0.1100 - accuracy: 0.9742 - val_loss: 1.6296 - val_accuracy: 0.6810
```

Remarks: Since the model with dropout layer with drop rate of 20% gives a slightly higher accuracy. We will proceed with this model

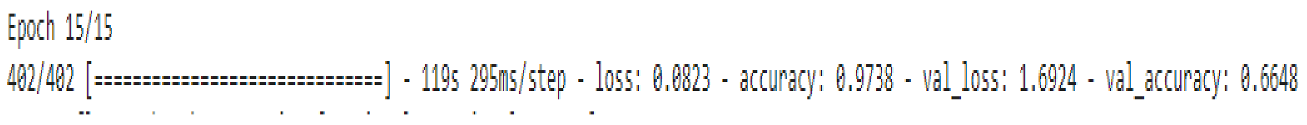
5.1.5. Phase 5:

We proceed with the next step with the model 5 with adding a ***bi-directional layer***. Bi-directional duplicates the first lstm layer in the network and puts the two layers side-by-side. The input sequence is fed as it is to the first layer while reversed copy of the input is fed to the second layer. In simpler words the **bi-directional layer computes the inputs in two ways: past to future and future to past.** The idea behind bi-directional layer is to have better context while computing the word at any given timestamp. Due to computation of sequence and reverse sequence, at any given timestamp, lstm will have the context not only from the previous words but also from the words that are yet to come in future in thatsequence. **Now since we will add a bi-directional layer on lstm with 100 units, the total units will be 200 in our case**

Model 6:



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 76)]              0
_________________________________________________________________
embedding_1 (Embedding)      (None, 76, 200)           3847000
_________________________________________________________________
bidirectional_1 (Bidirection (None, 76, 200)           240800
_________________________________________________________________
global_max_pooling1d_1 (Glob (None, 200)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 200)               0
_________________________________________________________________
dense_1 (Dense)              (None, 74)                14874
=================================================================
Total params: 4,102,674
Trainable params: 4,102,674
Non-trainable params: 0
```



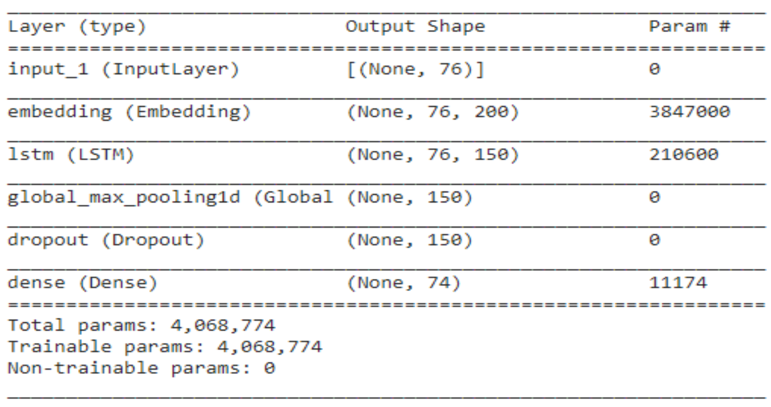
```
Epoch 15/15
402/402 [==============================] - 119s 295ms/step - loss: 0.0823 - accuracy: 0.9738 - val_loss: 1.6924 - val_accuracy: 0.6648
```

Remarks: This model gives us similar results as our previous model (*a little lower*) but since we have added one more LSTM layer it takes double the time for its computation. So we do not add this bi-directional layer to our model.

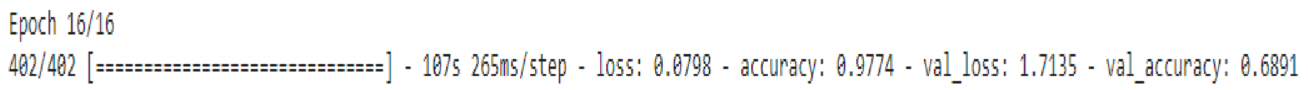
5.1.6. Phase 6

Continuing with model 5 phase 4, we will now add more ***LSTM units*** to this architecture. We increase the units from 100 to 150.

Model 7:



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         [(None, 76)]              0         
_________________________________________________________________
embedding (Embedding)        (None, 76, 200)           3847000   
_________________________________________________________________
lstm (LSTM)                  (None, 76, 150)           210600    
_________________________________________________________________
global_max_pooling1d (Global (None, 150)               0         
_________________________________________________________________
dropout (Dropout)            (None, 150)               0         
_________________________________________________________________
dense (Dense)                (None, 74)                11174     
=================================================================
Total params: 4,068,774
Trainable params: 4,068,774
Non-trainable params: 0
_________________________________________________________________
```



```
Epoch 16/16
402/402 [==============================] - 107s 265ms/step - loss: 0.0798 - accuracy: 0.9774 - val_loss: 1.7135 - val_accuracy: 0.6891
```

Remarks: Again, not much performance difference from model 5, but training time increased 2 times**.**

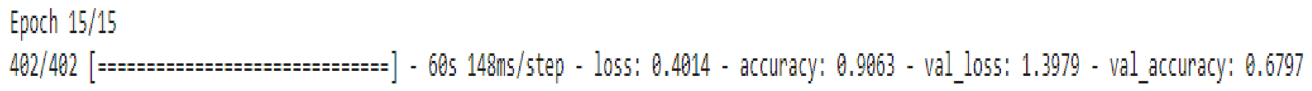
We have covered all the steps from deciding the architecture. **We select model 5 from phase 4 to be the apt architecture, which we will take forward to part 2**

5.2. Part 2 Tuning

In tuning part 2 we will try a different optimizer, tune the learning rate and go for longer training session with various callbacks in order to avoid overfitting

5.2.1. Phase 1

In phase 1 we will be trying a different optimizer. We have been using Adam as an optimizer which was the default optimizer. In this phase we will be implementing it with RMSProp. Model architecture is model 5 from phase 4 as we have stated.



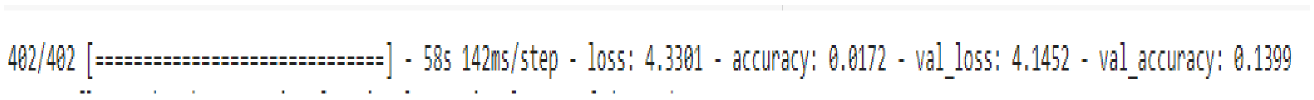
```
Epoch 15/15
402/402 [==============================] - 60s 148ms/step - loss: 0.4014 - accuracy: 0.9063 - val_loss: 1.3979 - val_accuracy: 0.6797
```

Remarks: There is not much change in the outcome if we change the optimizer so we will be proceeding with Adam.

5.2.2. Phase 2

In this part we will work on finding the optimum learning rate but first do some sanity checks for the model architecture

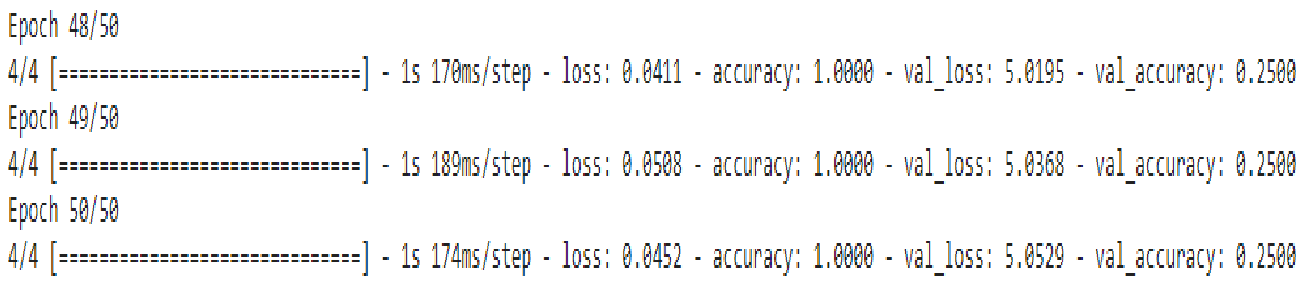
Sanity Check 1:
We start by training just one epoch with a very low learning rate of 1 e-5.



```
402/402 [==============================] - 58s 142ms/step - loss: 4.3301 - accuracy: 0.0172 - val_loss: 4.1452 - val_accuracy: 0.1399
```

We have 74 classes, hence because of random initilisation of weights, probability of prediction of each class is 1/74 i.e 0.0134. So for any class, log loss/crossentropy loss would be -log_e(0.0134) = 4.31. **We get approximately the same loss hence we pass the first check.**

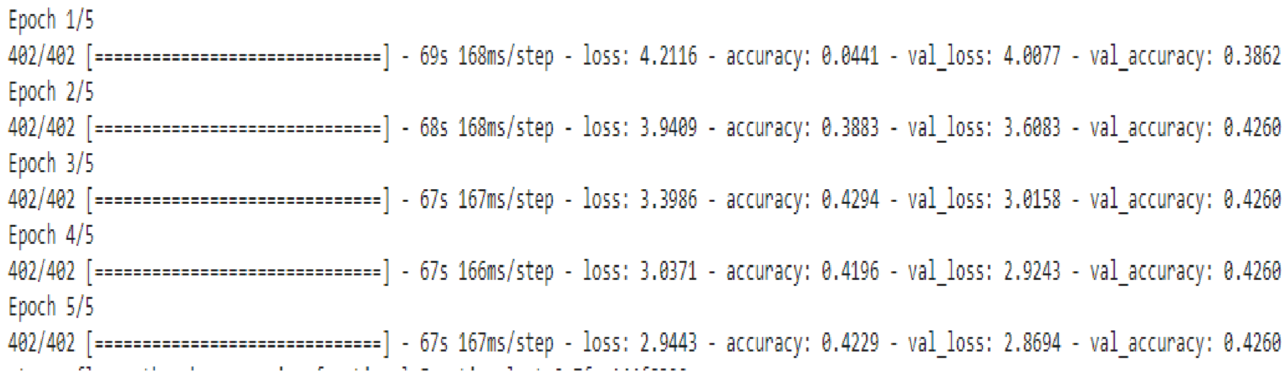
Sanity Check 2:
We will now try to overfit the model on a small subset of data. We'll just take 20 records from the data



```
Epoch 48/50
4/4 [==============================] - 1s 170ms/step - loss: 0.0411 - accuracy: 1.0000 - val_loss: 5.0195 - val_accuracy: 0.2500
Epoch 49/50
4/4 [==============================] - 1s 189ms/step - loss: 0.0508 - accuracy: 1.0000 - val_loss: 5.0368 - val_accuracy: 0.2500
Epoch 50/50
4/4 [==============================] - 1s 174ms/step - loss: 0.0452 - accuracy: 1.0000 - val_loss: 5.0529 - val_accuracy: 0.2500
```

From the above image we can find that the model is completely able to overfit the data thereby passing the second sanity check too. Hence our model architecture looks good.

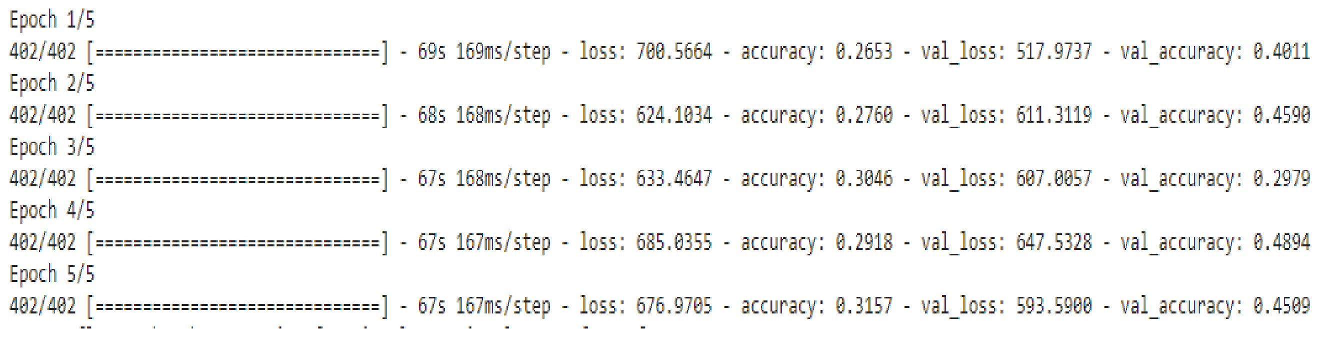
Lets explore the learning rate here for a few epochs, say 5. We'll start with a very small learning rate; 1e-5



```
Epoch 1/5
402/402 [==============================] - 69s 168ms/step - loss: 4.2116 - accuracy: 0.0441 - val_loss: 4.0077 - val_accuracy: 0.3862
Epoch 2/5
402/402 [==============================] - 68s 168ms/step - loss: 3.9409 - accuracy: 0.3883 - val_loss: 3.6083 - val_accuracy: 0.4260
Epoch 3/5
402/402 [==============================] - 67s 167ms/step - loss: 3.3986 - accuracy: 0.4294 - val_loss: 3.0158 - val_accuracy: 0.4260
Epoch 4/5
402/402 [==============================] - 67s 166ms/step - loss: 3.0371 - accuracy: 0.4196 - val_loss: 2.9243 - val_accuracy: 0.4260
Epoch 5/5
402/402 [==============================] - 67s 167ms/step - loss: 2.9443 - accuracy: 0.4229 - val_loss: 2.8694 - val_accuracy: 0.4260
```

Learning rate is indeed too low. No training whatsoever.

Let’s take a big learning rate, say 10. Training for the same number of epochs as before.



```
Epoch 1/5
402/402 [==============================] - 69s 169ms/step - loss: 700.5664 - accuracy: 0.2653 - val_loss: 517.9737 - val_accuracy: 0.4011
Epoch 2/5
402/402 [==============================] - 68s 168ms/step - loss: 624.1034 - accuracy: 0.2760 - val_loss: 611.3119 - val_accuracy: 0.4590
Epoch 3/5
402/402 [==============================] - 67s 168ms/step - loss: 633.4647 - accuracy: 0.3046 - val_loss: 607.0057 - val_accuracy: 0.2979
Epoch 4/5
402/402 [==============================] - 67s 167ms/step - loss: 685.0355 - accuracy: 0.2918 - val_loss: 647.5328 - val_accuracy: 0.4894
Epoch 5/5
402/402 [==============================] - 67s 167ms/step - loss: 676.9705 - accuracy: 0.3157 - val_loss: 593.5900 - val_accuracy: 0.4509
```

As expected the loss is too high but we got the window in which we can tune the learning rate; 1e-5 to 1e1 (10). Using a cross validation loop, we randomly selected learning rate 10 times from within this range. Each iteration is listed below:

Learning Rate	Train		Test	
	Loss	Accuracy	Loss	Accuracy
0.0016088962399883817	0.11538372933864594	0.974191546440124	1.5570141077041626	0.6548507213592529
0.0016260926005471018	0.04329139739274978	0.984141767024993	1.8360795974731445	0.6735074520111084
0.00010021318937978307	0.03587121516466141	0.985852003097534	1.9167197942733765	0.6735074520111084
0.05293836873733082	1.61825191974639	0.649253726005554	2.4460535049438477	0.5764925479888916]
9.505935946411968	120.46714782714844	0.512126863002777	162.6533966064453	0.47450247406959534
0.0012472552940708005	109.37472534179688	0.525963902473449	150.99366760253906	0.4856965243816376
7.222623073525155e-05	108.88406372070312	0.526741266250610	150.47802734375	0.4856965243816376
0.00033438146002369174	106.56829833984375	0.529850721359252	148.23422241210938	0.4894278645515442
0.02211490766201775	65.55039978027344	0.579757452011108	107.9630355834961	0.503731369972229
2.097970618547511	29.79146957397461	0.588463902473449	59.17458724975586	0.48009949922561646

We can notice from the above table that the learning rate of 0.00162 (2nd iteration) works best which is close to the default learning rate (1e-3) of adam

5.2.3. Phase 3

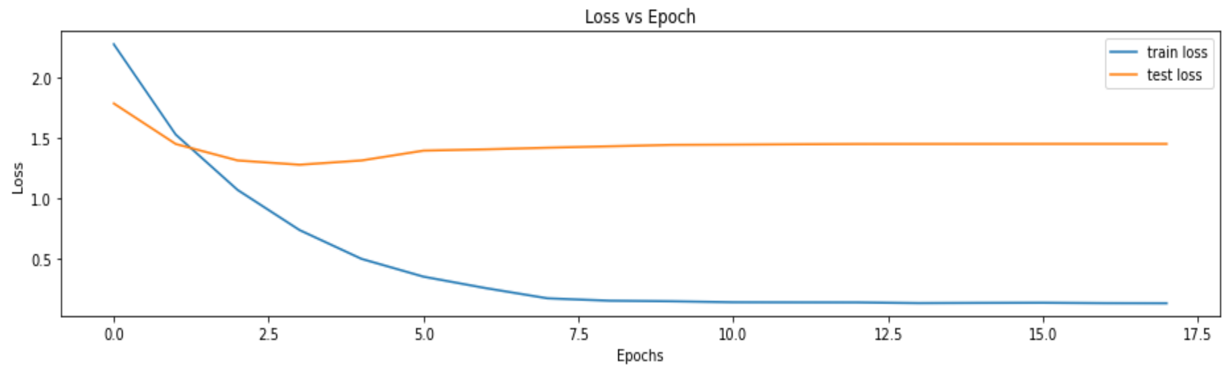
Finally, we can now go for the longer training session with ReduceLROnPlateau, EarlyStopping and ModelCheckpoint. **The callback parameters are explained below:**

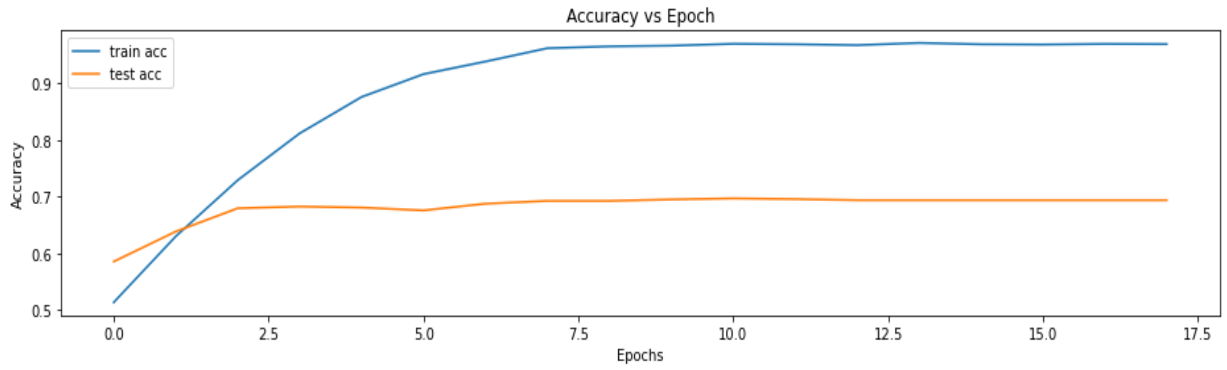
- We will start with the **learning rate of 0.00162** and reduce it by **10% or 0.1 (factor)** in the training process itself if **validation loss (monitor)** doesnt decrease by **0.01 (min_delta)** for **3 epochs (patience)**. We will reduce untill **lr = 1e-7 (min_lr)**. This is our **ReduceLROnPlateau**

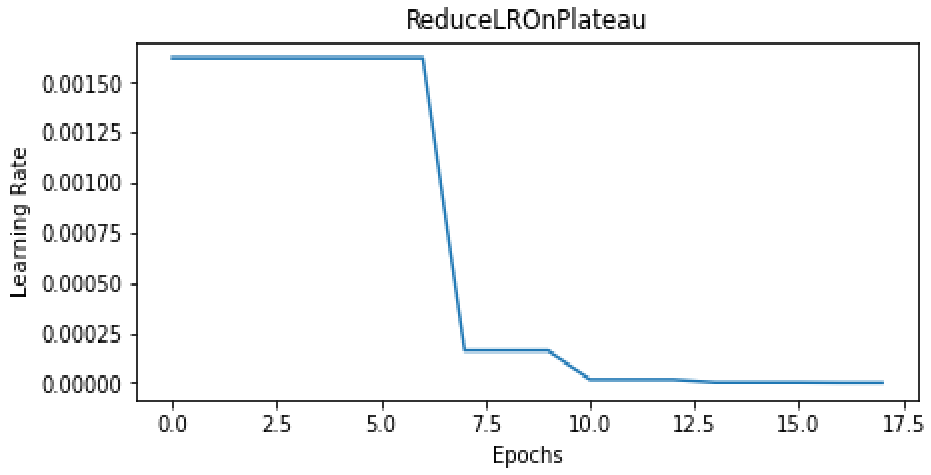
- To avoid overfitting we will stop the training process earlier. If the **loss(monitor)** does not decrease by **0.1 (min_delta)** for **10 epochs (patience)**, we will stop the training and **save the weights from best epoch (restore_best_weights)**. This is our **EarlyStopping**

- We will save the model and its weights every time the **validation accuracy (monitor)** is improved. The previous model will be **over-written by new one with better validation accuracy (save_best_only)**. We will save the model and its weights with name **model5 (chkpnt_filename)** in .h5 format. Model at the epoch with the best validation accuracy will be saved by the end of the training. This is our **ModelCheckpoint**

Below is our training graphs and ReduceLrOnPlateau graph:







Let’s look at the evaluation metrics that we have been using until now:

```
Train set performance:
Accuracy: 0.9752798507462687
(macro) ROC-AUC: 0.999779393903313
(macro) F1 Score: 0.9575847723733336

Test set performance:
Accuracy: 0.697139303482587
(macro) ROC-AUC: 0.942417674736818
(macro) F1 Score: 0.5388389768286452

**As explained in part2a - micro-metrics will be similar to accuracy
```

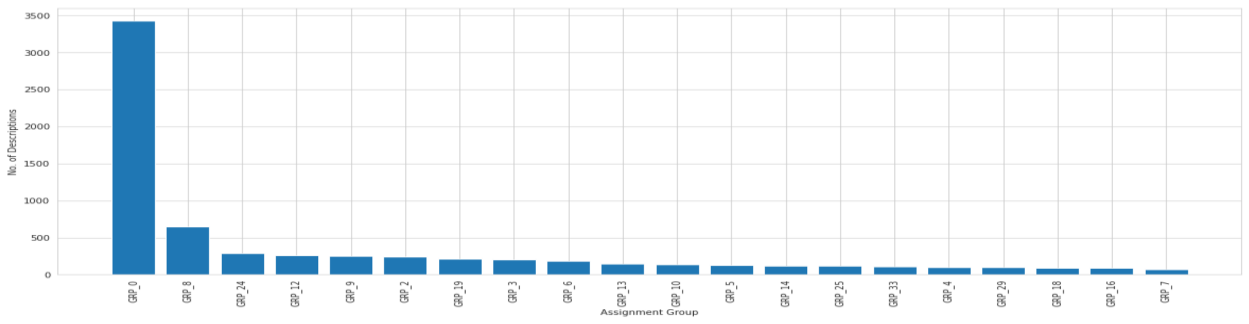
So on TEST SET the best accuracy that we got is 69.71 %. This is also equivalent to all the Micro Metrics. We said that Micro is the overall performance of the model despite having imbalanced classification. Macro on the other hand reflects the effect of imbalance in classes. Macro f1-score is 53.88 % while One VS One AUC is 94% on the TEST SET.

Short-comings: Overall, the performance of the model on test set is 70% despite having class imbalance and lack of data. We know how deep learning is data hungry and training on a dataset of size 6432 records (80% train set) does not provide as good metrics as one would expect but given the dataset we have, 94.24% Macro AUC seems fair. Also, we have seen that the target classes are skewed. There are certain assignment groups that do not have much data to support.

5.3. Part 3 Tuning

In the light of imbalance in target classes we will upsample the minority classes a bit for the model to train from it. We will take the same model 5 from phase 4 ahead in this part 3 along with the learning rate (0.00162) that we found in part 2 tuning.

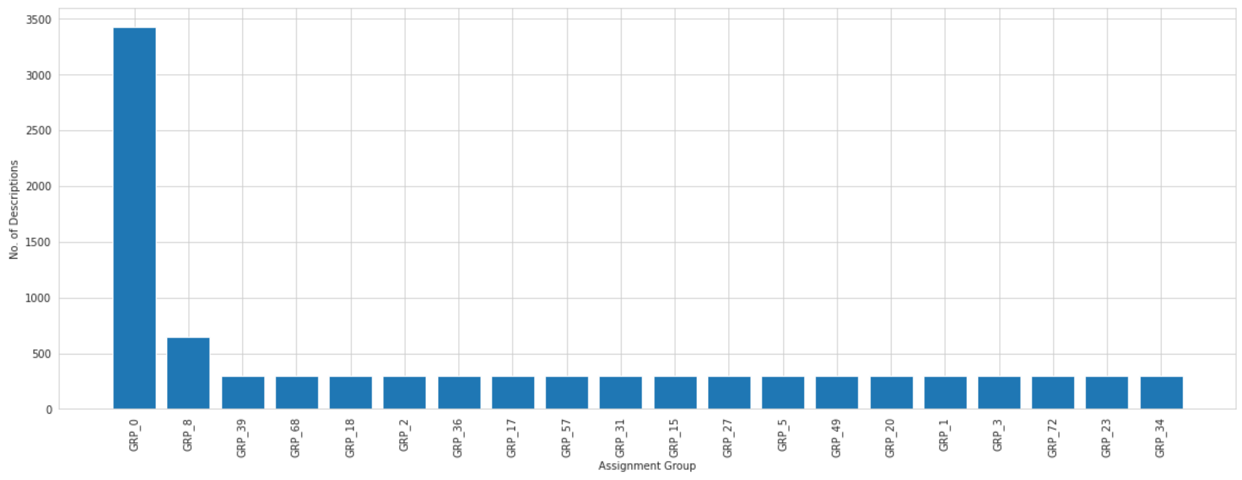
In this part we will upsample the previously upsampled data. If you remember, in Base Approach 1, we slightly upsampled the data and made all the classes having less than 10 records equal to 10. We did that for a different purpose then (refer Base Approach 1 for more details), but here we will upsample to have adequate samples for minority classes so that they could be trained and understood by the model



If we look at the top 5 records, except first 2 groups, all are below 300 and skews even further which we have already seen. We have also seen that most of the groups have around 70-150 records only and some go as low as just 10 records.

The model performs averagely for groups that have 70, 100, 200 records (that we saw in support column of the classification report) and very poorly for records that are below 50. The training is overshadowed by the first few groups below (the top 5), specially by group 0 and group 8
Thus we will upsample all the groups below 300 to 300. This means that from group 24, all the groups will be upsampled

WHAT HAPPENS WHEN WE UPSAMPLE TEXT DATA?
When we upsample the data, it will simply create duplicate records for that particular group in order to increase its sample size and make it recognizable to the model. E.g. Let's say a group has 20 records. During upsampling, a record from these 20 records will be picked and duplicated, and this process repeats until that group has 300 records (which is set by us)
The histogram below shows how our data looks after upsampling.



6. Final Tuned Model

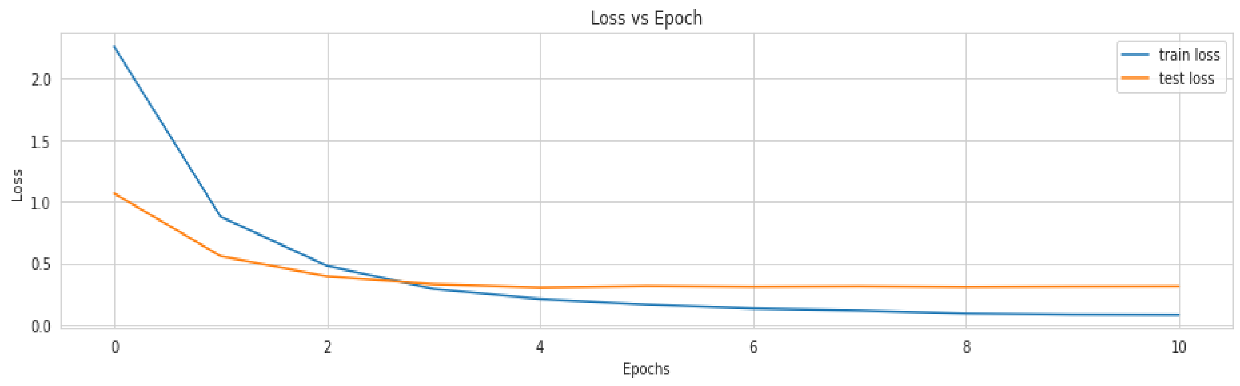
After the upsampling is done we will create a vocabulary that is same as our base model 2 approach. The vocabulary will be created of length 19235. We will map the vocabulary to numerical values and vice versa and convert the dataset to numerical tokens. The target variable is transformed

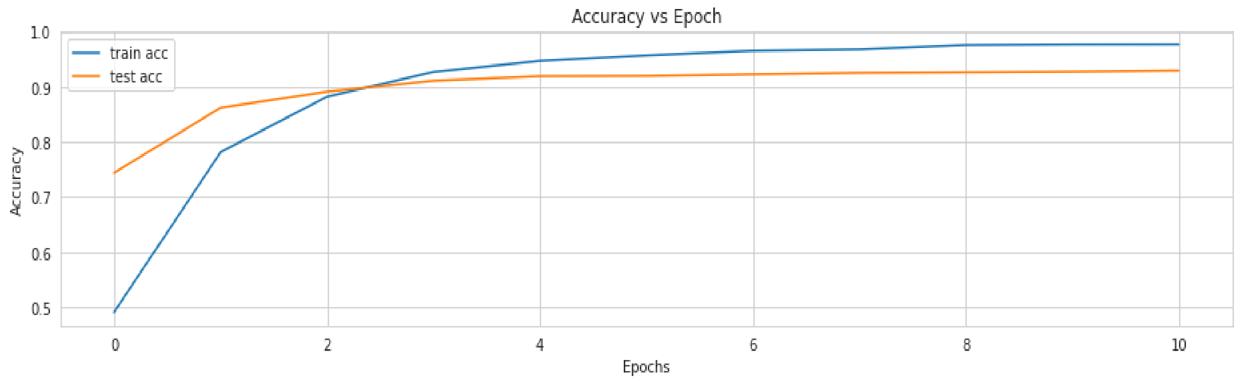
After that the data is split to 70:30 for training and testing. We will be loading the previous glove embedding matrix since the vocabulary is the same but just the upsampling is done.

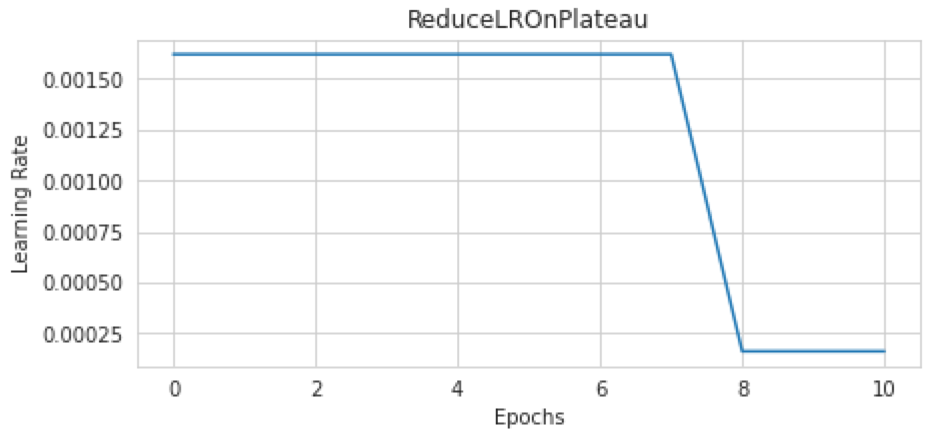
As we said in previous part we will take model 5 forward along with learning rate = 0.00162 which we tuned in the last part. We will go with the same training process again but with new upsampled data. We just have a few minor changes based on our experience from previous training:

- We have reduced the **patience for early stopping to 5 from 10**
- Now that we have more data, we have **increased the batch size to 32 from 16**
- Again, since we have more data, the model will aggressively overfit, hence **we have increased the dropout rate to 40%**
- And finally, **we will monitor validation_loss instead of validation_accuracy to save the model_checkpoint.** The model at epoch with least validation_loss will be saved.

Below is our training graphs and ReduceLrOnPlateau graph







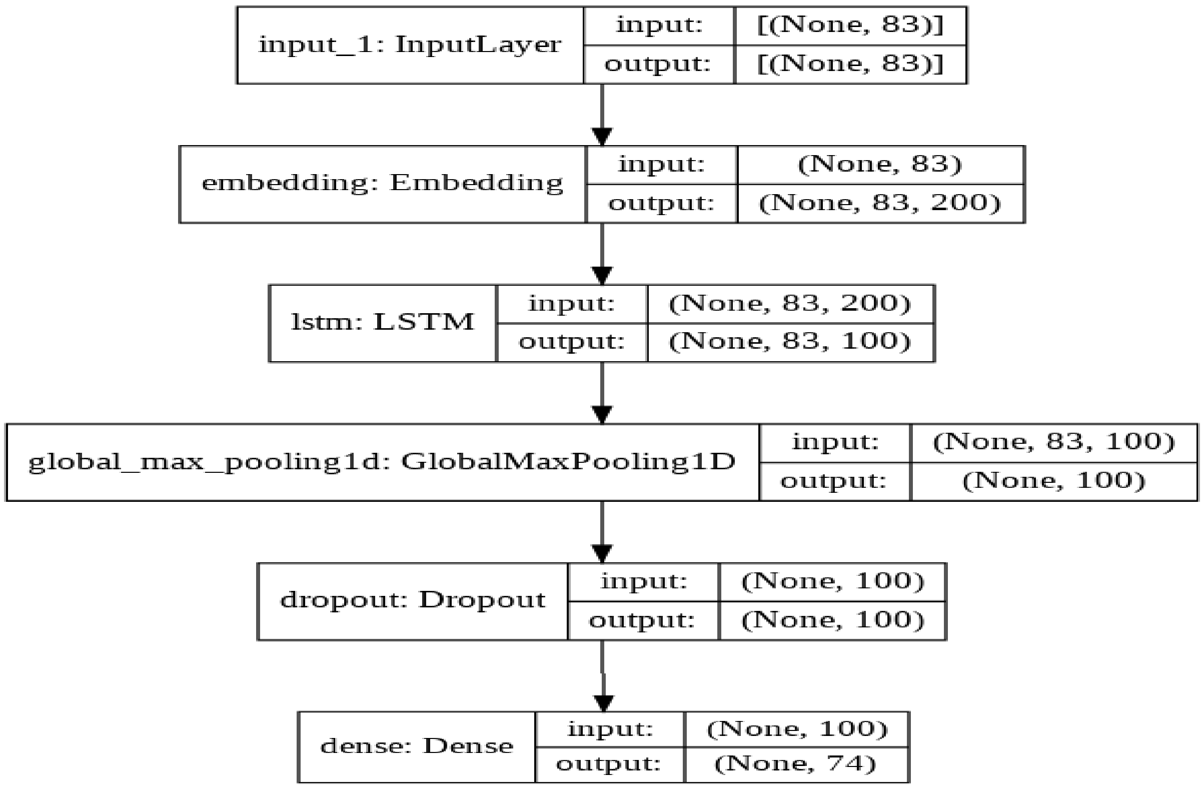
Let's look at the evaluation metrics:

```
Train set performance:
Accuracy: 0.9713983640309387
(macro) ROC-AUC: 0.9997131408720261
(macro) F1 Score: 0.9746532702542502

Test set performance:
Accuracy: 0.9195014282004674
(macro) ROC-AUC: 0.99724265540487
(macro) F1 Score: 0.9317530012493747

**As explained in part2a - micro-metrics will be similar to accuracy
```

We can see massive improvement after upsampling. Test set performance drastically improved from 53.8% macro f1-score in previous part to 93.17% in this part. Test accuracy improved from 69.71% to 91.95% (which is also equal all the micro metrics).

7. Model Architecture

This is our model architecture. We have seen above in detail how we reached this model architecture. This is the same model 5 from model tuning phase 4, except that it has 40% dropout rate instead of 20%



8. Implications

While we have to deal with high volumes of issues arising every day in an organisation, it takes a lot of man power and human interference to assign the issue to the right department/group. This can be both time consuming as well as expensive.

Our model is very well able to handle the most common issues for which the tickets are raised very frequently. The model peforms a great job when dealing with tickets in english language. However our model falls short for the issues that are very rare or in a different language.

9. Limitations

There are certain limitations to this dataset

- The major limitation is that there is less number of records for most of the target variables which gives us a huge disadvantage over the classification.
- Next is a multilingual description in the dataset. Most of the entries in the description and short description columns were of different languages and it gives us a narrow space for forming the word clouds.

10. Closing Reflection

In the research and development of this project we faced certain issues that could be treated to better conceptualise this project starting with collecting more data specifically for under sampled assignment groups for which the tickets are raised very rarely. This would enhance the model performance and generalize it for using it in production.

We could also use language translation libraries to better make use of multi-linguistic nature of the dataset and adapt it to the most common spoken languages. This would increase the scope and practicality of this project.

----------------------------- The End -----------------------------

GitHub Link to the code