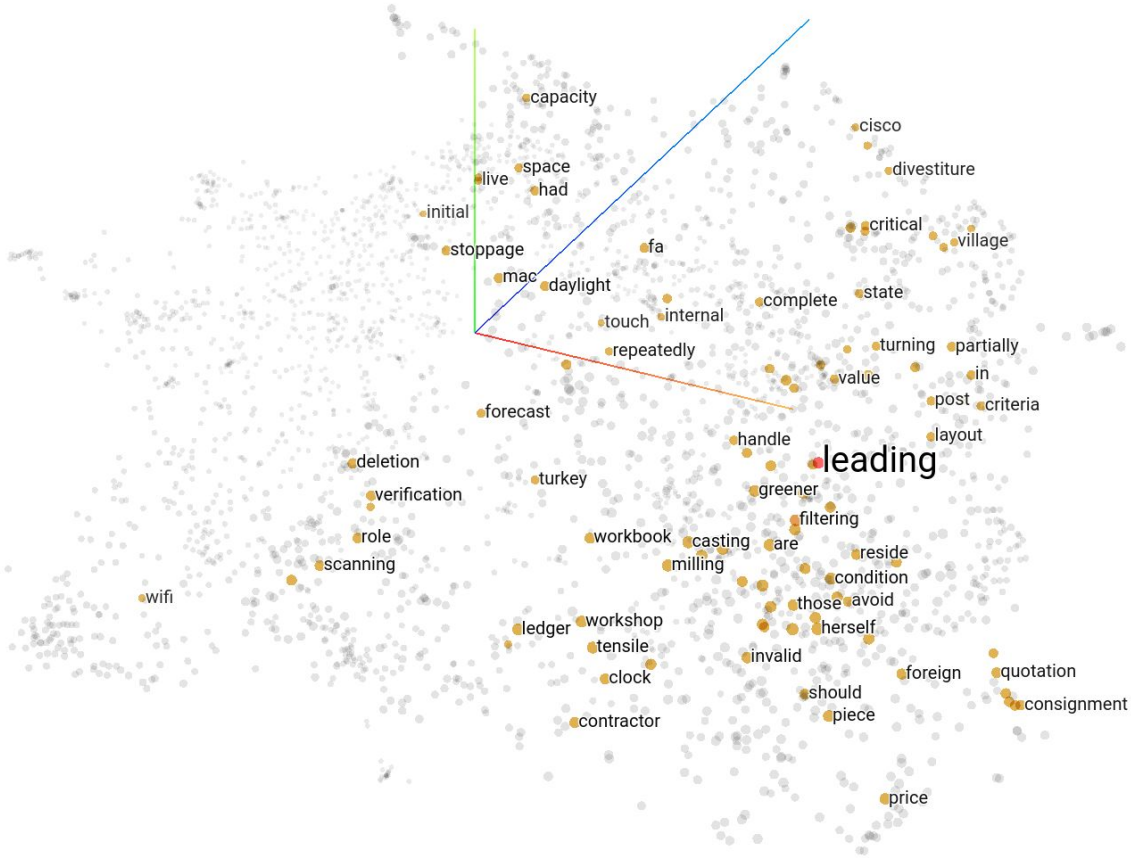


AIML Online Capstone - AUTOMATIC TICKET ASSIGNMENT - Final Report

December 27th, 2020

Vineeth Govind, Sridhar Krishnaswamy
AIML Online Batch Group 4 - Year 2020
Mentor: Sumit Kumar

Table of Contents

Summary of problem statement, data and findings

Incident Management and Response is a key component of any IT Service Management Strategy. These are the typical steps involved in the Incident Management Process:

a. Receipt of the issue
b. Create a ticket
c. Review of the ticket by L1/L2 teams
d. Attempt to resolve the ticket using Standard Operating Procedures by L1/L2
e. If needed, transfer the ticket to the appropriate L3 team for further review and resolving.

Current 'Pain' Points

Currently the organization sees these issues in the Incident Ticket Management Process:

a. The process is largely 'manual'. L1/L2 teams need to spend time to review Standard Operating Procedures (SOPs) before assigning to functional teams. Minimum 25-30% incidents need to be reviewed for SOPs before ticket assignment.

b. **Minimum 1 FTE effort needed only for incident assignment to L3 teams**

c. Human error - many times the incident gets assigned to the wrong L3 team. So additional effort needed to reassign to the correct team after re-review of the ticket, this not only increases the manual effort needed BUT *also leads to customer dis-satisfaction because the customer who opened the ticket is left frustrated because the ticket is in limbo being tossed between various teams before getting to the actual team who can help resolve the issued.*

Objective of this Project

The dataset provided has about 8500 records showing Ticket Long & Short Description, the caller, and the Group to which the Ticket has been assigned to.

Create various Machine Learning Models that can help classify incidents and assign them to the right Functional Group. Our objective is to create NLP models that can predict with at least 80% accuracy.

Milestone - 2 : Where do we go from here (after reaching Milestone-1)?

For the second half of the project assignment, we decided to work on these aspects:

a. Explore building a Neural Network Model from ground up using Continuous Bag of Words
b. Explore building a Neural Network Model from ground up using Word2Vec
c. Explore more visualization techniques
d. Explore Feature Engineering
e. Apply Feature Engineering on a basic MLP or CNN model and see its performance.
f. Explore Model Interpretation techniques
g. Explore these Deep Learning Models & Fine tune the models with Hyper Parameter Tuning:
    i. Attention Encoder-Decoder
    ii. Bi Directional LSTM
    iii. BERT
    iv. GPT2
h. Explore approach for Production Implementation for Real-time calls to an API that can return the Group number the ticket is assigned to.
i. Final Summary

Explore building a Neural Network Model from ground up using Continuous Bag of Words

For details and the complete image, please refer to notebook: capstone_nlp_FeatureEngg.ipynb

1. We first build a word2vec model using a CBOW (Continuous Bag of Words) neural network architecture

```
Vocabulary Size: 4425
```

```
Vocabulary Sample: [('to', 1), ('the', 2), ('from', 3), ('in', 4),
('is', 5), ('please', 6), ('not', 7), ('and', 8),
```

2. We then build the context word generator. Given a list of words the function tries to generate the next possible word.

```
Context (X): ['login', 'issue', 'manager', 'name'] -> Target (Y): user

Context (X): ['issue', 'user', 'name', 'checked'] -> Target (Y): manager

Context (X): ['user', 'manager', 'checked', 'the'] -> Target (Y): name

Context (X): ['manager', 'name', 'the', 'name'] -> Target (Y): checked

Context (X): ['name', 'checked', 'name', 'in'] -> Target (Y): the

Context (X): ['checked', 'the', 'in', 'ad'] -> Target (Y): name

Context (X): ['the', 'name', 'ad', 'and'] -> Target (Y): in

Context (X): ['name', 'in', 'and', 'reset'] -> Target (Y): ad

Context (X): ['in', 'ad', 'reset', 'password'] -> Target (Y): and

Context (X): ['ad', 'and', 'password', 'advised'] -> Target (Y): reset

Context (X): ['and', 'reset', 'advised', 'to'] -> Target (Y): password
```

3. We build a Deep Network Model using Continous Bag of Words Technique:

```
cbow = Sequential()

cbow.add(Embedding(input_dim=vocab_size, output_dim=embed_size,
input_length=window_size*2))

cbow.add(Lambda(lambda x: K.mean(x, axis=1),
output_shape=(embed_size,)))

cbow.add(Dense(vocab_size, activation='softmax'))

cbow.compile(loss='categorical_crossentropy', optimizer='rmsprop')

print(cbow.summary())

Model: "sequential"

_________________________________________________________________

Layer (type)                 Output Shape              Param #

=================================================================

embedding (Embedding)        (None, 4, 100)            442500

_________________________________________________________________
```

```
lambda (Lambda)              (None, 100)               0
_________________________________________________________________
dense (Dense)                (None, 4425)              446925
=================================================================
Total params: 889,425
Trainable params: 889,425
Non-trainable params: 0
```

4. We train this model over 5 epochs.

   Epoch: 1

   Processed 100000 (context, word) pairs

   Processed 200000 (context, word) pairs

   Processed 300000 (context, word) pairs

   Processed 400000 (context, word) pairs

   Epoch: 2

   Processed 100000 (context, word) pairs

   Processed 200000 (context, word) pairs

   Processed 300000 (context, word) pairs

   Processed 400000 (context, word) pairs

   Epoch: 3

   Processed 100000 (context, word) pairs

   Processed 200000 (context, word) pairs

   Processed 300000 (context, word) pairs

   Processed 400000 (context, word) pairs

   Epoch: 4

   Processed 100000 (context, word) pairs

   Processed 200000 (context, word) pairs

   Processed 300000 (context, word) pairs

   Processed 400000 (context, word) pairs

   Epoch: 5

   Processed 100000 (context, word) pairs

   Processed 200000 (context, word) pairs

Processed 300000 (context, word) pairs

Processed 400000 (context, word) pairs

5. We generate the word embeddings.

(3854, 100)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	1
work	2.193774	-1.931448	1.625629	1.707282	1.312286	-1.464201	-3.898633	1.725463	0.882685	4.681824	2.771838	1.401485	-2.929445	-2.438774	2
job	3.994073	-2.398642	3.950107	5.141868	2.856045	-6.167782	-0.391784	-0.473690	5.190012	3.504515	1.902044	4.054296	0.643980	-3.658626	3
tool	1.215631	-1.081742	0.838637	0.691373	0.436138	-1.612272	-1.052779	1.636212	0.396336	1.074898	1.247917	0.508198	-2.122839	-0.916266	1
vendor	1.981678	-1.372047	1.358577	1.254511	1.300622	-1.964491	-1.752083	1.259274	1.157773	2.168831	1.630616	1.689415	-1.613821	-1.626819	1
network	3.339270	-1.331338	2.309378	0.892507	1.519923	-0.652464	-2.494762	2.177704	2.142085	1.505862	1.290429	0.262064	-2.083684	-1.501395	2
power	1.594538	-1.808415	1.521464	0.585430	1.506623	-1.145880	-1.340903	1.576423	1.260696	0.931479	1.118735	1.463444	-1.521951	-0.773487	1
error	1.361299	-1.830052	3.143207	1.816359	2.884147	-1.353626	-3.159086	3.191547	2.025674	1.504273	1.613817	2.510046	-2.504871	-2.143100	1
need	1.392441	-1.641640	1.491902	1.408130	1.442747	-1.639605	-1.691422	1.356411	1.468530	1.395505	1.745522	1.397045	-1.614374	-1.603075	1

6. We then Build a distance matrix to view the most similar words (contextually)

{'lockout': ['frequent', 'town', 'duck', 'filterwithing', 'prognose'],

'network': ['take', 'make', 'hello', 'office', 'problem'],

'password': ['mail', 'meet', 'current', 'forward', 'every'],

'receive': ['tool', 'get', 'need', 'check', 'send'],

'schedule': ['immediately', 'telephony', 'true', 'lot', 'nothing'],

'system': ['go', 'one', 'order', 'error', 'issue'],

'ticket': ['show', 'one', 'set', 'plant', 'customer'],
'work': ['backup', 'need', 'site', 'maint', 'see']}

Explore building a Neural Network Model from ground up using Word2Vec

1. We now implement a word2vec model using a skip-gram neural network architecture. We use the keras tokenizer API

```
tokenizer = text.Tokenizer()
tokenizer.fit_on_texts(mydata['Combined Description Cleaned'])
word2id = tokenizer.word_index
id2word = {v:k for k, v in word2id.items()}
```

2. Some sample skip grams ((word1, word2) -> relevancy)

Vocabulary Size: 3855
Vocabulary Sample: [('receive', 1), ('work', 2), ('job', 3), ('tool', 4), ('vendor', 5), ('network', 6), ('power', 7), ('error', 8), ('need', 9), ('issue', 10)]

(able (49), login (76)) -> 1

(advise (159), issue (10)) -> 1

(issue (10), able (49)) -> 1

(resolve (137), able (49)) -> 1

(password (24), gain (1570)) -> 0

(name (55), able (49)) -> 1

(check (34), space (188)) -> 0

(advise (159), ad (288)) -> 1

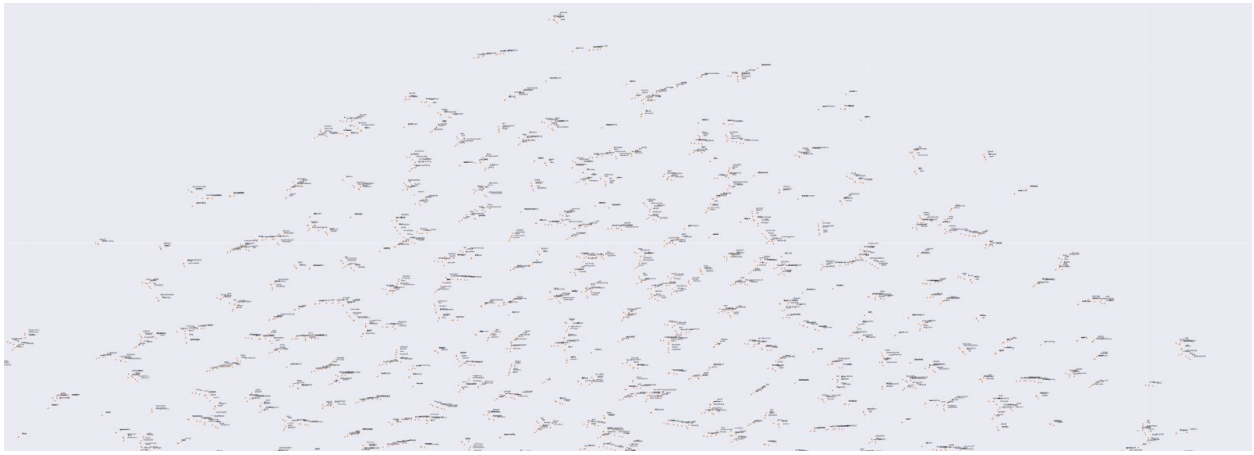
(caller (725), check (34)) -> 1
(issue (10), reg (2080)) -> 0

We also built a word2vec model using gensim

{'issue': ['error', 'problem', 'need', 'unable', 'onesue'],
'job': ['schedule', 'statistic', 'cluster', 'arc', 'abort'],
'login': ['connect', 'password', 'logwithin', 'log', 'relate'],
'manager': ['van', 'would', 'today', 'requirement', 'reply'],
'network': ['na', 'vendor', 'outage', 'power', 'yes'],
'password': ['thim', 'login', 'unlock', 'lock', 'pthessword'],
'schedule': ['job', 'scneverdule', 'arc', 'resume', 'cluster'],
'ticket': ['e', 'secure', 'telephoffy', 'telephouty', 'powder']}

Explore more visualization techniques

The figure below shows the visualization of the word embeddings using TSNE.



We also attempted to use the FastText library to create similar words:

```
{'issue': ['system', 'able', 'try', 'error', 'get'],
 'job': ['schedule', 'arc', 'cold', 'abort', 'hot'],
 'login': ['password', 'caller', 'unlock', 'log', 'kiosk'],
 'manager': ['management', 'help', 'caller', 'able', 'hi'],
 'network': ['power', 'outage', 'yes', 'circuit', 'divestiture'],
 'password': ['management', 'reset', 'unlock', 'lock', 'login'],
 'schedule': ['job', 'arc', 'cold', 'hot', 'abort'],
 'ticket': ['remote', 'incident', 'network', 'yes', 'type']}
```

We think the techniques we used earlier in Milestone-1 for Data Augmentation yielded better 'synonyms'.

Explore Feature Engineering for Machine Learning

For details, please refer to notebook: capstone_nlp_FeatureEngg-Part2.ipynb

The text and diagram presented here are borrowed from
https://medium.com/wisio/a-gentle-introduction-to-doc2vec-db3e8c0cce5e

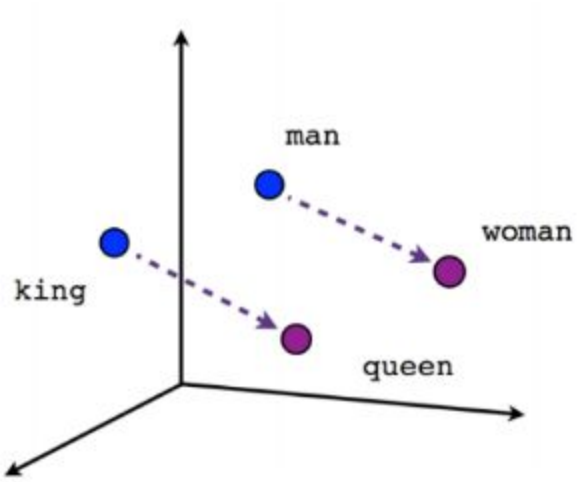
https://towardsdatascience.com/another-twitter-sentiment-analysis-with-python-part-6-doc2vec-603f11832504

Having tried the Word2Vec in Milesone-1, we decided to try out Doc2Vec in this iteration. Doc2vec is a method first presented in 2014 by **Mikilov and Le** in this article

What is Doc2Vec and How different is it from **word2vec**:

word2vec is a well known concept, used to generate representation vectors out of words. In general, when you like to build some model using words, simply labeling/one-hot encoding them is a plausible way to go. However, when using such encoding, the words lose their meaning. e.g, if we encode *Paris* as *id_4*, *France* as *id_6* and *power* as *id_8*, *France* will have the same relation to *power* as with *Paris*. We would prefer a representation in which *France* and *Paris* will be closer than *France* and *power*.

Such representations, encapsulate different relations between words, like synonyms, antonyms, or analogies, such as this one:

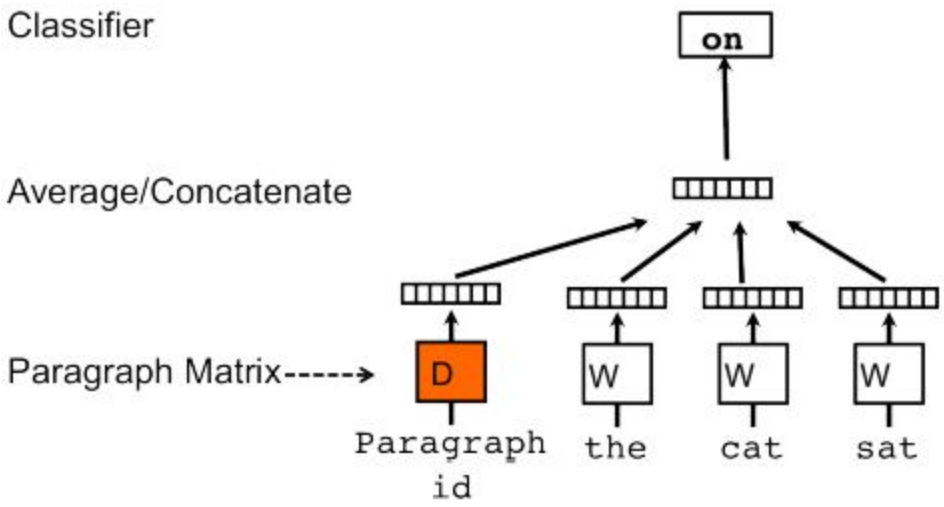


Male-Female

word2vec representation is created using 2 algorithms: Continuous Bag-of-Words model (**CBOW**) and the **Skip-Gram** model.

The goal of **doc2vec** is to create a numeric representation of a document, regardless of its length. But unlike words, documents do not come in logical structures such as words, so the another method has to be found.

The concept that Mikilov and Le have used was simple, yet clever: they have used the **word2vec** model, and added another vector (Paragraph ID below), like so:



So, when training the word vectors W, the document vector D is trained as well, and in the end of training, it holds a numeric representation of the document.

The model above is called *Distributed Memory version of Paragraph Vector* (PV-DM). It acts as a memory that remembers what is missing from the current context — or as the topic of the paragraph. While the word vectors represent the concept of a word, the document vector intends to represent the concept of a document.

There is also another technique Distributed Bag of Words.

DBOW: This is the Doc2Vec model analogous to Skip-gram model in Word2Vec. The paragraph vectors are obtained by training a neural network on the task of predicting a probability distribution of words in a paragraph given a randomly-sampled word from the paragraph.

We tried out these techniques:

1. DBOW (Distributed Bag of Words)
2. DMC (Distributed Memory Concatenated)
3. DMM (Distributed Memory Mean)
4. DBOW + DMC
5. DBOW + DMM

We created unigrams, bigrams and tri-grams using each of the above techniques and ran the Logistic Regression classifier to evaluate the results.

Sample vectors:

```
LabeledSentence(words=['master', 'by', 'are', 'in', 'and', 'material',
'report', 'statistics', 'extended', 'on', 'august', 'which', 'we', 'had',
'to', 'correct', 'this', 'morning', 'today', 'that', 'were', 'blank',
'corrected', 'if', 'manually', 'these', 'the', 'system', 'does', 'not',
'allow', 'us', 'save', 'with', 'a', 'blank', 'need', 'populate', 'these'],
tags=['all_26342']),

 LabeledSentence(words=['when', 'grade', 'on', 'material', 'engineering',
'utility', 'is', 'not', 'if', 'selected', 'used', 'to', 'change', 'the',
'a', 'logic', 'that', 'valid', 'for', 'new', 'grade', 'this', 'working',
'correctly', 'save', 'id', 'screen', 'and', 'invalid', 'are', 'being',
'left', 'affecting', 'all', 'just', 'one', 'needs', 'corrected', 'save',
'delete'], tags=['all_22260']),

 LabeledSentence(words=['our', 'we', 'the', 'error', 'message',
'transfer', 'unit', 'missing'], tags=['all_2149']),

 LabeledSentence(words=['wrong', 'in', 'received', 'from', 'hello', 'a',
'drawing', 'was', 'saved', 'under', 'the', 'number', 'hesitate', 'delete',
'best'], tags=['all_24441']),

 LabeledSentence(words=['collaboration', 'platform', 'site', 'request',
'hello', 'can', 'you', 'add', 'the', 'below', 'to', 'as', 'team', 'i',
'should', 'have', 'ownership', 'edit', 'access'], tags=['all_364']),

 LabeledSentence(words=['job', 'job', 'in', 'job', 'schedule', 'time',
'received', 'from', 'connection', 'error', 'to', 'system'],
tags=['all_22246']),

 LabeledSentence(words=['tool', 'application', 'sending', 'multiple',
'via', 'hi', 'as', 'there', 'send', 'also', 'out', 'of', 'on', 'a',
'daily', 'basis', 'the', 'server', 'same', 'for', 'last', 'since', 'th',
'are', 'duplicate', 'example', 'see', 'attached', 'i', 'have', 'been',
'daily', 'one', 'more', 'thing', 'to', 'note', 'is', 'that', 'count',
'sent', 'per', 'day', 'but', 'not', 'doubled', 'be', 'it', 'twice',
'instead', 'them', 'times', 'issue', 'escalate', 'higher', 'management',
'greatly', 'appreciate', 'if', 'you', 'could', 'look', 'into', 'this',
'let', 'me', 'know', 'need'], tags=['all_17176']),
```

Then we attempted to create similar words to some of the words in our Auto Ticket Assignment vocab:

```
model_ug_dmc.most_similar('network')
```

```
[('dynamic', 0.5188397169113159),
 ('legible', 0.4633473753929138),
 ('wireless', 0.46207863092422485),
 ('electromechanical', 0.45785877108573914),
 ('estate', 0.4521123170852661),
 ('circuit', 0.42887771129608154),
 ('divestiture', 0.4207233786582947),
 ('spread', 0.41820594668388367),
 ('gecko', 0.4153391718864441),
 ('cat', 0.4108252227306366)]
```

Pretty impressive results if once considers the fact tht CAT is actually a network terminology!!!

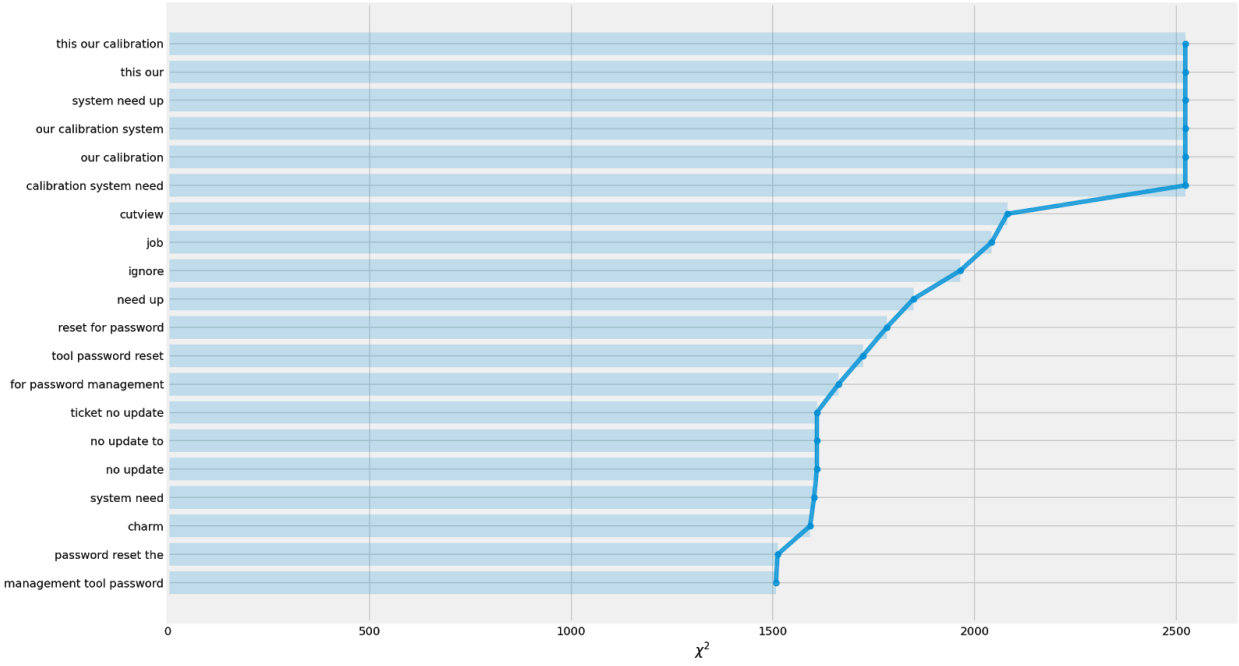
The actual scores returned using the trigram is shown here. Please refer to the notebook capstone_nlp_FeatureEngg-Part2.ipynb

```
RESULT FOR TRIGRAM WITH STOP WORDS (Tfidf)

LogisticRegression(C=1.0, class_weight=None, dual=False,
fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=1000,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001,
verbose=0,
                   warm_start=False)


Validation result for 10000 features
null accuracy: 88.14%
accuracy score: 86.54%
model is 1.60% less accurate than null accuracy
train and test time: 91.88s
--------------------------------------------------------------------------
------
Validation result for 20000 features
null accuracy: 88.14%
accuracy score: 87.50%
model is 0.64% less accurate than null accuracy
train and test time: 116.03s
--------------------------------------------------------------------------
------
Validation result for 30000 features
null accuracy: 88.14%
```

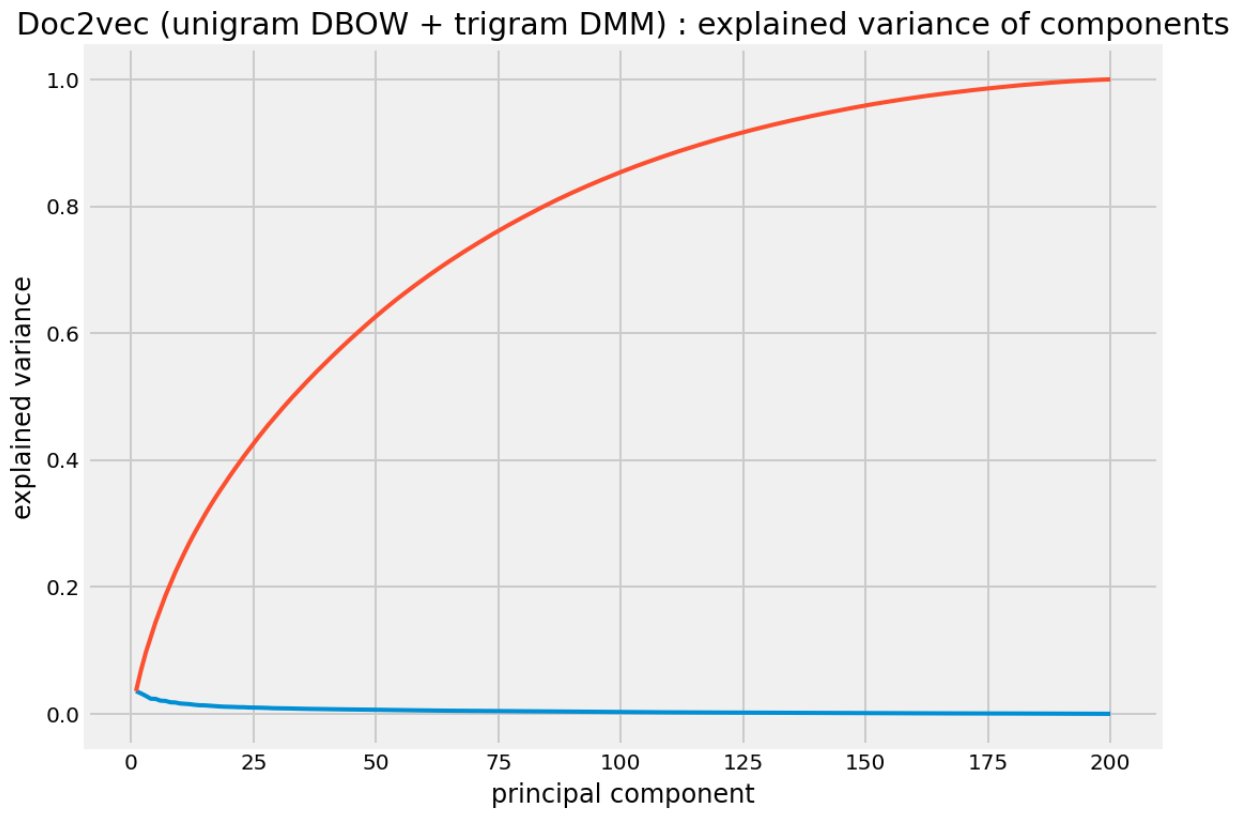
```
accuracy score: 87.82%
model is 0.32% less accurate than null accuracy
train and test time: 138.31s
------------------------------------------------------------------------
------
Validation result for 40000 features
null accuracy: 88.14%
accuracy score: 87.50%
model is 0.64% less accurate than null accuracy
train and test time: 163.69s
------------------------------------------------------------------------
------
Validation result for 50000 features
null accuracy: 88.14%
accuracy score: 88.14%
model has the same accuracy with the null accuracy
train and test time: 181.48s
------------------------------------------------------------------------
------
Validation result for 60000 features
null accuracy: 88.14%
accuracy score: 87.18%
model is 0.96% less accurate than null accuracy
train and test time: 193.35s
------------------------------------------------------------------------
------
Validation result for 70000 features
null accuracy: 88.14%
accuracy score: 87.82%
model is 0.32% less accurate than null accuracy
train and test time: 221.55s
------------------------------------------------------------------------
------
Validation result for 80000 features
null accuracy: 88.14%
accuracy score: 89.10%
model is 0.96% more accurate than null accuracy
train and test time: 239.93s
------------------------------------------------------------------------
------
Validation result for 90000 features
null accuracy: 88.14%
accuracy score: 89.42%
model is 1.28% more accurate than null accuracy
train and test time: 245.18s
------------------------------------------------------------------------
------
Validation result for 100000 features
null accuracy: 88.14%
accuracy score: 89.42%
```

```
model is 1.28% more accurate than null accuracy
train and test time: 261.37s
--------------------------------------------------------------------------------
CPU times: user 38min 27s, sys: 31min 57s, total: 1h 10min 25s
Wall time: 30min 52s
```

Then we attempted Dimensionality Reduction using Chi2.



Finally we attempted to Explain the features using PCA:



The above graph shows the optimal number of features is 175 to explain at least 95% variance.

Apply Feature Engineering on a basic MLP or CNN model and see its performance.

For details please refer to notebook: capstone_nlp_FeatureEngg-NN-Part3.ipynb.

We concatenated the Unigram and Trigram vectors and provided them as input to a simple MLP model:

```
model_d2v_06 = Sequential()
model_d2v_06.add(Dense(128, activation='relu', input_dim=200))
model_d2v_06.add(Dense(128, activation='relu'))
model_d2v_06.add(Dense(128, activation='relu'))
model_d2v_06.add(Dense(74, activation='softmax'))
model_d2v_06.compile(optimizer='adam',
              loss='categorical_crossentropy',
```

```
              metrics=['accuracy'])
```

```
Epoch 1/10
781/781 - 2s - loss: 2.4543 - accuracy: 0.3675 - val_loss: 1.8468 -
val_accuracy: 0.4725
Epoch 2/10
781/781 - 2s - loss: 1.5588 - accuracy: 0.5582 - val_loss: 1.4775 -
val_accuracy: 0.5682
Epoch 3/10
781/781 - 2s - loss: 1.1528 - accuracy: 0.6608 - val_loss: 1.2146 -
val_accuracy: 0.6323
Epoch 4/10
781/781 - 2s - loss: 0.8901 - accuracy: 0.7319 - val_loss: 1.1318 -
val_accuracy: 0.6685
Epoch 5/10
781/781 - 2s - loss: 0.7100 - accuracy: 0.7859 - val_loss: 1.0117 -
val_accuracy: 0.7050
Epoch 6/10
781/781 - 2s - loss: 0.5812 - accuracy: 0.8173 - val_loss: 0.9748 -
val_accuracy: 0.7124
Epoch 7/10
781/781 - 2s - loss: 0.4927 - accuracy: 0.8445 - val_loss: 0.9662 -
val_accuracy: 0.7136
Epoch 8/10
781/781 - 2s - loss: 0.4266 - accuracy: 0.8662 - val_loss: 0.9688 -
val_accuracy: 0.7188
Epoch 9/10
781/781 - 2s - loss: 0.3648 - accuracy: 0.8828 - val_loss: 0.9658 -
val_accuracy: 0.7252
Epoch 10/10
781/781 - 2s - loss: 0.3240 - accuracy: 0.8943 - val_loss: 0.9607 -
val_accuracy: 0.7335
<tensorflow.python.keras.callbacks.History at 0x7f13c9e2b240>
```

We implemented Word2Vec models, one with CBOW (Continuous Bag Of Words) model, and the other with skip-gram model. CBOW model predicts the current word from a window of surrounding context words, while Skip-gram model predicts surrounding context words given the current word. In Gensim package, you can specify whether to use CBOW or Skip-gram by passing the argument "sg" when implementing Word2Vec. By default (sg=0), CBOW is used. Otherwise (sg=1), skip-gram is employed.

```
from gensim.models import KeyedVectors

model_ug_cbow = KeyedVectors.load('w2v_model_ug_cbow.word2vec')

model_ug_sg = KeyedVectors.load('w2v_model_ug_sg.word2vec')

embeddings_index = {}

for w in model_ug_cbow.wv.vocab.keys():

   embeddings_index[w] =
np.append(model_ug_cbow.wv[w],model_ug_sg.wv[w])
```

Keras' 'Tokenizer' will split each word in a sentence, then we can call 'texts_to_sequences' method to get a sequential representation of each sentence. We also need to pass 'num_words' which is a number of vocabularies you want to use, and this will be applied when you call 'texts_to_sequences' method.

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
tokenizer =
Tokenizer(num_words=100000)tokenizer.fit_on_texts(x_train)sequen
ces = tokenizer.texts_to_sequences(x_train)
```

The maximum number of words in a sentence within the training data is 150. Let's decide the maximum length to be a bit longer than this, let's say 155.

```
sequences_val = tokenizer.texts_to_sequences(x_validation)

x_val_seq = pad_sequences(sequences_val, maxlen=155)
```

We will use Embedding layer in Keras. With Embedding layer, we can either pass pre-defined embedding, which we prepared as 'embedding_matrix' above, or Embedding layer itself can learn word embeddings as the whole model trains. And another possibility is we can still feed the pre-defined embedding but make it trainable so that it will update the values of vectors as the model trains.In order to check which method performs better, we defined a simple shallow neural network with one hidden layer.

We first defined a simple CNN as shown below:

```
Model: "model"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_1 (InputLayer)         [(None, 200)]             0
_________________________________________________________________________________________________
embedding_9 (Embedding)      (None, 200, 200)          20000000    input_1[0][0]
_________________________________________________________________________________________________
conv1d_6 (Conv1D)            (None, 199, 100)          40100       embedding_9[0][0]
_________________________________________________________________________________________________
conv1d_7 (Conv1D)            (None, 198, 100)          60100       embedding_9[0][0]
_________________________________________________________________________________________________
conv1d_8 (Conv1D)            (None, 197, 100)          80100       embedding_9[0][0]
_________________________________________________________________________________________________
global_max_pooling1d_5 (GlobalM (None, 100)            0           conv1d_6[0][0]
_________________________________________________________________________________________________
global_max_pooling1d_6 (GlobalM (None, 100)            0           conv1d_7[0][0]
_________________________________________________________________________________________________
global_max_pooling1d_7 (GlobalM (None, 100)            0           conv1d_8[0][0]
_________________________________________________________________________________________________
concatenate (Concatenate)    (None, 300)               0           global_max_pooling1d_5[0][0]
                                                                   global_max_pooling1d_6[0][0]
                                                                   global_max_pooling1d_7[0][0]
_________________________________________________________________________________________________
dense_14 (Dense)             (None, 256)               77056       concatenate[0][0]
_________________________________________________________________________________________________
dropout (Dropout)            (None, 256)               0           dense_14[0][0]
_________________________________________________________________________________________________
dense_15 (Dense)             (None, 74)                19018       dropout[0][0]
_________________________________________________________________________________________________
activation (Activation)      (None, 74)                0           dense_15[0][0]
=================================================================================================
Total params: 20,276,374
Trainable params: 20,276,374
Non-trainable params: 0


Epoch 1/5
781/781 [==============================] - 91s 115ms/step - loss: 2.5075 - accuracy: 0.3919 - val_loss:
0.7200 - val_accuracy: 0.7944
Epoch 2/5
781/781 [==============================] - 89s 115ms/step - loss: 0.6113 - accuracy: 0.8314 - val_loss:
0.3773 - val_accuracy: 0.8901
Epoch 3/5
```

```
781/781 [==============================] - 90s 115ms/step - loss: 0.3056 - accuracy: 0.9104 - val_loss: 0.3877 - val_accuracy: 0.8831

Epoch 4/5

781/781 [==============================] - 89s 114ms/step - loss: 0.2449 - accuracy: 0.9255 - val_loss: 0.3417 - val_accuracy: 0.8985

Epoch 5/5

781/781 [==============================] - 90s 115ms/step - loss: 0.1812 - accuracy: 0.9434 - val_loss: 0.3387 - val_accuracy: 0.9023

<tensorflow.python.keras.callbacks.History at 0x7fdb615f3780>
```

In the final run we merged the bi-gram, tri-gram, four-gram vectors as shown below:

(For details please refer to notebook: capstone_nlp_FeatureEngg_Part4_cnn_w2v.ipynb)

```
ticket_input = Input(shape=(200,), dtype='int32')


ticket_encoder = Embedding(100000, 200, weights=[embedding_matrix], input_length=200, trainable=True)(ticket_input)

bigram_branch = Conv1D(filters=100, kernel_size=2, padding='valid', activation='relu', strides=1)(ticket_encoder)

bigram_branch = GlobalMaxPooling1D()(bigram_branch)

trigram_branch = Conv1D(filters=100, kernel_size=3, padding='valid', activation='relu', strides=1)(ticket_encoder)

trigram_branch = GlobalMaxPooling1D()(trigram_branch)

fourgram_branch = Conv1D(filters=100, kernel_size=4, padding='valid', activation='relu', strides=1)(ticket_encoder)

fourgram_branch = GlobalMaxPooling1D()(fourgram_branch)

merged = concatenate([bigram_branch, trigram_branch, fourgram_branch], axis=1)


merged = Dense(256, activation='relu')(merged)

merged = Dropout(0.2)(merged)

merged = Dense(74)(merged)

output = Activation('softmax')(merged)

model = Model(inputs=[ticket_input], outputs=[output])

model.compile(loss='categorical_crossentropy',

              optimizer='adam',

              metrics=['accuracy'])
model.summary()
```

```
Model: "model"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_1 (InputLayer)         [(None, 200)]             0
_________________________________________________________________________________________________
embedding_9 (Embedding)      (None, 200, 200)          20000000    input_1[0][0]
_________________________________________________________________________________________________
conv1d_6 (Conv1D)            (None, 199, 100)          40100       embedding_9[0][0]
_________________________________________________________________________________________________
conv1d_7 (Conv1D)            (None, 198, 100)          60100       embedding_9[0][0]
_________________________________________________________________________________________________
conv1d_8 (Conv1D)            (None, 197, 100)          80100       embedding_9[0][0]
_________________________________________________________________________________________________
global_max_pooling1d_5 (GlobalM (None, 100)            0           conv1d_6[0][0]
_________________________________________________________________________________________________
global_max_pooling1d_6 (GlobalM (None, 100)            0           conv1d_7[0][0]
_________________________________________________________________________________________________
global_max_pooling1d_7 (GlobalM (None, 100)            0           conv1d_8[0][0]
_________________________________________________________________________________________________
concatenate (Concatenate)    (None, 300)               0           global_max_pooling1d_5[0][0]
                                                                   global_max_pooling1d_6[0][0]
                                                                   global_max_pooling1d_7[0][0]
_________________________________________________________________________________________________
dense_14 (Dense)             (None, 256)               77056       concatenate[0][0]
_________________________________________________________________________________________________
dropout (Dropout)            (None, 256)               0           dense_14[0][0]
_________________________________________________________________________________________________
dense_15 (Dense)             (None, 74)                19018       dropout[0][0]
_________________________________________________________________________________________________
activation (Activation)      (None, 74)                0           dense_15[0][0]
=================================================================================================
Total params: 20,276,374
Trainable params: 20,276,374
Non-trainable params: 0
```

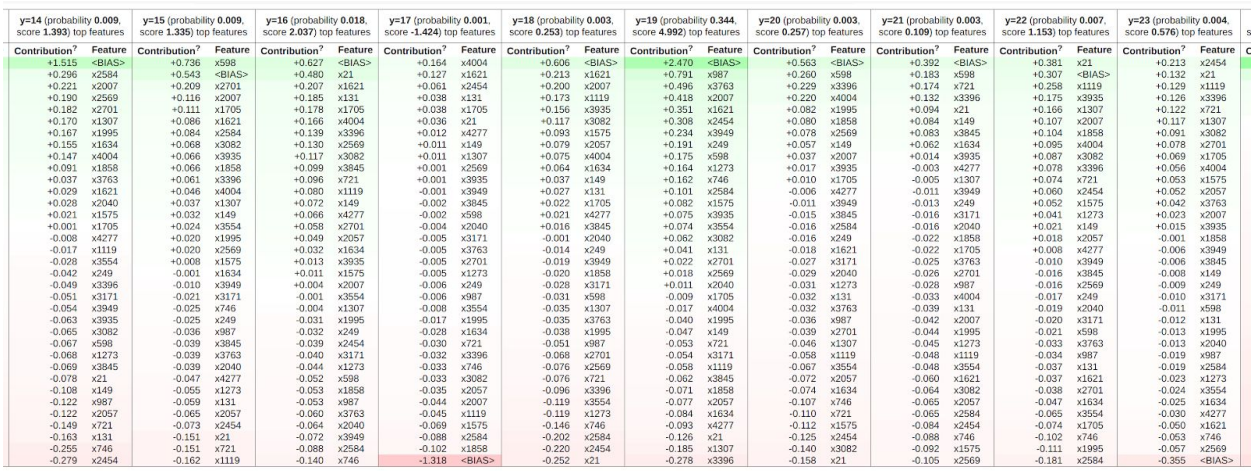
We then attempted an Hyper Parameter Tuning with Random Search:

```
Trial 10 Complete [00h 27m 58s]
val_accuracy: 0.9061499238014221
```

```
Best val_accuracy So Far: 0.9167200326919556

Total elapsed time: 04h 49m 52s

INFO:tensorflow:Oracle triggered exit


Epoch 1/100

1561/1561 [==============================] - 169s 108ms/step - loss:
0.2487 - accuracy: 0.9199 - val_loss: 0.3775 - val_accuracy: 0.9061

Learning Rate =  <tf.Variable 'Adam/learning_rate:0' shape=()
dtype=float32, numpy=0.001>

Epoch 2/100

1561/1561 [==============================] - 168s 107ms/step - loss:
0.2547 - accuracy: 0.9234 - val_loss: 0.4267 - val_accuracy: 0.9017

Learning Rate =  <tf.Variable 'Adam/learning_rate:0' shape=()
dtype=float32, numpy=0.001>

Epoch 3/100

1561/1561 [==============================] - 168s 108ms/step - loss:
0.2379 - accuracy: 0.9281 - val_loss: 0.4342 - val_accuracy: 0.9081

Learning Rate =  <tf.Variable 'Adam/learning_rate:0' shape=()
dtype=float32, numpy=0.001>

Epoch 4/100

1561/1561 [==============================] - 168s 108ms/step - loss:
0.2356 - accuracy: 0.9278 - val_loss: 0.4028 - val_accuracy: 0.9116

Learning Rate =  <tf.Variable 'Adam/learning_rate:0' shape=()
dtype=float32, numpy=0.001>

Epoch 5/100

1561/1561 [==============================] - 168s 108ms/step - loss:
0.2315 - accuracy: 0.9305 - val_loss: 0.4783 - val_accuracy: 0.9042

Learning Rate =  <tf.Variable 'Adam/learning_rate:0' shape=()
dtype=float32, numpy=0.001>

Epoch 6/100

1561/1561 [==============================] - 168s 108ms/step - loss:
0.2179 - accuracy: 0.9329 - val_loss: 0.4472 - val_accuracy: 0.9084

Learning Rate =  <tf.Variable 'Adam/learning_rate:0' shape=()
dtype=float32, numpy=0.001>

Epoch 00006: early stopping
```

Explore Model Interpretation techniques

We used ELI5 to explain the model results from Logistic Regression. For details please refer to notebook capstone-nlp-ML-Model-interpretation.ipynb



y=14 (probability 0.009, score 1.393) top features		y=15 (probability 0.009, score 1.335) top features		y=16 (probability 0.018, score 2.037) top features		y=17 (probability 0.001, score -1.424) top features		y=18 (probability 0.003, score 0.253) top features		y=19 (probability 0.344, score 4.992) top features		y=20 (probability 0.003, score 0.257) top features		y=21 (probability 0.003, score 0.109) top features		y=22 (probability 0.007, score 1.153) top features		y=23 (probability 0.004, score 0.576) top features		s
Contribution?	Feature	Contribution?	Feature	Contribution?	Feature	Contribution?	Feature	Contribution?	Feature	Contribution?	Feature	Contribution?	Feature	Contribution?	Feature	Contribution?	Feature	Contribution?	Feature	
+1.515	<BIAS>	+0.736	x598	+0.627	<BIAS>	+0.164	x4004	+0.606	<BIAS>	+2.470	<BIAS>	+0.563	<BIAS>	+0.392	<BIAS>	+0.381	x21	+0.213	x2454	
+0.296	x2584	+0.543	<BIAS>	+0.480	x21	+0.127	x1621	+0.213	x1621	+0.791	x987	+0.260	x598	+0.183	x598	+0.307	<BIAS>	+0.132	x21	
+0.221	x2007	+0.209	x2701	+0.207	x1621	+0.061	x2454	+0.200	x2007	+0.496	x3763	+0.229	x3396	+0.174	x721	+0.258	x1119	+0.129	x1119	
+0.190	x2569	+0.116	x2007	+0.185	x131	+0.038	x131	+0.173	x1119	+0.418	x2007	+0.220	x4004	+0.132	x3396	+0.175	x3935	+0.126	x3396	
+0.182	x2701	+0.111	x1705	+0.178	x1705	+0.038	x1705	+0.156	x3935	+0.351	x1621	+0.082	x1995	+0.094	x21	+0.166	x1307	+0.122	x721	
+0.170	x1307	+0.086	x1621	+0.166	x4004	+0.036	x21	+0.117	x3082	+0.308	x2454	+0.080	x1858	+0.084	x149	+0.107	x2007	+0.117	x1307	
+0.167	x1995	+0.084	x2584	+0.139	x3396	+0.012	x4277	+0.093	x1575	+0.234	x3949	+0.078	x2569	+0.083	x3845	+0.104	x1858	+0.091	x3082	
+0.155	x1634	+0.068	x3082	+0.130	x2569	+0.011	x149	+0.079	x2057	+0.191	x249	+0.057	x149	+0.062	x1634	+0.095	x4004	+0.078	x2701	
+0.147	x4004	+0.066	x3935	+0.117	x3082	+0.011	x1307	+0.075	x4004	+0.175	x598	+0.037	x2007	+0.014	x3935	+0.087	x3082	+0.069	x1705	
+0.091	x1858	+0.066	x1858	+0.099	x3845	+0.001	x2569	+0.064	x1634	+0.164	x1273	+0.017	x3935	-0.003	x4277	+0.078	x3396	+0.056	x4004	
+0.037	x3763	+0.061	x3396	+0.096	x721	+0.001	x3935	+0.037	x149	+0.162	x746	+0.010	x1705	-0.005	x1307	+0.074	x721	+0.053	x1575	
+0.029	x1621	+0.046	x4004	+0.080	x1119	-0.001	x3949	+0.027	x131	+0.101	x2584	-0.006	x4277	-0.011	x3949	+0.060	x2454	+0.052	x2057	
+0.028	x2040	+0.037	x1307	+0.072	x149	-0.002	x3845	+0.022	x1705	+0.082	x1575	-0.011	x3949	-0.013	x249	+0.052	x1575	+0.042	x3763	
+0.021	x1575	+0.032	x149	+0.066	x4277	-0.002	x598	+0.021	x4277	+0.075	x3935	-0.015	x3845	-0.016	x3171	+0.041	x1273	+0.023	x2007	
+0.001	x1705	+0.024	x3554	+0.058	x2701	-0.004	x2040	+0.016	x3845	+0.074	x3554	-0.016	x2584	-0.016	x2040	+0.021	x149	+0.015	x3935	
-0.008	x4277	+0.020	x1995	+0.049	x2057	-0.005	x3171	-0.001	x2040	+0.062	x3082	-0.016	x249	-0.022	x1858	+0.018	x2057	-0.001	x1858	
-0.017	x1119	+0.020	x2569	+0.032	x1634	-0.005	x3763	-0.014	x249	+0.041	x131	-0.018	x1621	-0.022	x1705	+0.008	x4277	-0.006	x3949	
-0.028	x3554	+0.008	x1575	+0.013	x3935	-0.005	x2701	-0.019	x3949	+0.022	x2701	-0.027	x3171	-0.025	x3763	-0.010	x3949	-0.006	x3845	
-0.042	x249	-0.001	x1634	+0.011	x1575	-0.005	x1273	-0.020	x1858	+0.018	x2569	-0.029	x2040	-0.026	x2701	-0.016	x3845	-0.008	x149	
-0.049	x3396	-0.010	x3949	+0.004	x2007	-0.006	x249	-0.028	x3171	+0.011	x2040	-0.031	x1273	-0.028	x987	-0.016	x2569	-0.009	x249	
-0.051	x3171	-0.021	x3171	-0.001	x3554	-0.006	x987	-0.031	x598	-0.009	x1705	-0.032	x131	-0.033	x4004	-0.017	x249	-0.010	x3171	
-0.054	x3949	-0.025	x746	-0.004	x1307	-0.008	x3554	-0.035	x1307	-0.017	x4004	-0.032	x3763	-0.039	x131	-0.019	x2040	-0.011	x598	
-0.063	x3935	-0.025	x249	-0.031	x1995	-0.017	x1995	-0.035	x3763	-0.040	x1995	-0.036	x987	-0.042	x2007	-0.020	x3171	-0.012	x131	
-0.065	x3082	-0.036	x987	-0.032	x249	-0.028	x1634	-0.038	x1995	-0.047	x149	-0.039	x2701	-0.044	x1995	-0.021	x598	-0.013	x1995	
-0.067	x598	-0.039	x3845	-0.039	x2454	-0.030	x721	-0.051	x987	-0.053	x721	-0.046	x1307	-0.045	x1273	-0.033	x3763	-0.013	x2040	
-0.068	x1273	-0.039	x3763	-0.040	x3171	-0.032	x3396	-0.068	x2701	-0.054	x3171	-0.058	x1119	-0.048	x1119	-0.034	x987	-0.019	x987	
-0.069	x3845	-0.039	x2040	-0.044	x1273	-0.033	x746	-0.076	x2569	-0.058	x1119	-0.067	x3554	-0.048	x3554	-0.037	x131	-0.019	x2584	
-0.078	x21	-0.047	x4277	-0.052	x598	-0.033	x3082	-0.076	x721	-0.062	x3845	-0.072	x2057	-0.060	x1621	-0.037	x1621	-0.023	x1273	
-0.108	x149	-0.055	x1273	-0.053	x1858	-0.035	x2057	-0.096	x3396	-0.071	x1858	-0.074	x1634	-0.064	x3082	-0.038	x2701	-0.024	x3554	
-0.122	x987	-0.059	x131	-0.053	x987	-0.044	x2007	-0.119	x3554	-0.077	x2057	-0.107	x746	-0.065	x2057	-0.047	x1634	-0.025	x1634	
-0.122	x2057	-0.065	x2057	-0.060	x3763	-0.045	x1119	-0.119	x1273	-0.084	x1634	-0.110	x721	-0.065	x2584	-0.065	x3554	-0.030	x4277	
-0.149	x721	-0.073	x2454	-0.064	x2040	-0.069	x1575	-0.146	x746	-0.093	x4277	-0.112	x1575	-0.084	x2454	-0.074	x1705	-0.050	x1621	
-0.163	x131	-0.151	x21	-0.072	x3949	-0.088	x2584	-0.202	x2584	-0.126	x21	-0.125	x2454	-0.088	x746	-0.102	x746	-0.053	x746	
-0.255	x746	-0.151	x721	-0.088	x2584	-0.102	x1858	-0.220	x2454	-0.185	x1307	-0.140	x3082	-0.092	x1575	-0.111	x1995	-0.057	x2569	
-0.279	x2454	-0.162	x1119	-0.140	x746	-1.318	<BIAS>	-0.252	x21	-0.278	x3396	-0.158	x21	-0.105	x2569	-0.181	x2584	-0.355	<BIAS>	

Explore these Deep Learning Models & Fine tune the models with Hyper Parameter Tuning:

Attention Encoder-Decoder

We attempted to tune these parameters using Bayesian using Keras HyperTuner:

- Learning rate
- Batch Size

The code is below:

#Attention with encoder decoder

inp = Embedding(max_features,emb_dim,input_length=inp_len)(inputs)

lstm_out = LSTM(64, return_sequences=**True**)(inp)

attention = Dense(1, activation='relu')(lstm_out)

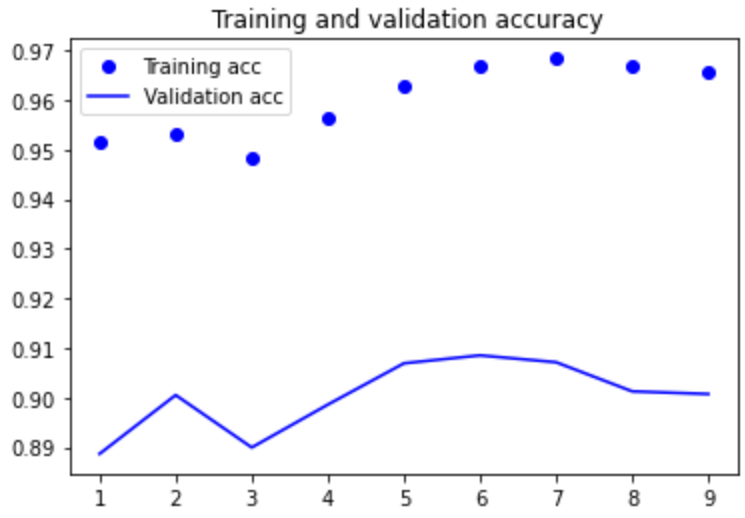
attention = Flatten()(attention)

attention = Activation('softmax')(attention)

attention = RepeatVector(64)(attention)

```
#attention = LSTM(64,return_sequences=False)(attention)
attention = Permute([2,1])(attention)

combined = concatenate([lstm_out, attention])
combined_mul = Flatten()(combined)
decode = RepeatVector(64)(combined_mul)

decode = LSTM(64, return_sequences=True)(decode)
decode=Flatten()(decode)
decode = (Dense(74))(decode)

decode = Activation('softmax')(decode)

model_enc_dec_att = tf.keras.Model(inputs=inputs, outputs=decode)

model_enc_dec_att.compile(optimizer='adam', loss='categorical_crossentropy',metrics =
['accuracy'])
print(model_enc_dec_att.summary())
def ENC_DEC_ATT_model(hp):#attention with encoder decoder
 model = model_enc_dec_att
 print(model.summary())
 lr = hp.Choice('learning_rate', [1e-2, 1e-3, 1e-4])
 model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

 return model
Hyperparameters for ENC_DEC_ATT are
{'learning_rate': 0.001, 'batch_size': 128}
```

The training stops at 9 epochs due to early stopping callback.



Bi Directional LSTM (using Bayesian Hyper Parameter Tuning)

Please refer to notebook capstone-nlp-DL-GlOve-BILSTM.ipynb

We attempted to tune these parameters using Bayesian, Random Search techniques using Keras HyperTuner:

- Units for each layer (eg: Dense, LSTM, DropOut)
- Learning rate
- Momentum
- Epochs
- Batch Size

Only Bayesian allowed us to tune the epochs & batch size . The code is below:

```
def build_model(hp):
    #word_index, embeddings_matrix, nclasses = get_params()
    model = Sequential()
    hidden_layer = 6
    #gru_node = 32
    dropout=0.2
    #HP_DROPOUT = hp2.HParam('dropout', hp2.RealInterval(0.1, 0.2))
```

```
model.add(Embedding(len(word_index) + 1,
                    EMBEDDING_DIM,
                    weights=[embedding_matrix],
                    input_length=MAX_SEQUENCE_LENGTH,
                    trainable=True))
#print(gru_node)
for i in range(0,hidden_layer):
model.add(tf.keras.layers.Bidirectional(tf.compat.v1.keras.layers.CuDNNLSTM(
units=hp.Int('lstm_node_1_units', min_value=16, max_value=256, step=16),
#units=hp.Choice('lstm_node_1_units', values = [32,128,256]),
return_sequences=True
)))
model.add(
Dropout(
hp.Choice('dropout_1', values = [0.1,0.2,0.3,0.4,0.5]),
))
model.add(BatchNormalization())
model.add(tf.keras.layers.Bidirectional(
#tf.compat.v1.keras.layers.CuDNNGRU(gru_node)
tf.compat.v1.keras.layers.CuDNNLSTM(
units=hp.Int('lstm_node_2_units', min_value=16, max_value=256, step=16)
#units=hp.Choice('lstm_node_2_units', values = [32,128,256])
)))
model.add(
Dropout(
hp.Choice('dropout_2', values = [0.1,0.2,0.3,0.4,0.5]),
))
model.add(BatchNormalization())
```

```
        model.add(
        #Dense(256, activation='relu')
        Dense(units=hp.Int('dense_1_units', min_value=32, max_value=512, step=16),
        #units=hp.Choice('dense_1_units', values = [32,128,256]),
        activation='relu'
        ))
        model.add(BatchNormalization())
        model.add(Dense(74, activation='softmax'))
        #lr = hp.Choice('learning_rate', values=[0.001, 1e-2, 1e-3, 1e-4])
        #momentum = hp.Choice('momentum', values=[0.0, 0.2, 0.4, 0.6, 0.8, 0.9])
        optimizer=tf.keras.optimizers.SGD(hp.Choice('learning_rate', values = [1e-2, 1e-3,
1e-4]),
                          hp.Choice('momentum', values=[0.0, 0.2, 0.4, 0.6, 0.8, 0.9]))
        model.compile(optimizer=optimizer,
        loss = 'sparse_categorical_crossentropy',
        metrics=['accuracy']
        )
        return model

tuner_search = RandomSearch(build_model,
                    objective='val_accuracy',
                    max_trials=100,
                    directory='outputs',
                    project_name='TicketAssignment'
                    )

class MyTuner(kt.tuners.BayesianOptimization):
  def run_trial(self, trial, *args, **kwargs):
        # You can add additional HyperParameters for preprocessing and custom training
loops
```

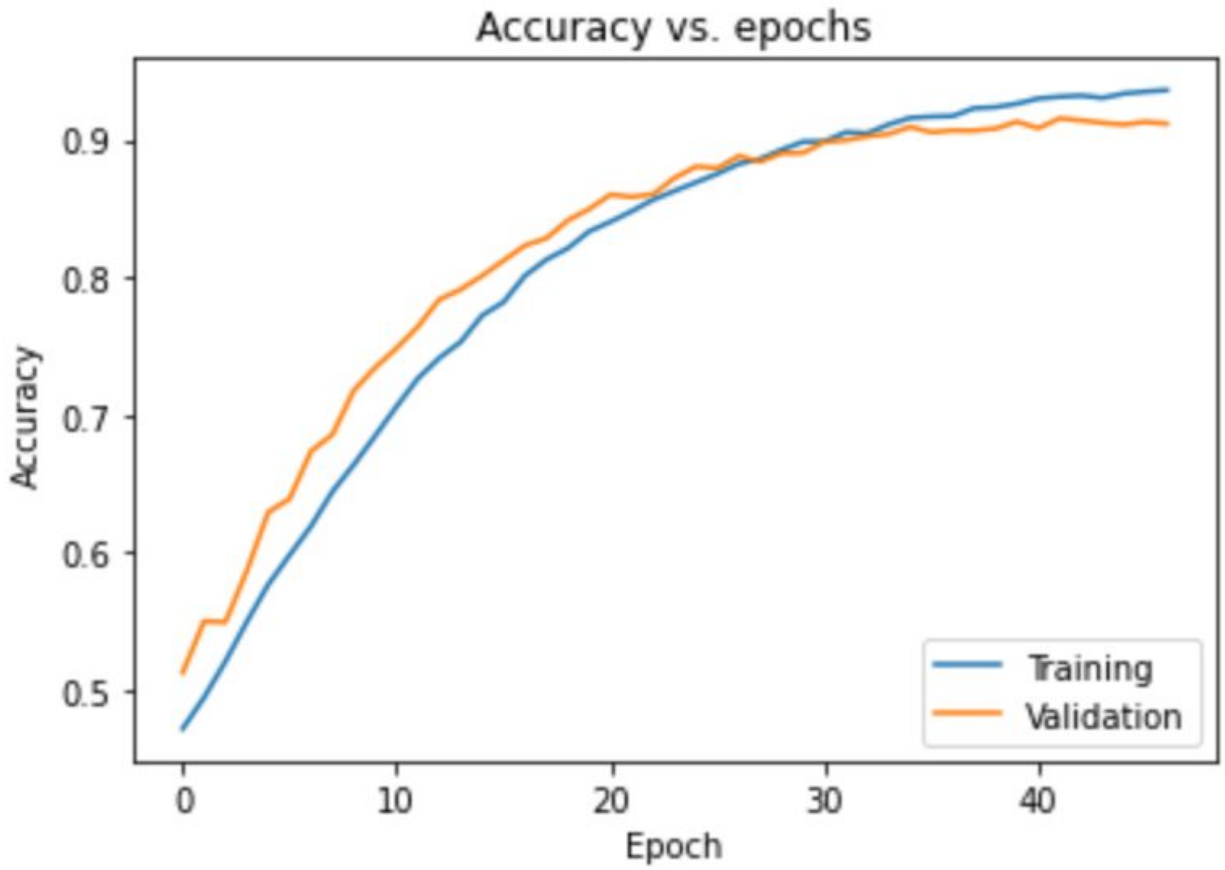
```
# via overriding `run_trial`
kwargs['batch_size'] = trial.hyperparameters.Int('batch_size', 16, 256, step=32)
super(MyTuner, self).run_trial(trial, *args, **kwargs)
```

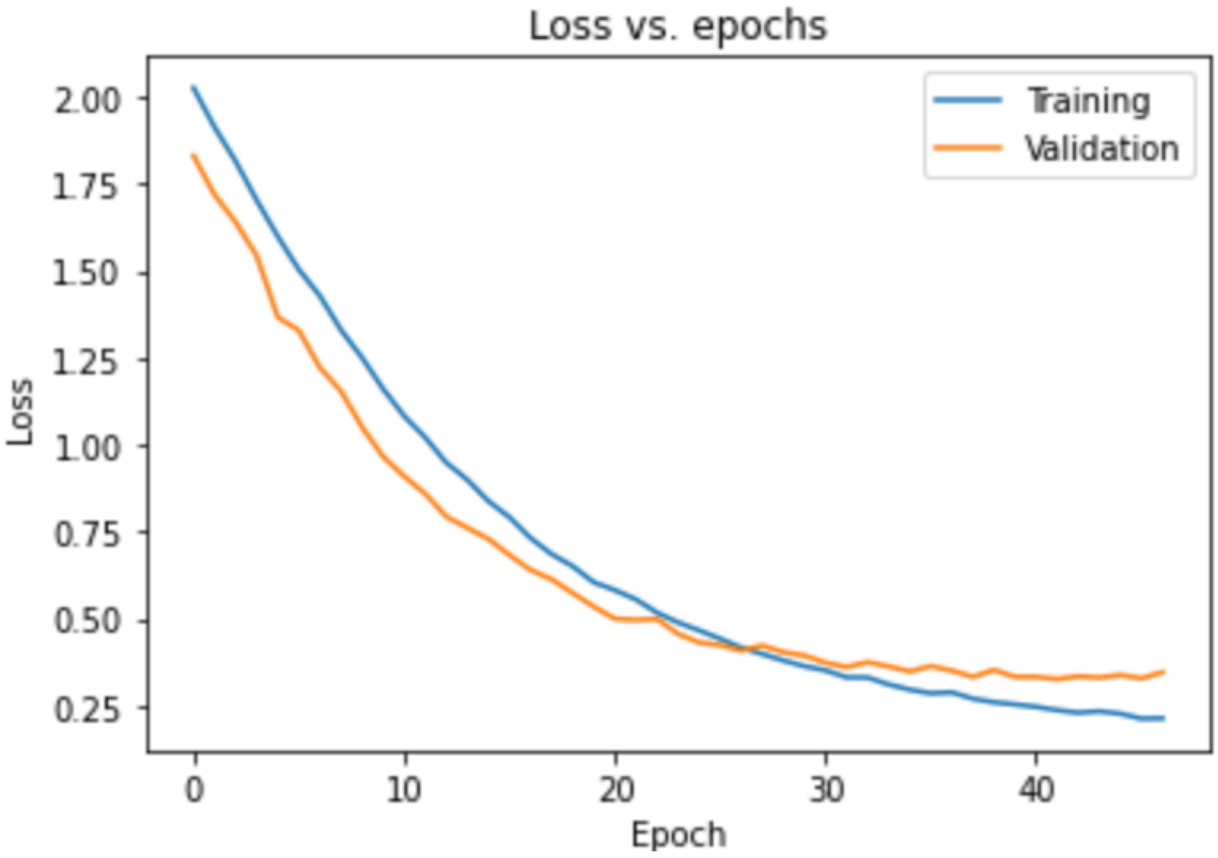
The results achieved are shown below:

Hyperparameter	Value	Best Value So Far
lstm_node_1_units	128	128
dropout_1	0.1	0.1
lstm_node_2_units	256	256
dropout_2	0.3	0.3
dense_1_units	256	256
learning_rate	0.001	0.001
momentum	0	0
batch_size	16	16
epochs	10	10

The model is able to achieve high accuracy with the least number of epochs when the batch_size is set to 16.

```
Epoch 00044: saving model to checkpoints_best_only/checkpoint
Learning Rate =  <tf.Variable 'SGD/learning_rate:0' shape=() dtype=float32, numpy=0.001>
Epoch 45/200
1558/1558 [==============================] - 402s 258ms/step - loss: 0.2267 - accuracy: 0.9335 - val_loss: 0.3381 - val_accuracy: 0.9109

val/train: 1.49

Epoch 00045: saving model to checkpoints_best_only/checkpoint
Learning Rate =  <tf.Variable 'SGD/learning_rate:0' shape=() dtype=float32, numpy=0.001>
Epoch 46/200
1558/1558 [==============================] - 379s 243ms/step - loss: 0.2128 - accuracy: 0.9350 - val_loss: 0.3286 - val_accuracy: 0.9129

val/train: 1.54

Epoch 00046: saving model to checkpoints_best_only/checkpoint
Learning Rate =  <tf.Variable 'SGD/learning_rate:0' shape=() dtype=float32, numpy=0.001>
Epoch 47/200
1558/1558 [==============================] - 364s 234ms/step - loss: 0.2135 - accuracy: 0.9360 - val_loss: 0.3461 - val_accuracy: 0.9116

val/train: 1.62

Epoch 00047: saving model to checkpoints_best_only/checkpoint
Learning Rate =  <tf.Variable 'SGD/learning_rate:0' shape=() dtype=float32, numpy=0.001>
Epoch 00047: early stopping
```


Accuracy vs. epochs
Accuracy
0.9
0.8
0.7
0.6
0.5
0
10
20
30
40
Epoch
Training
Validation



BiLSTM Performance with RandomSearch:

Please refer to notebook capstone_nlp_DL_GlOve_BILSTM_RandomSearch.ipynb

```
Epoch 00030: saving model to checkpoints_best_only/checkpoint
Learning Rate =  <tf.Variable 'SGD/learning_rate:0' shape=() dtype=float32, numpy=0.01>
Epoch 31/100
781/781 [==============================] - 278s 356ms/step - loss: 0.2068 - accuracy: 0.9304 - val_loss: 0.3924 - val_accuracy: 0.8924

val/train: 1.90

Epoch 00031: saving model to checkpoints_best_only/checkpoint
Learning Rate =  <tf.Variable 'SGD/learning_rate:0' shape=() dtype=float32, numpy=0.01>
Epoch 32/100
781/781 [==============================] - 284s 364ms/step - loss: 0.2166 - accuracy: 0.9282 - val_loss: 0.4100 - val_accuracy: 0.8884

val/train: 1.89

Epoch 00032: saving model to checkpoints_best_only/checkpoint
Learning Rate =  <tf.Variable 'SGD/learning_rate:0' shape=() dtype=float32, numpy=0.01>
Epoch 33/100
781/781 [==============================] - 283s 363ms/step - loss: 0.2068 - accuracy: 0.9305 - val_loss: 0.4003 - val_accuracy: 0.8945

val/train: 1.94

Epoch 00033: saving model to checkpoints_best_only/checkpoint
Learning Rate =  <tf.Variable 'SGD/learning_rate:0' shape=() dtype=float32, numpy=0.01>
Epoch 34/100
781/781 [==============================] - 282s 361ms/step - loss: 0.2089 - accuracy: 0.9303 - val_loss: 0.4496 - val_accuracy: 0.8818

val/train: 2.15

Epoch 00034: saving model to checkpoints_best_only/checkpoint
Learning Rate =  <tf.Variable 'SGD/learning_rate:0' shape=() dtype=float32, numpy=0.01>
Epoch 00034: early stopping
```

Evaluation Accuracy: 0.882

Evaluation Loss: 0.450

Mean Accuracy for the validation dataset:

0.8372015216771294

Mean Loss for the validation dataset:

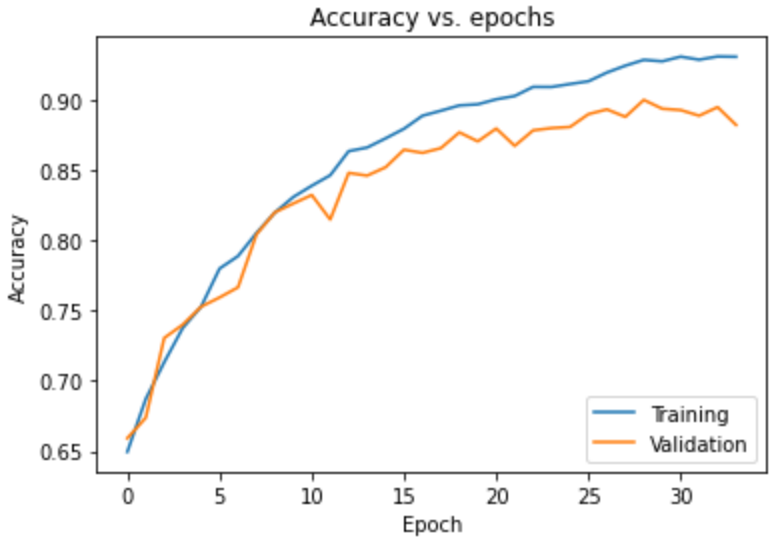
0.5667102152810377

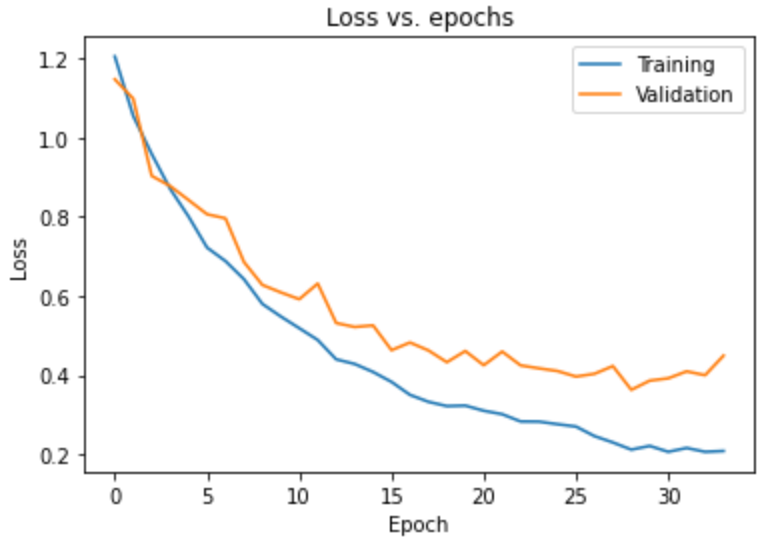
Mean Accuracy for the training dataset:

0.8577045763240141

Mean Loss for the training dataset:

0.4572651806999655





BERT initial run:

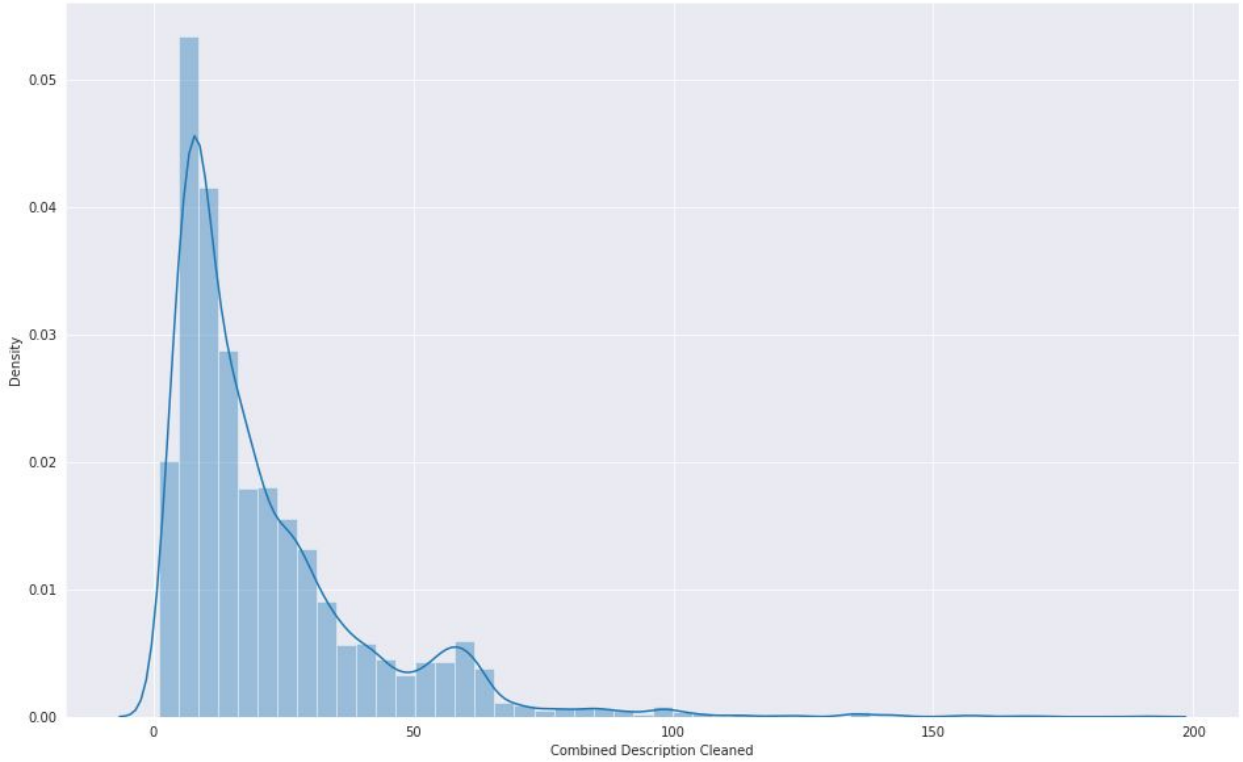
Please refer to notebook Bert-Transformers.ipynb
http://jalammar.github.io/illustrated-gpt2/

https://www.topbots.com/generalized-language-models-bert-openai-gpt2/

BERT, short for **Bidirectional Encoder Representations from Transformers** (Devlin, et al., 2019) is a direct descendant to GPT: train a large language model on free text and then fine-tune on specific tasks without customized network architectures.

Compared to GPT, the largest difference and improvement of BERT is to make training **bi-directional**. The model learns to predict both context on the left and right. The paper according to the ablation study claimed that:

Determining Sequence Length:



Model: "model"

Layer (type)	Output Shape	Param #	Connected to
input_ids (InputLayer)	[(None, 100)]	0	
attention_mask (InputLayer)	[(None, 100)]	0	
tf_bert_model (TFBertModel)	TFBaseModelOutputWit	108310272	input_ids[0][0] attention_mask[0][0]
global_max_pooling1d (GlobalMax	(None, 768)	0	tf_bert_model[0][0]
batch_normalization (BatchNorma	(None, 768)	3072	global_max_pooling1d[0][0]
dense (Dense)	(None, 128)	98432	batch_normalization[0][0]
dropout_37 (Dropout)	(None, 128)	0	dense[0][0]
dense_1 (Dense)	(None, 32)	4128	dropout_37[0][0]
outputs (Dense)	(None, 74)	2442	dense_1[0][0]

Total params: 108,418,346
Trainable params: 106,538
Non-trainable params: 108,311,808

```
Epoch 00097: ReduceLROnPlateau reducing learning rate to 1.0000000656873453e-06.
Epoch 98/100
877/877 [==============================] - 60s 68ms/step - loss: 1.1671 - accuracy: 0.6828 - val_loss: 0.4613 - va
l_accuracy: 0.8691
Epoch 99/100
877/877 [==============================] - 60s 68ms/step - loss: 1.1700 - accuracy: 0.6836 - val_loss: 0.5109 - va
l_accuracy: 0.8588
Epoch 100/100
877/877 [==============================] - 60s 68ms/step - loss: 1.1692 - accuracy: 0.6824 - val_loss: 0.5206 - va
l_accuracy: 0.8559
```

Evaluation Accuracy: 0.864

Evaluation Loss: 0.475

Mean Accuracy for the validation dataset:

0.8167302113771439

Mean Loss for the validation dataset:

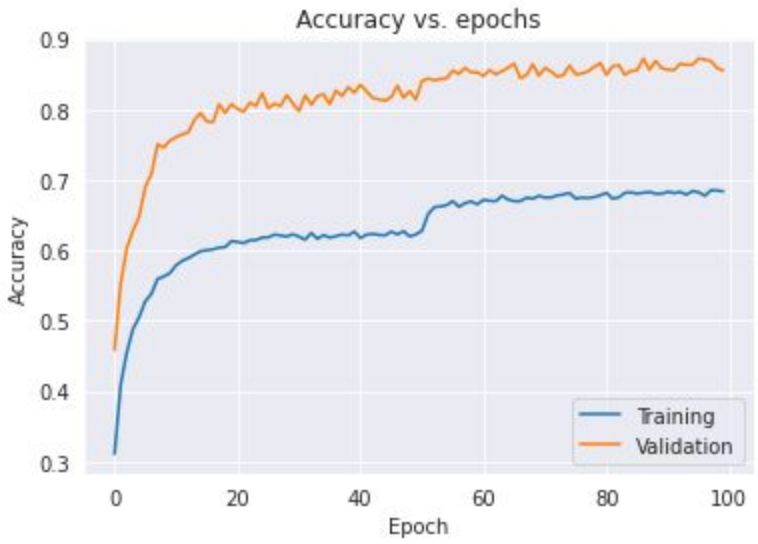
0.6493391236662864

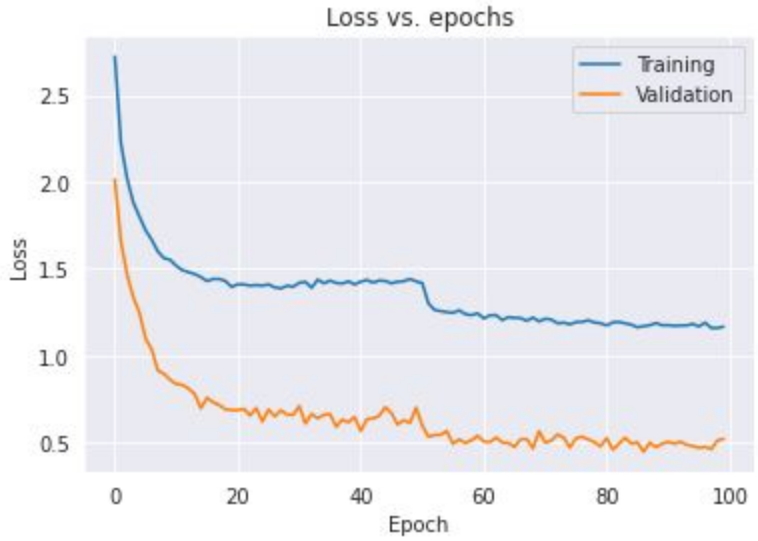
Mean Accuracy for the training dataset:

0.6322445100545884

Mean Loss for the training dataset:

1.3612296533584596





Observations:

a. The validation accuracy was consistently higher than the training accuracy
b. We found about 395+ words that are part of the corpus of our dataset not present in BERT Vocab. A sample list is below:

   Subsystem, rudimentary, telephony, reselect, reallocate, stoppage, unresolved, magenta, intercom, pax, wallpaper, projector, charger, legit, wird, revisit, tort, populate, disconnection, deactivation, unrestricted, presume, airway, scannable, misconfiguration, salesperson, headset, extrude, kiosk, resale, mailbox

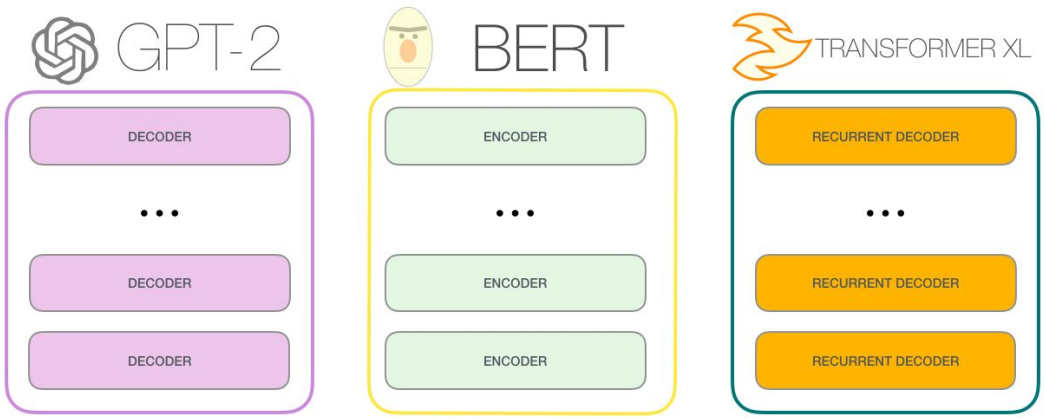
   We think this is one of the reasons why the BERT accuracy could not cross the 80s range.

GPT2 initial run:

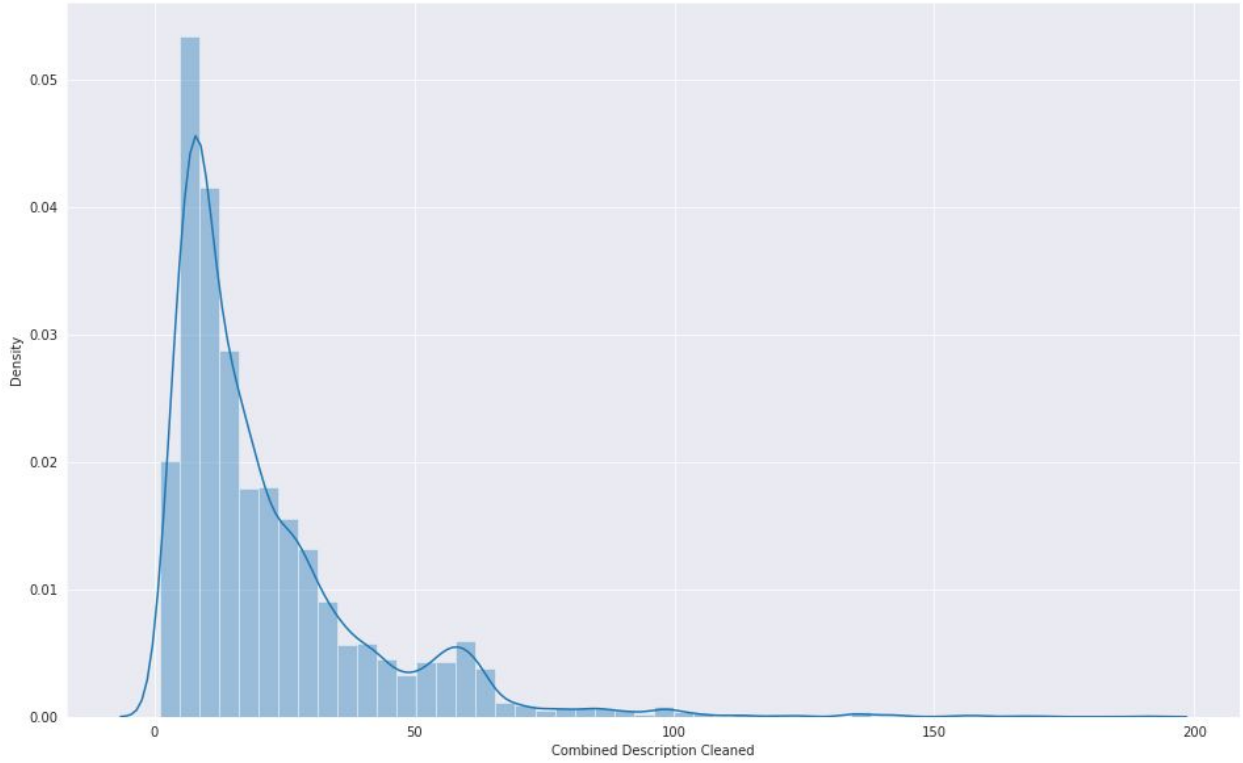
Please refer to notebook: GPT2-Transformers2.ipynb

BERT vs GPT2

The GPT-2 is built using transformer decoder blocks. BERT, on the other hand, uses transformer encoder blocks. But one key difference between the two is that GPT2, like traditional language models, outputs one token at a time. Let's for example prompt a well-trained GPT-2 to recite the first law of robotics:



Best Sequence Length:



```
Model: "model"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_ids (InputLayer)       [(None, 100)]             0
_________________________________________________________________________________________________
attention_mask (InputLayer)  [(None, 100)]             0
_________________________________________________________________________________________________
tfgp_t2model (TFGPT2Model)   TFBaseModelOutputWit 124439808   input_ids[0][0]
                                                               attention_mask[0][0]
_________________________________________________________________________________________________
global_max_pooling1d (GlobalMax (None, 768)            0           tfgp_t2model[0][0]
_________________________________________________________________________________________________
batch_normalization (BatchNorma (None, 768)            3072        global_max_pooling1d[0][0]
_________________________________________________________________________________________________
dense (Dense)                (None, 128)               98432       batch_normalization[0][0]
_________________________________________________________________________________________________
dropout_37 (Dropout)         (None, 128)               0           dense[0][0]
_________________________________________________________________________________________________
dense_1 (Dense)              (None, 32)                4128        dropout_37[0][0]
_________________________________________________________________________________________________
outputs (Dense)              (None, 74)                2442        dense_1[0][0]
=================================================================================================
Total params: 124,547,882
Trainable params: 106,538
Non-trainable params: 124,441,344
_________________________________________________________________________________________________
```

```
877/877 [==============================] - 45s 52ms/step - loss: 1.0895 - accuracy: 0.7104 - val_loss: 0.9310 - v
al_accuracy: 0.8275
Epoch 99/100
877/877 [==============================] - 45s 52ms/step - loss: 1.0578 - accuracy: 0.7127 - val_loss: 1.0654 - v
al_accuracy: 0.8223
Epoch 100/100
877/877 [==============================] - 45s 52ms/step - loss: 1.0612 - accuracy: 0.7126 - val_loss: 1.2418 - v
al_accuracy: 0.8162
```

Evaluation Accuracy: 0.814

Evaluation Loss: 1.271

Mean Accuracy for the validation dataset:

0.7966688188910485

Mean Loss for the validation dataset:

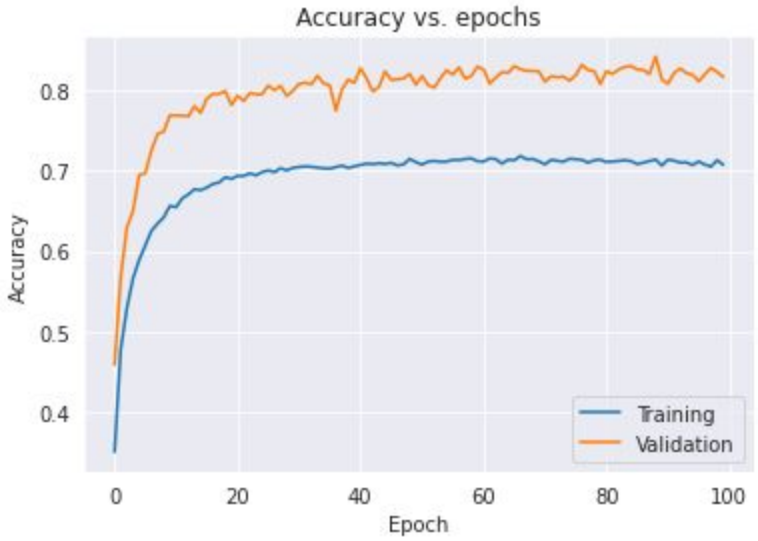
1.00599816262722

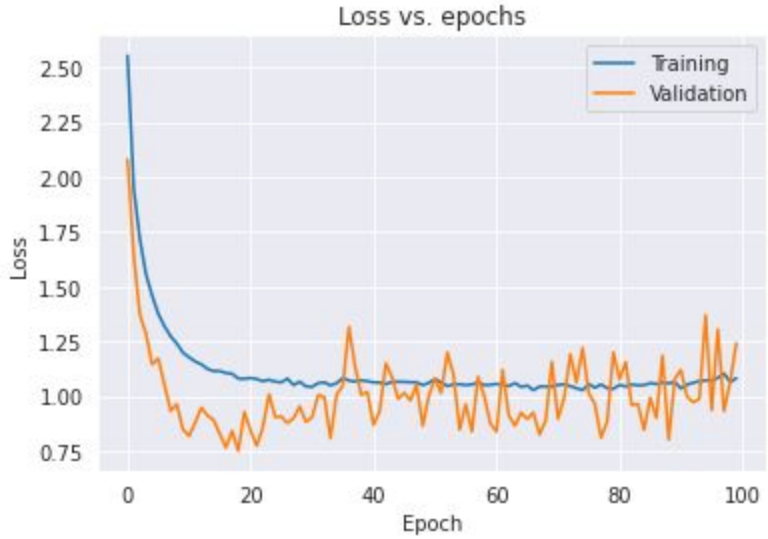
Mean Accuracy for the training dataset:

0.6913536873459816

Mean Loss for the training dataset:

1.114402573108673





GPT2 with Hyperparams:

```
Trial 9 Complete [00h 05m 14s]
val_accuracy: 0.5159935355186462

Best val_accuracy So Far: 0.5159935355186462
Total elapsed time: 00h 47m 40s
INFO:tensorflow:Oracle triggered exit
Model: "model"
```

Layer (type)	Output Shape	Param #	Connected to
input_ids (InputLayer)	[(None, 100)]	0	
attention_mask (InputLayer)	[(None, 100)]	0	
tfgp_t2model (TFGPT2Model)	TFBaseModelOutputWit	124439808	input_ids[0][0] attention_mask[0][0]
global_max_pooling1d (GlobalMax	(None, 768)	0	tfgp_t2model[38][0]
batch_normalization (BatchNorma	(None, 768)	3072	global_max_pooling1d[0][0]
dense (Dense)	(None, 240)	184560	batch_normalization[0][0]
dropout (Dropout)	(None, 240)	0	dense[0][0]
dense_1 (Dense)	(None, 128)	30848	dropout[0][0]
outputs (Dense)	(None, 74)	9546	dense_1[0][0]

```
Total params: 124,667,834
Trainable params: 226,490
Non-trainable params: 124,441,344
```

The hyperparameter search is complete. The best parameters are:

first densely-connected layer is 240

the optimal learning rate for the optimizer is 0.0001

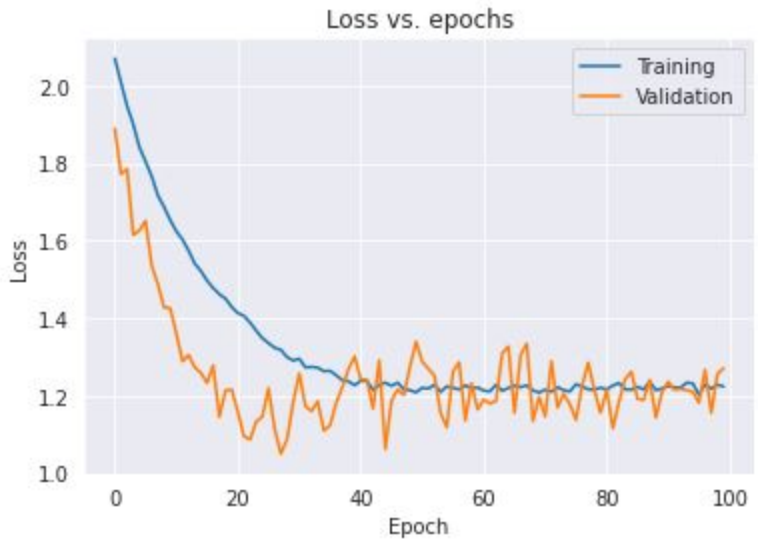
the dropout unit is 0.1

the momentum us 0.4

the second densely-connected layer is 128

the batch size is 128

```
Epoch 99/100
877/877 [==============================] - 49s 56ms/step - loss: 1.2287 - accuracy: 0.6902 - val_loss: 1.2561 - v
al_accuracy: 0.7528
Learning Rate =  <tf.Variable 'RMSprop/learning_rate:0' shape=() dtype=float32, numpy=1e-06>
Epoch 100/100
877/877 [==============================] - 51s 58ms/step - loss: 1.2236 - accuracy: 0.6913 - val_loss: 1.2705 - v
al_accuracy: 0.7596
Learning Rate =  <tf.Variable 'RMSprop/learning_rate:0' shape=() dtype=float32, numpy=1e-06>
```



Mean Accuracy for the validation dataset:

0.7376090431213379

Mean Loss for the validation dataset:

1.2496958827972413

Mean Accuracy for the training dataset:

0.6613184145092964

Mean Loss for the training dataset:

1.3292483127117156

Evaluation Accuracy: 0.767

Evaluation Loss: 1.279

Explore approach for Production Implementation for Real-time calls to an API that can return the Group number the ticket is assigned to.

While we did not have the time to do the implementation, we do want to spend a few minutes on how this model can be invoked in production after training to assign the ticket to the right group.

There are 3 major steps :

Initial Training:

1. Model Training: Using one of the models built from scratch like the CNN or the BiLSTM, or using GPT2 and BERT pretrained models, our objective is to achieve a validation accuracy of above 95%.

Daily Fine Tuning

2. Then this model will be fed with new data in batch mode every night . This can be new tickets collected during the course of that business day. The objective is for the model to update itself based on new information.

Real-time API to call the model.predict

3. At this point, the model is ready to make real-time predictions. We will expose the model.predict as a REST Method behind an API Gateway. The API Gateway will perform the AAA functions (Authentication, Authorization & Audit). Once the call is vetted, the API Gateway will pass on the call to the Application Server that will have the implementation logic. We suggest using an Application Framework like FLASK in Python to implement the model.predict method. Prior to calling this method, the

application will need to convert the input string to the appropriate vector (like word2vec, doc2vec, etc). After the prediction, the data is converted back to string and returned back to the caller as a string - GRP_, GRP_2, etc.

Closing Reflections

The table below shows some of the most significant Model Results:

As show in the green highlighted rows - we were able to increase the Validation accuracy results on the simple CNN using merged vectors (bi-gram, trigram, four-gram) compared to Milistone-1

As shown in the yellow highlighted rows for BiLSTM - we were able to get the margin between validation & Training accuracy to under 3% .

Model	Validation Acc	Training Acc	Val Loss	Training Loss	Ref Page	Notebook
Simple CNN with multi-gram Feature Engg & Hyper Param Tuning	93.29	90.84	0.44	0.32	21	capstone_nlp_FeatureEngg_Part4_cnn_w2v
Attention Encoder-Decoder with Hyper Param	90.07	96.56	0.44	0.08	23	Attention_hyp
Bi-LSTM with GloVe and Bayesian Hyper Param tuning	91.16	93.6	0.34	0.21	27	capstone-nlp-DL-GlOve-BILSTM.ipynb
BERT	86	69	0.52	1.16	33	Bert-Transformers.ipynb
GPT-2	81.62	71.26	1.24	1.06	36	GPT2-Transformers2.ipynb
Simple CNN (Milestone-1)	92.91	84.29	0.59	0.22	60	capstone-nlp-DL-GlOve.ipynb
Bi-LSTM (Milestone-1)	91.7	95.6	0.31	0.12	48	capstone-nlp-DL-Model-initial-run..ipynb

We think any of the above models can be used to further fine-tune for Production.

Limitations

The most significant limitation we encountered that may have prevented us in achieving better validation accuracy results is the lack of variation in the data . This can be seen in the difference in accuracy between unaugment dataset and the augmented dataset - but we think even the augment dataset (about 31000 records) is not enough. In production, using the different techniques we have shown in this project, the dataset volume needs to be significantly increased to upwards of 100,000 records.

Meta-Learning:

One other technique that can be used is the Meta learning technique in Production. In this technique, the model continues to learn incrementally based on tickets that get created every day .

Implications

Using the meta-learning technique and using stacking or voting classifiers, our suggestion would be to use at least 2 of the top Machine Learning Models plus 2 of the Deep Learning Models in production actual prediction.

Comparison to Benchmark

At the beginning of this project we had set this goal:

> Create various Machine Learning Models that can help classify incidents and assign them to the right Functional Group. Our objective is to create NLP models that can predict with at least 80% accuracy.

As the table above shows, we have achieved this goal - all the top models have an accuracy of 80% or higher.
Why and How did better the benchmark expectations:

1. Data Augmentation techniques we used helped us significantly better results
2. Hyper Parameter tuning
3. Using multi-gram techniques

What have you learnt from the process?

1. Quality Data is the most significant input to model building. This along with good data augmentation techniques improve the accuracy of models significantly.

2. Machine Learning Models also provide impressive results and should NOT be ignored .

----------The above concludes Milestone-2----------

Initial Run of the Machine Learning & Deep Learning Models :

Please note: This section was presented for Milestone - 1 : repeated here for continuation purposes.

Deciding Models and Model Building

The dataset indicates a Multi-Classification problem with the 74 for labels. Our initial thinking is these models will be applicable in this class of problem.

We used the dataset after Augmentation for running the Machine Learning models. The dataset name is input_data_after_data_for_ml__with_augmentation_knn_round2.csv

We did these steps to prepare the data for feeding into the ML algorithms:

a. We applied the Label Encoder on the labels.
b. For the Features, we used the Count Vectorizer and did a fit transform
c. We applied the Tfidf Transformer to create a matrix of Tf-idf features.
d. We used the sklearn train test split to do a 80:20 split of the Training and Test data.

Machine Learning Models:

For more details on the Machine Learning Models, please review this notebook: ***capstone-nlp-ML-Model-initial-run.ipynb***

1. Logistic Regression: Logistic regression is one of the most simple Machine Learning models. They are easy to understand, interpretable, and can give pretty good results. Every practitioner using Logistic Regression out there must know about the Log-Odds which is the main concept behind this learning algorithm. The Logistic regression is very much interpretable considering the business needs and explanation regarding how the model works concerning different independent variables used in the model.

Parameters we used for Logistic Regression:

solver='lbfgs', max_iter=1000, multi_class='multinomial'

For multiclass problems, 'lbfgs' handlesmultinomial loss

For 'multinomial' the loss minimised is the multinomial loss fit across the entire probability distribution, *even when the data is binary*.

Results from Logistic Regression:

	Algorithm	Accuracy	f1	Precision Score	Recall Score
1	LR 30% Test Data after Data Augmentation Word Embedding Round 2	76.48%	76.10%	78.98%	76.77%

2. Support Vector Machines - *SVM is a sparse technique*. Like nonparametric methods, SVM requires that all the training data be available, that is, stored in memory during the training phase, when the parameters of the SVM model are learned. However, once the model parameters are identified, SVM depends only on a subset of these training instances, called *support vectors*, for future prediction. Support vectors define the margins of the hyperplanes.

Parameters we used for SVC:

We first used the GridSearchCV to find the best parameters, Some Parameters we are trying the find the ideal values for:

- C is used during the training phase and says how much outliers are taken into account in calculating Support Vectors

- gamma determines the radius of the area of influence of the support vectors if it is too large, the radius of the area of influence of the support vectors only includes the support vector itself and no amount of regularization with C will be able to prevent overfitting. When gamma is very small, the model is too constrained and cannot capture the complexity or “shape” of the data.
- kernel parameter selects the type of hyperplane used to separate the data.
- We will let the Hypersearch determine the best kernel to use among
    - rbf : The RBF kernel is a stationary kernel. It is also known as the “squared exponential” kernel.
    - poly : A polynomial kernel is a more generalized form of the linear kernel. The polynomial kernel can distinguish curved or nonlinear input space. The polynomial and RBF are especially useful when the data-points are not linearly separable.
    - sigmoid: The Sigmoid Kernel comes from the Neural Networks field, where the bipolar sigmoid function is often used as an activation function for artificial neurons
- probability: We will choose between True & False

Results from SVC:

	Algorithm	Accuracy	f1	Precision Score	Recall Score
1	SVC 30% Test Data after Data Augmentation Word Embedding Round 2	91.43%	91.44%	94.37%	89.24%
2	LR 30% Test Data after Data Augmentation Word Embedding Round 2	76.48%	76.10%	78.98%	76.77%

3. K Nearest Neighbors (KNN) - KNN is another useful algorithm for Multi classification problems. The KNN algorithm assumes that similar things exist in close proximity. In other words, similar things are near to each other . As we note later, we used this algorithm also to efficiently find synonyms of the key words in our corpus.

Parameters we used for KNN:

We first ran the KNN to find the optimal k between 1 and 20. We then applied the KNN Classifier with the optimal k value found .

Results from SVC:

	Algorithm	Accuracy	f1	Precision Score	Recall Score
1	SVC 30% Test Data after Data Augmentation Word Embedding Round 2	91.43%	91.44%	94.37%	89.24%
2	KNN 30% Test Data after Data Augmentation Word Embedding Round 2	88.81%	88.70%	89.36%	90.07%
3	LR 30% Test Data after Data Augmentation Word Embedding Round 2	76.48%	76.10%	78.98%	76.77%

4. Ensembles
    a. Bagging Classifier: Bagging uses a simple approach that shows up in statistical analyses again and again — improve the estimate of one by combining the estimates of many. Bagging constructs n classification trees using bootstrap sampling of the training data and then combines their predictions to produce a final meta-prediction.

Parameters used for Bagging Classifier:

n_estimators=10, max_samples= .7, bootstrap=True

Results from Bagging Classifier:

	Algorithm	Accuracy	f1	Precision Score	Recall Score
1	SVC 30% Test Data after Data Augmentation Word Embedding Round 2	91.43%	91.44%	94.37%	89.24%
2	KNN 30% Test Data after Data Augmentation Word Embedding Round 2	88.81%	88.70%	89.36%	90.07%
3	Bagging Classifier 30% Test Data after Data Augmentation Word Embedding Round 2	83.94%	83.94%	92.17%	84.95%
4	DT 30% Test Data after Data Augmentation Word Embedding Round 2	78.37%	78.35%	82.03%	79.56%
5	LR 30% Test Data after Data Augmentation Word Embedding Round 2	76.48%	76.10%	78.98%	76.77%

    b. Decision Tree: Decision tree builds classification or regression models in the form of a tree structure. It breaks down a dataset into smaller and smaller subsets while at the same time an associated decision tree is incrementally developed. The final result is a tree with **decision nodes** and **leaf nodes**. A decision node (e.g., Outlook) has two or more branches (e.g., Sunny, Overcast and Rainy). Leaf node (e.g., Play) represents a classification or decision. The topmost decision node in a tree which corresponds to the best predictor called **root node**. Decision trees can handle both categorical and numerical data.

Parameters used for Decision Tree:

We built our model using the DecisionTreeClassifier function. Using default 'gini' criteria to split. Another option is 'entropy'.

Gini: measures how often a randomly chosen element from the set would be incorrectly labeled.

Entropy: It is used to measure the impurity or randomness of a dataset

Results from Decision Tree:

	Algorithm	Accuracy	f1	Precision Score	Recall Score
1	SVC 30% Test Data after Data Augmentation Word Embedding Round 2	91.43%	91.44%	94.37%	89.24%
2	KNN 30% Test Data after Data Augmentation Word Embedding Round 2	88.81%	88.70%	89.36%	90.07%
3	DT 30% Test Data after Data Augmentation Word Embedding Round 2	78.37%	78.35%	82.03%	79.56%
4	LR 30% Test Data after Data Augmentation Word Embedding Round 2	76.48%	76.10%	78.98%	76.77%

c. XGBoost: The XGBoost algorithm is effective for a wide range of regression and classification predictive modeling problems. It is an efficient implementation of the stochastic gradient boosting algorithm and offers a range of hyperparameters that give fine-grained control over the model training procedure. Although the algorithm performs well in general, even on imbalanced classification datasets, it offers a way to tune the training algorithm to pay more attention to misclassification of the minority class for datasets with a skewed class distribution

Parameters used for XGB:

We used GridSearchCV to look for best parameters for XGB. These are some of the parameters fed into the GridSearchCV algorithm:

- min_child_weight : Minimum sum of instance weight (hessian) needed in a child.
- gamma: Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger gamma is, the more conservative the algorithm will be.
- subsample: Subsample ratio of the training instances. Setting it to 0.5 means that XGBoost would randomly sample half of

the training data prior to growing trees. and this will prevent overfitting.

- colsample_bytree: is the subsample ratio of columns when constructing each tree. Subsampling occurs once for every tree constructed.

Results from XGB:

	Algorithm	Accuracy	f1	Precision Score	Recall Score
1	SVC 30% Test Data after Data Augmentation Word Embedding Round 2	91.43%	91.44%	94.37%	89.24%
2	KNN 30% Test Data after Data Augmentation Word Embedding Round 2	88.81%	88.70%	89.36%	90.07%
3	XGBoost Classifier 30% Test Data after Data Augmentation Word Embedding Round 2	84.06%	84.16%	90.65%	83.31%
4	Bagging Classifier 30% Test Data after Data Augmentation Word Embedding Round 2	83.94%	83.94%	92.17%	84.95%
5	DT 30% Test Data after Data Augmentation Word Embedding Round 2	78.37%	78.35%	82.03%	79.56%
6	LR 30% Test Data after Data Augmentation Word Embedding Round 2	76.48%	76.10%	78.98%	76.77%

5. Stacker:

   We also tried with Stacker Classifier with layer 1 and layer 2 estimators. In Layer 1 we used these estimators: SVC, Logistic Regression, Decision Tree. Final Estimator used was XGB.

   Layer 2 - we used these estimators: Bagging Classifier, Logistic Regression, XG Boost. Layer 2 served as the final estimator for Layer 1.

	Algorithm	Accuracy	f1	Precision Score	Recall Score
1	SVC 30% Test Data after Data Augmentation Word Embedding Round 2	91.43%	91.44%	94.37%	89.24%
2	Stacker 30% Test Data after Data Augmentation Word Embedding Round 2	90.50%	90.46%	93.74%	88.29%
3	KNN 30% Test Data after Data Augmentation Word Embedding Round 2	88.81%	88.70%	89.36%	90.07%
4	XGBoost Classifier 30% Test Data after Data Augmentation Word Embedding Round 2	84.06%	84.16%	90.65%	83.31%
5	Bagging Classifier 30% Test Data after Data Augmentation Word Embedding Round 2	83.94%	83.94%	92.17%	84.95%
6	DT 30% Test Data after Data Augmentation Word Embedding Round 2	78.37%	78.35%	82.03%	79.56%
7	LR 30% Test Data after Data Augmentation Word Embedding Round 2	76.48%	76.10%	78.98%	76.77%

Deep Learning Models

For more details on the Deep Learning Models, please review these notebooks:

capstone-nlp-DL-Model-initial-run.ipynb

capstone-nlp-DL-GlOve.ipynb

So far we have implemented the following models for unaugmented data, level1 augmented data and level2 augmented data.

Deep learning models

We have implemented the following 7 deep learning models

1) **LSTM**-

cp_dl.ipynb(without hyperparameter tuning)

cp_dl _hyp.ipynb(with hyperparameter tuning)

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network capable of learning order dependence in sequence prediction problems.This is a behavior required in complex problem domains like machine translation, speech recognition, and more.A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell.LSTM is the response to RNN's Vanishing Gradient Problem.The intuition behind LSTM is that the machine will learn the importance of previous words, so that we will not lose information from older hidden states.

Advantages of LSTM

*Non-decaying error backpropagation.

*For long time lag problems, LSTM can handle noise and continuous values.

*No parameter fine tuning.

*Memory for long time periods

2) **GRU**-

cp_dl.ipynb(without hyperparameter tuning)

cp_dl _hyp.ipynb(with hyperparameter tuning)

GRU stands for Gated Recurrent Units.They are also provided with a gated mechanism to effectively and adaptively capture dependencies of different time scales. They have an update gate and a reset gate. The

former is responsible for selecting what piece of knowledge is to be carried forward, whereas the latter lies in between two successive recurrent units and decides how much information needs to be forgotten.

Advantages of GRU

*Shorter training time

*Requires fewer data points to capture the properties of the data

3) **Bidirectional LSTM**-

cp_dl.ipynb(without hyperparameter tuning)

cp_dl _hyp.ipynb(with hyperparameter tuning)

Bidirectional Lstm are really just putting two independent Lstms together. This structure allows the networks to have both backward and forward information about the sequence at every time step.Using bidirectional will run your inputs in two ways, one from past to future and one from future to past and what differs this approach from unidirectional is that in the LSTM that runs backward you preserve information from the future and using the two hidden states combined you are able in any point in time to preserve information from both past and future.

Advantage of Bidirectional LSTM

*it understands the context better.

4)**Bidirectional GRU**

cp_dl.ipynb(without hyperparameter tuning)

cp_dl _hyp.ipynb(with hyperparameter tuning)

Bidirectional GRU's are a type of bidirectional recurrent neural networks with only the input and forget gates. It allows for the use of information from both previous time steps and later time steps to make predictions about the current state

Advantage of Bidirectional GRU

*it understands the context better.

5)**Encoder Decoder(without attention mechanism):**

Transformer_no_att_no_hyp.ipynb(without hyperparameter tuning)

Transformer_no_att_hyp.ipynb(with hyperparameter tuning)

A sequence to sequence model lies behind numerous systems which you face on a daily basis. For instance, seq2seq model powers applications like Google Translate, voice-enabled devices and online chatbots.The model consists of 3 parts: encoder, intermediate (encoder) vector and decoder.

Encoder

- A stack of several recurrent units (LSTM or GRU cells for better performance) where each accepts a single element of the input sequence, collects information for that element and propagates it forward.

Encoder Vector

- This is the final hidden state produced from the encoder part of the model.
- This vector aims to encapsulate the information for all input elements in order to help the decoder make accurate predictions.
- It acts as the initial hidden state of the decoder part of the model.

Decoder

- A stack of several recurrent units where each predicts an output *y_t* at a time step *t*.

- Each recurrent unit accepts a hidden state from the previous unit and produces an output as well as its own hidden state.

6) **Sequence to sequence model with attention mechanism**

Attention_no_hyp.ipynb(without hyperparameter tuning)

Attention_hyp.ipynb(with hyperparameter tuning)

: Two variations of the model have been implemented

a)Encoder with attention mechanism

b)Encoder Decoder with attention mechanism

The main drawback of this sequence to sequence model is evident. If the encoder makes a bad summary, the prediction will also be bad. And indeed it has been observed that the encoder creates a bad summary when it tries to understand longer sentences. It is called the long-range dependency problem of RNN/LSTMs.RNNs cannot remember longer sentences and sequences due to the vanishing/exploding gradient problem. It can remember the parts which it has just seen. The performance of the encoder-decoder network degrades rapidly as the length of the input sentence increases.

Although an LSTM is supposed to capture the long-range dependency better than the RNN, it tends to become forgetful in specific cases.

The basic idea: each time the model predicts an output word, it only uses parts of an input where the most relevant information is concentrated instead of an entire sentence. In other words, it only pays attention to some input words. .Encoder works as usual, and the difference is only on the decoder's part. As you can see from a picture, the decoder's hidden state is computed with a context vector, the previous output and the previous hidden state. But now we use not a single context vector c, but a separate context vector for each target word.These context vectors are computed as a weighted sum of *annotations* generated by the encoder. The weight of each annotation is computed by an *alignment model* which scores how well the inputs and the output match

Implementations

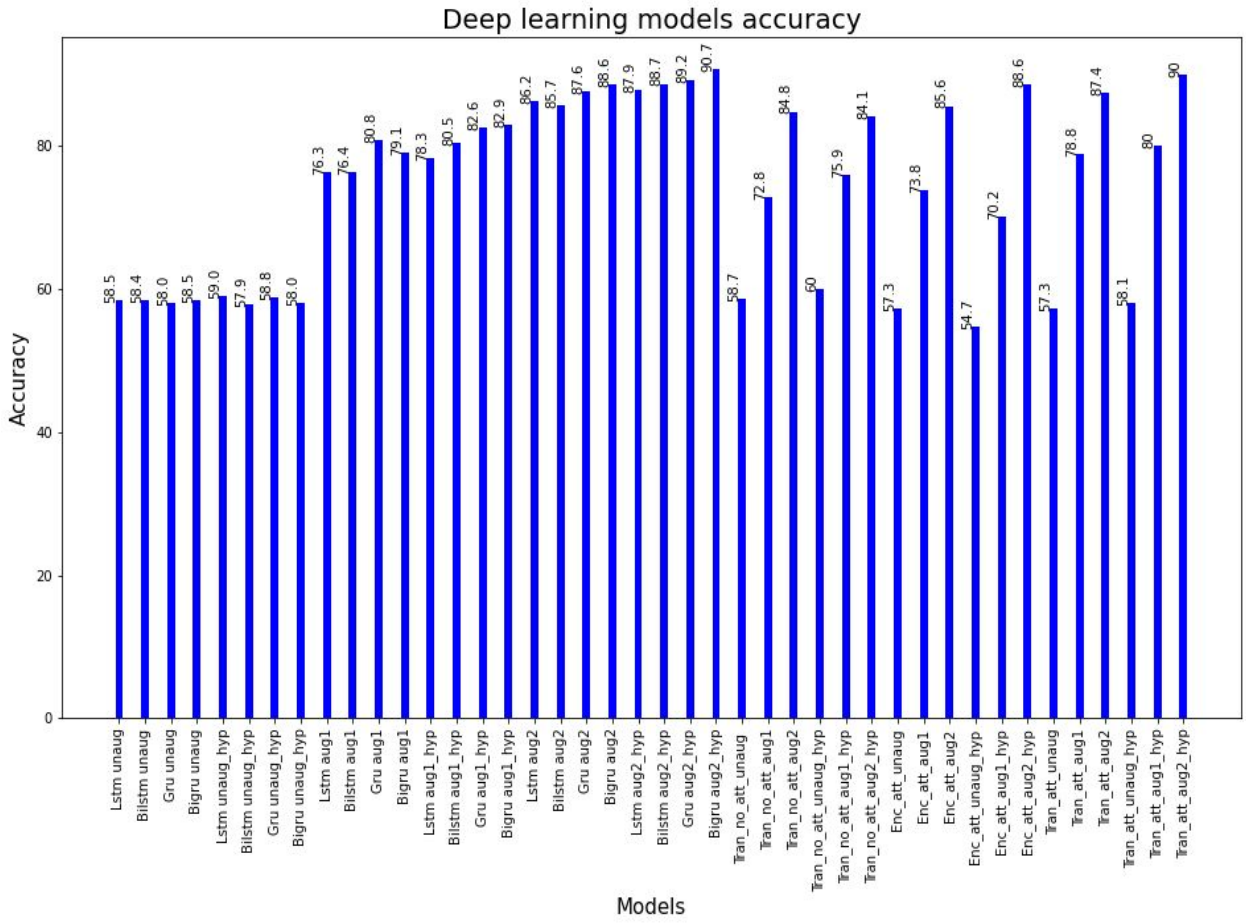
There are 2 variations of each model one without hyperparameter tuning and one with hyperparameter tuning. The hyperparameter tuning was done for batch size, and learning rate. Thus we have a total of 42 models. The accuracy scores are shown in the table below

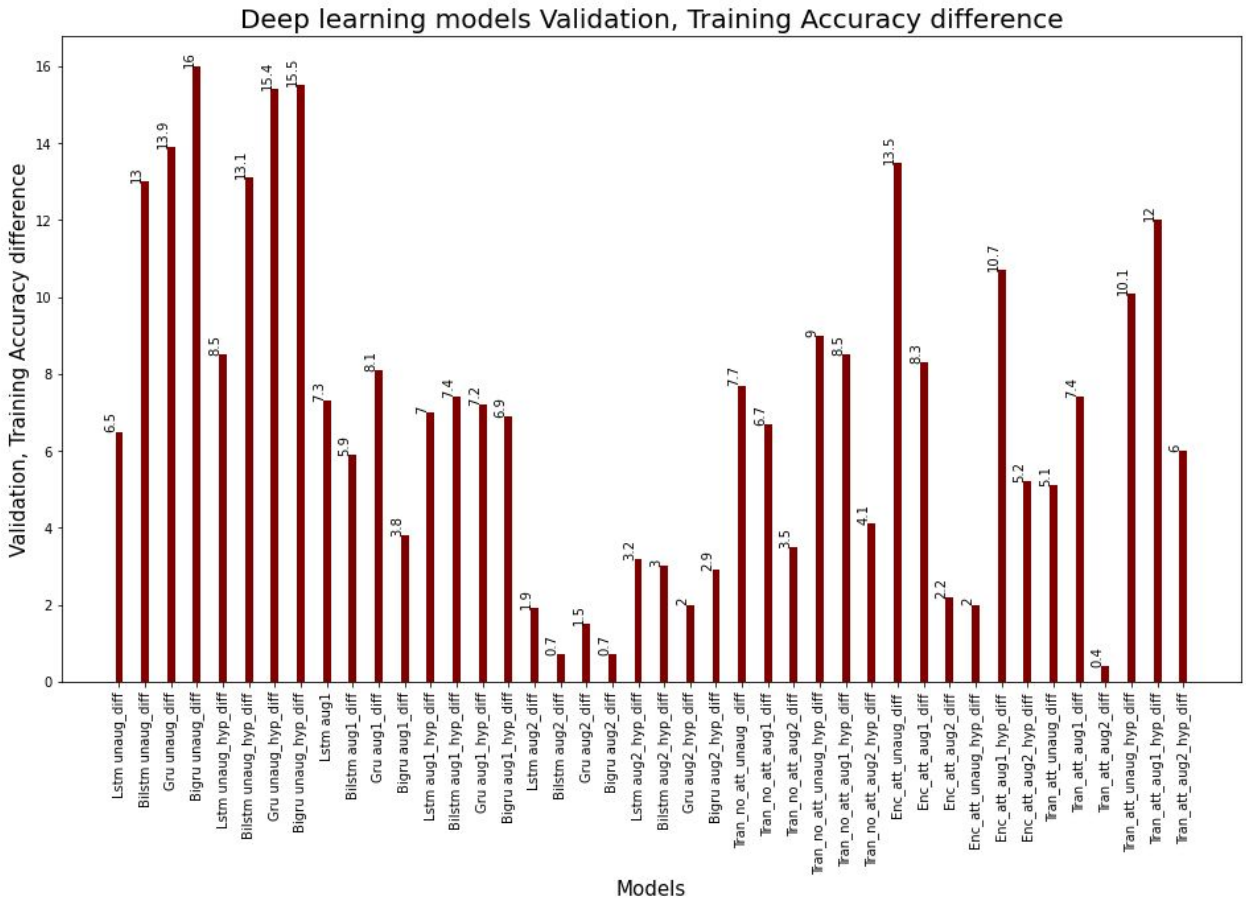
Model	Unaug Acc/Train Acc	Aug1 Acc/Train Acc	Aug2 Acc/Train Acc	Unaug Acc/Train Acc with Hyp	Aug1 Acc/Train Acc with Hyp	Aug2 Acc/Train Acc with Hyp
Lstm	**58.5/ 65.0**	**76.3/ 83.6**	**86.2/ 88.1**	**59.0/ 67.5**	**78.3/ 85.3**	**87.9/ 91.2**
Bilstm	**58.4/ 71.4**	**76.4/ 82.3**	**85.7/ 86.4**	**57.9/ 71.0**	**80.5/ 87.9**	**88.7/ 91.7**
Gru	**58.0/ 71.9**	**80.8/ 88.9**	**87.6/ 89.1**	**58.8/ 74.2**	**82.6/ 89.8**	**89.2/ 91.2**
BiGru	**58.5/ 74.5**	**79.1/ 82.9**	**88.6/ 89.3**	**58.0/ 73.5**	**82.9/ 89.8**	**90.7/ 93.6**

***Trans_ no_att**	**58.7/ 66.4**	**72.8/ 79.5**	**84.8/ 88.3**	**60.0/ 69.0**	**75.9/ 84.4**	**84.1/ 88.2**
Enc_att	**57.3/ 70.8**	**73.8/ 82.1**	**85.6/ 87.8**	**54.7/ 56.7**	**70.2/ 80.9**	**88.6/ 93.8**
***Trans_ att**	**57.3/ 62.4**	**78.8/ 86.2**	**87.4/ 87.8**	**58.1/ 68.2**	**80.0/ 92.0**	**90.0/ 96.0**

*Trans-Transformer(Encoder-Decoder)

Table1:Accuracy Scores





Conclusions

It can be seen that augmentation of the data gives a substantial improvement in the accuracy score and hyperparameter tuning gives a marginal improvement in most cases. The Bidirectional GRU using level 2 augmented data gives the highest accuracy of 90.7%,followed by Transformer with attention with an accuracy of 90%.The level 2

augmented data with hyperparameter tuned models have an accuracy improvement of at least over 25% in all models and these models have an **accuracy in the range 84.1 to 90.7.**Transformer models are more suited for language translation but were used to compare accuracies.They were designed using the least number of dense layers(64) as anything more was very resource intensive especially when the repeatvector layers were increased .The training of transformer alone took around 3 days on a Nvidia Gtx 1650 Gpu.

The 2nd plot shows the difference in training and validation accuracy.The unaugmented models underfit with a difference between training and validation accuracy in the range 6.5% to 15.5.But once the data is augmented the models have more data to work with and hence the difference in accuracy reduces.**Transformer with attention and level 2 augmentation and hyperparameter tuning has the least difference in accuracy of 0.4** followed by the Gru and and Bilstm with no hyperparameter tuning both of which have an accuracy difference of 0.7

Deep Learning Model Training with GloVe: Global Vectors for Word Representation.

For details please refer to notebook capstone-nlp-DL-GlOve.ipynb

GloVe is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space.

We tried these Deep Learning Models after processing the dataset to use the GloVe Vectors . The glove dataset used was glove.840B.300d.txt.

We first used the sklearn train test split to split the dataset. We converted the dataset to GloVe embedded matrix.

Parameters used:

- EMBEDDING_FILE = 'embeddings/glove.840B.300d/glove.840B.300d.txt'
- MAX_SEQUENCE_LENGTH = 500
- EMBEDDING_DIM=300
- MAX_NB_WORDS=75000
- BATCH_SIZE = 32

The models trained, parameters used and results are below:

CNN:

Optimizer used: tf.keras.optimizers.SGD(lr=0.001, momentum=0.9)

```
Model: "model"
_____________________________________________________________________________________
_____________
Layer (type)                 Output Shape              Param #     Connected to
=====================================================================================
============
input_1 (InputLayer)         [(None, 500)]             0
_____________________________________________________________________________________
_____________
embedding (Embedding)        (None, 500, 300)          1353000     input_1[0][0]
_____________________________________________________________________________________
_____________
conv1d (Conv1D)              (None, 499, 128)          76928       embedding[0][0]
_____________________________________________________________________________________
_____________
conv1d_1 (Conv1D)            (None, 498, 128)          115328      embedding[0][0]
_____________________________________________________________________________________
_____________
conv1d_2 (Conv1D)            (None, 497, 128)          153728      embedding[0][0]
_____________________________________________________________________________________
_____________
conv1d_3 (Conv1D)            (None, 496, 128)          192128      embedding[0][0]
_____________________________________________________________________________________
_____________
conv1d_4 (Conv1D)            (None, 495, 128)          230528      embedding[0][0]
_____________________________________________________________________________________
_____________
max_pooling1d (MaxPooling1D) (None, 99, 128)           0           conv1d[0][0]
_____________________________________________________________________________________
_____________
max_pooling1d_1 (MaxPooling1D) (None, 99, 128)         0           conv1d_1[0][0]
_____________________________________________________________________________________
_____________
max_pooling1d_2 (MaxPooling1D) (None, 99, 128)         0           conv1d_2[0][0]
_____________________________________________________________________________________
_____________
max_pooling1d_3 (MaxPooling1D) (None, 99, 128)         0           conv1d_3[0][0]
```

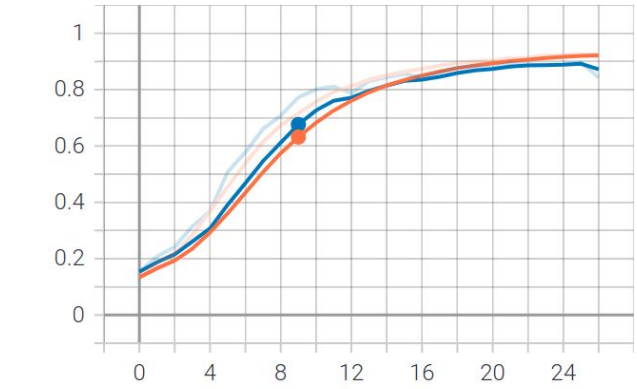
```
_____________________________________________________________________________
_____________
max_pooling1d_4 (MaxPooling1D)  (None, 99, 128)      0           conv1d_4[0][0]
_____________________________________________________________________________
_____________
concatenate (Concatenate)       (None, 495, 128)     0           max_pooling1d[0][0]
                                                                 max_pooling1d_1[0][0]
                                                                 max_pooling1d_2[0][0]
                                                                 max_pooling1d_3[0][0]
                                                                 max_pooling1d_4[0][0]
_____________________________________________________________________________
_____________
conv1d_5 (Conv1D)               (None, 491, 128)     82048       concatenate[0][0]
_____________________________________________________________________________
_____________
dropout (Dropout)               (None, 491, 128)     0           conv1d_5[0][0]
_____________________________________________________________________________
_____________
batch_normalization (BatchNorma (None, 491, 128)     512         dropout[0][0]
_____________________________________________________________________________
_____________
max_pooling1d_5 (MaxPooling1D)  (None, 98, 128)      0
batch_normalization[0][0]
_____________________________________________________________________________
_____________
conv1d_6 (Conv1D)               (None, 94, 128)      82048       max_pooling1d_5[0][0]
_____________________________________________________________________________
_____________
dropout_1 (Dropout)             (None, 94, 128)      0           conv1d_6[0][0]
_____________________________________________________________________________
_____________
batch_normalization_1 (BatchNor (None, 94, 128)      512         dropout_1[0][0]
_____________________________________________________________________________
_____________
max_pooling1d_6 (MaxPooling1D)  (None, 3, 128)       0
batch_normalization_1[0][0]
_____________________________________________________________________________
_____________
```

```
flatten (Flatten)                (None, 384)           0           max_pooling1d_6[0][0]
_____________________________________________________________________________________
_____________
dense (Dense)                    (None, 1024)          394240      flatten[0][0]
_____________________________________________________________________________________
_____________
dropout_2 (Dropout)              (None, 1024)          0           dense[0][0]
_____________________________________________________________________________________
_____________
dense_1 (Dense)                  (None, 512)           524800      dropout_2[0][0]
_____________________________________________________________________________________
_____________
dropout_3 (Dropout)              (None, 512)           0           dense_1[0][0]
_____________________________________________________________________________________
_____________
dense_2 (Dense)                  (None, 74)            37962       dropout_3[0][0]
=====================================================================================
============
Total params: 3,243,762
Trainable params: 3,243,250
Non-trainable params: 512
```

Results for CNN:

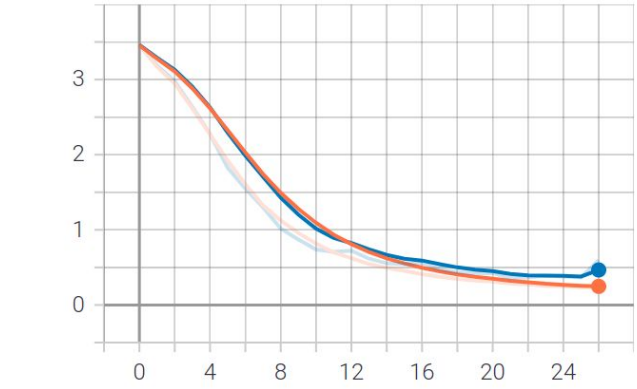
Orange - **training, blue validation**

epoch_accuracy

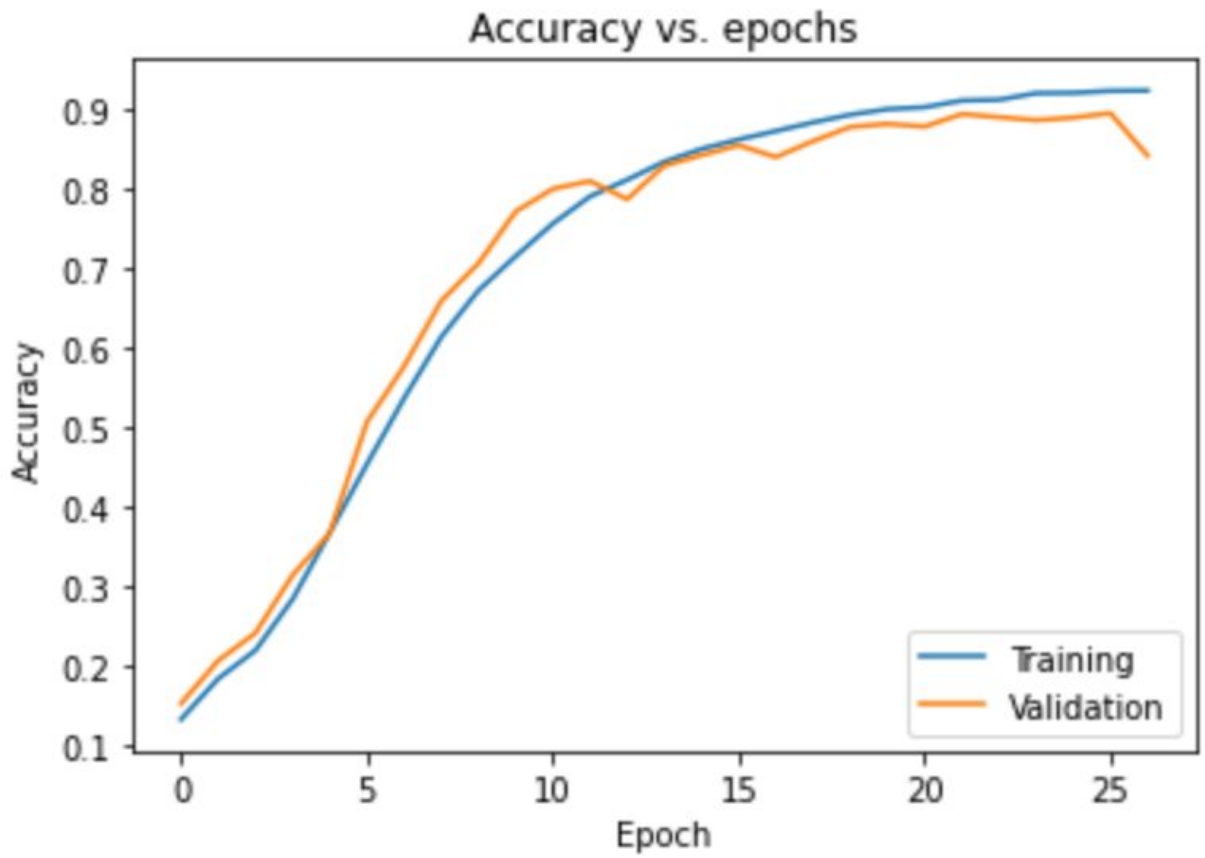


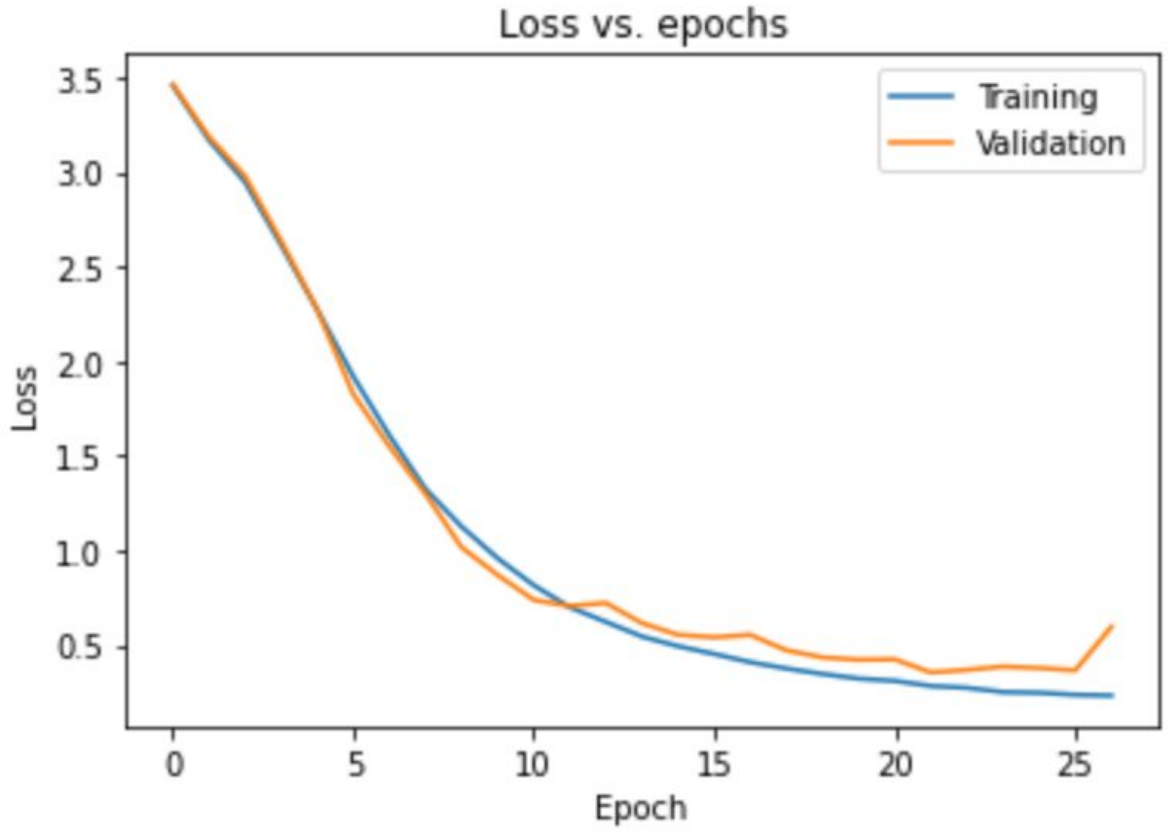
Name	Smoothed	Value	Step	Time	Relative
run_2020_12_15-10_15_48/train	0.6316	0.7164	9	Tue Dec 15, 10:18:17	2m 8s
run_2020_12_15-10_15_48/validation	0.6763	0.7726	9	Tue Dec 15, 10:18:17	2m 8s

epoch_loss



Unsmoothed plots of Accuracy & Loss for CNN:


Accuracy vs. epochs
Accuracy
Epoch
Training
Validation
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
5
10
15
20
25



Epoch 00025: saving model to checkpoints_best_only/checkpoint

Epoch 26/100

779/779 [==============================] - 15s 20ms/step - loss: 0.2281 - accuracy: 0.9276 - val_loss: 0.3667 - val_accuracy: 0.8964

Results at the end of the 27th epoch after which early stopping kicked in

Epoch 00026: saving model to checkpoints_best_only/checkpoint

Epoch 27/100

779/779 [==============================] - 15s 19ms/step - loss: 0.2239 - **accuracy: 0.9291** - **val_loss: 0.5970** - **val_accuracy: 0.8429**

val/train: 2.56

Epoch 00027: saving model to checkpoints_best_only/checkpoint

Epoch 00027: early stopping

Precision: 0.857542

Recall: 0.842942

F1 score: 0.838929

Evaluation Accuracy: 0.843

Evaluation Loss: 0.597

GRU:

Optimizer used: tf.keras.optimizers.SGD(lr=0.001, momentum=0.9)

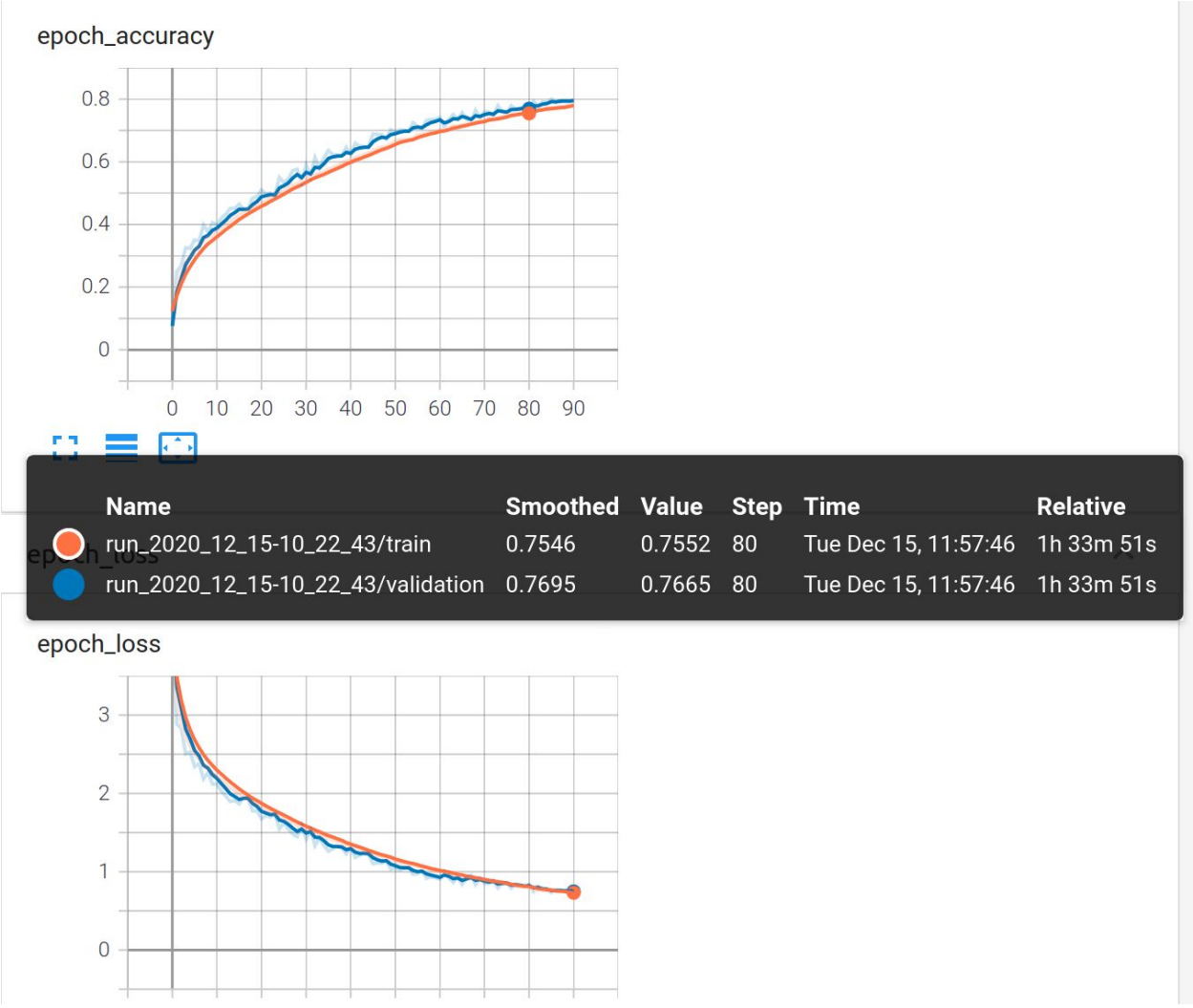
GRU Model Summary:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 500, 300)          1353000
_________________________________________________________________
cu_dnngru (CuDNNGRU)         (None, 500, 32)           32064
_________________________________________________________________
dropout (Dropout)            (None, 500, 32)           0
_________________________________________________________________
batch_normalization (BatchNo (None, 500, 32)           128
_________________________________________________________________
cu_dnngru_1 (CuDNNGRU)       (None, 500, 32)           6336
_________________________________________________________________
dropout_1 (Dropout)          (None, 500, 32)           0
_________________________________________________________________
```

```
batch_normalization_1 (Batch (None, 500, 32)           128
_________________________________________________________________
cu_dnngru_2 (CuDNNGRU)       (None, 500, 32)           6336
_________________________________________________________________
dropout_2 (Dropout)          (None, 500, 32)           0
_________________________________________________________________
batch_normalization_2 (Batch (None, 500, 32)           128
_________________________________________________________________
cu_dnngru_3 (CuDNNGRU)       (None, 500, 32)           6336
_________________________________________________________________
dropout_3 (Dropout)          (None, 500, 32)           0
_________________________________________________________________
batch_normalization_3 (Batch (None, 500, 32)           128
_________________________________________________________________
cu_dnngru_4 (CuDNNGRU)       (None, 500, 32)           6336
_________________________________________________________________
dropout_4 (Dropout)          (None, 500, 32)           0
_________________________________________________________________
batch_normalization_4 (Batch (None, 500, 32)           128
_________________________________________________________________
cu_dnngru_5 (CuDNNGRU)       (None, 500, 32)           6336
_________________________________________________________________
dropout_5 (Dropout)          (None, 500, 32)           0
_________________________________________________________________
batch_normalization_5 (Batch (None, 500, 32)           128
_________________________________________________________________
cu_dnngru_6 (CuDNNGRU)       (None, 32)                6336
_________________________________________________________________
dropout_6 (Dropout)          (None, 32)                0
_________________________________________________________________
batch_normalization_6 (Batch (None, 32)                128
_________________________________________________________________
dense (Dense)                (None, 256)               8448
_________________________________________________________________
```

```
batch_normalization_7 (Batch (None, 256)               1024
_________________________________________________________________
dense_1 (Dense)              (None, 74)                19018
=================================================================
Total params: 1,452,466
Trainable params: 1,451,506
Non-trainable params: 960
```

Results for GRU:



Epoch 00090: saving model to checkpoints_best_only/checkpoint

Epoch 91/100

779/779 [==============================] - 76s 97ms/step - loss: 0.6893 - accuracy: 0.7912 - val_loss: 0.7361 - val_accuracy: 0.7978

val/train: 1.03

Epoch 00091: saving model to checkpoints_best_only/checkpoint

Epoch 00091: early stopping

Evaluation Accuracy: 0.798

Evaluation Loss: 0.736

Mean Accuracy for the validation dataset:

0.6217235459403677

Mean Loss for the validation dataset:

1.3452635348498165

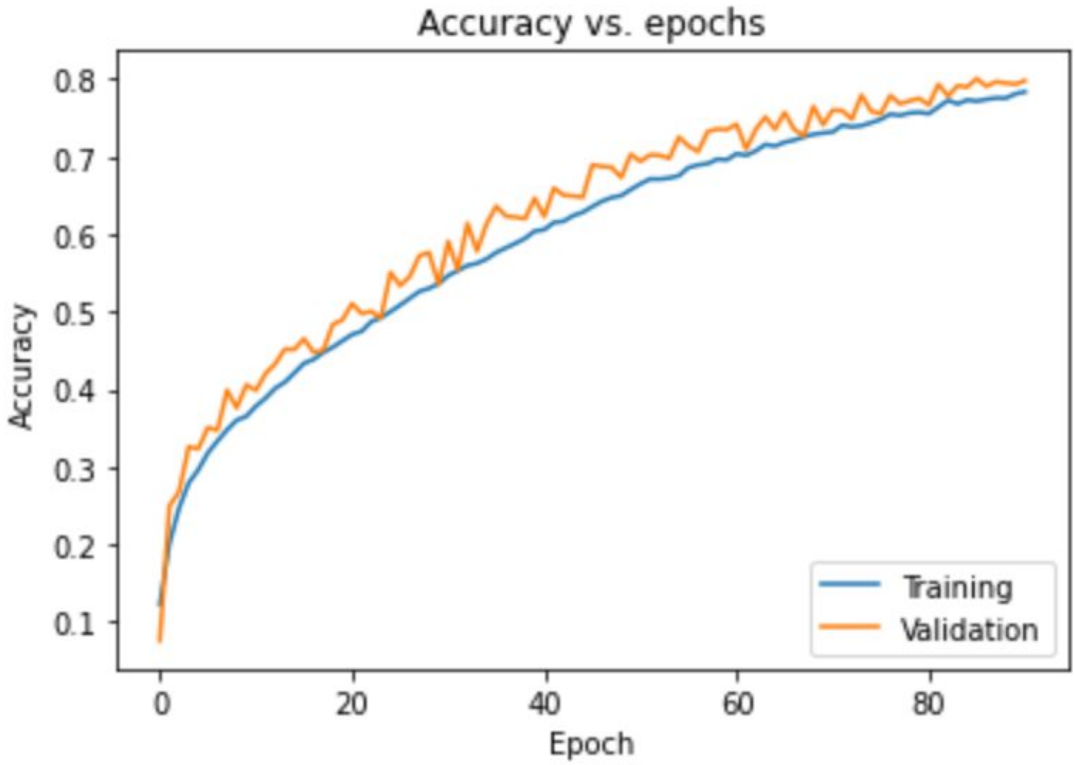
Mean Accuracy for the training dataset:

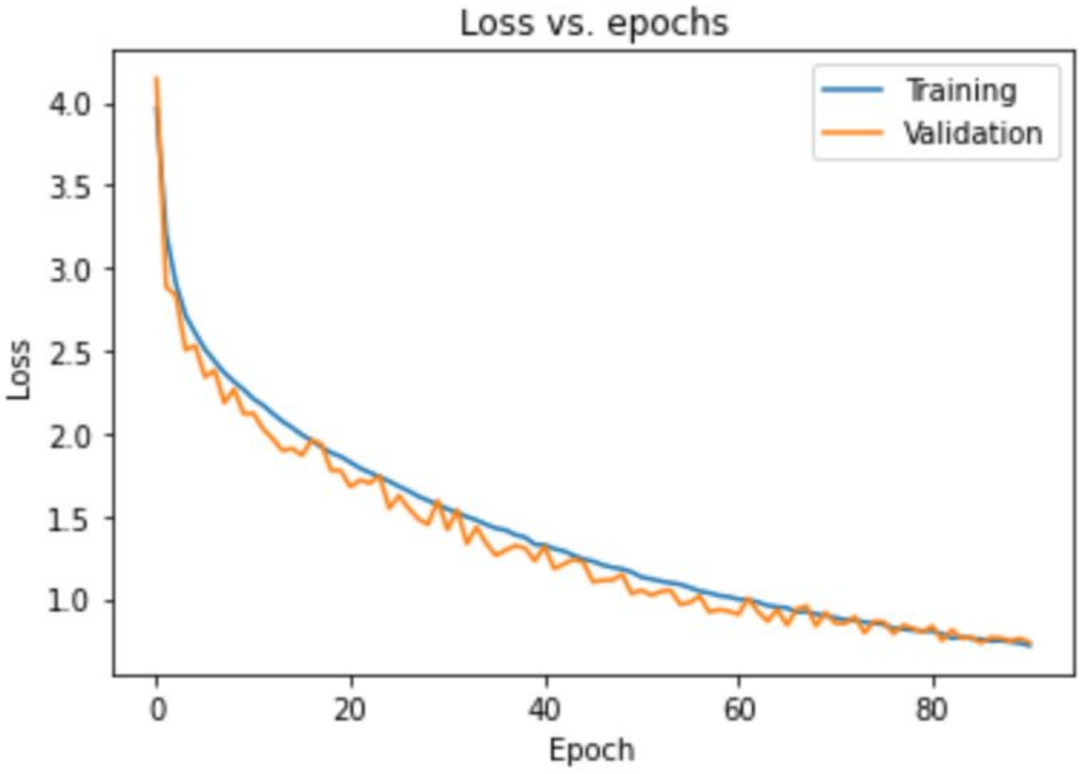
0.5949455665854307

Mean Loss for the training dataset:

1.4078370489916958

Unsmoothed Plots of Accuracy & Losses:





How to improve your model performance?

Please note : the model runs above incorporate the results from the Data Augmentation techniques outlined below.

As noted earlier, our first model run with Logistic Regression using the preprocessed data could ONLY score 62% accuracy.

It became pretty clear immediately, we had to do something to improve the scores.

We researched on Data Augmentation techniques and some pointers provided by our mentor proved very useful. We researched these techniques:

a. Translation based Data Augmentation - we pick each ticket , translate it into random language and translate it back to English. While this was powerful, the amount of time it took was huge and it did NOT provide the volume of data needed for our scores to make a difference.

   For details about translation based Data Augmentation please refer to notebook: capstone-nlp-Data-Augmentation-by-Translation.ipynb

b. Spacy based technique - we used spacy to find synonyms of words. The technique is to substitute random words in the ticket with their synonyms. This again took too much time for spacy to generate synonyms, so we had to abandon it. We think the above provide very good potential, but given the time we had we had to find another technique. For details about translation based Data Augmentation please refer to notebook: capstone-nlp-Data-Augmentation-by-spaCy.ipynb

c. We used KNN to find the synonyms - These are technically not synonyms but the words are very close based on their ‘distance’ from the candidate word for which we are finding the synonym.

Examples:

'reset': ['disable', 'restart', 'manually', 'disconnect'],

'issue': ['problem', 'question', 'concern', 'concerned'],

'unable': ['cannot', 'able', 'failing', 'however'],

'you': ['sure', 'want', 'can', 'know'],

'have': ['they', 'could', 'already', 'would'],

'user': ['login', 'automatically', 'functionality', 'interface'],

'my': ['me', 'myself', 'own', 'got'],

'error': ['incorrect', 'invalid', 'problem', 'exception'],

'access': ['accessible', 'allow', 'provide', 'secure'],

'hello': ['hi', 'hey', 'yes', 'dear'],

'be': ['should', 'being', 'not', 'will'],
'account': ['payment', 'personal', 'same', 'however'],

d. We also used Parts of Speech tagging in a similar way to identify synonyms.

Examples:

'trying': ['try'],

'working': ['work'],

'home': ['house'],

'unlocking': ['unlock'],

'locked': ['lock'],

'getting': ['get'],

'need': ['want'],

'dynamics': ['dynamic'],

'regarding': ['regard'],

'phone': ['telephone'],

'client': ['customer'],

'duplication': ['duplicate'],

'gentle': ['soft'],
'display': ['showing'],

For the detailed code for synonyms using KNN & POS tagging, please refer to the notebook:

capstone-nlp-Data-Augmentation-by-Word-Embeddings.ipynb

Because using KNN technique, we could find multiple synonyms for a word, we settled on using the synonyms from KNN to create Tickets where we subtitied a random word in the ticket with its synonym keeping the label the same.

For example, in the below sample, the word accessible was changed to reachable, so the same meaning is retained yet it helped us create a new ticket for the same group.

accessible hand language icon customer number telephone	GRP_3
reachable hand language explorer customer number telephone	GRP_3

We also excluded the majority group GRP_0 and concentrated on increasing the number of records for the minority groups.

Fig 12 - Distribution before Augmentation

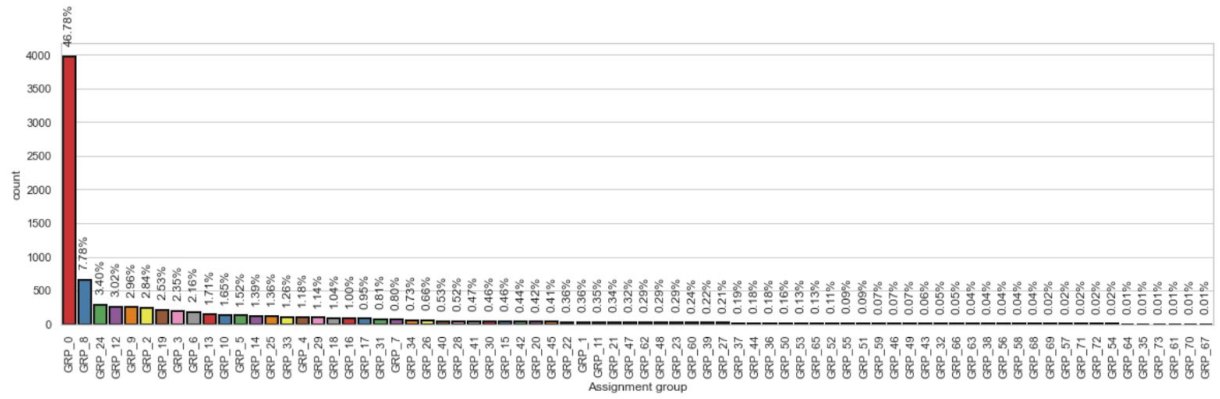
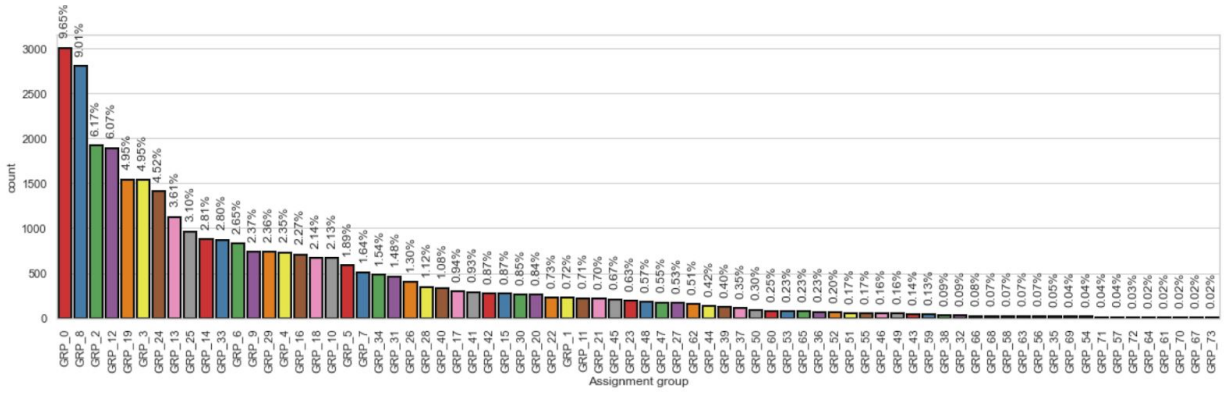


Fig 13 - Distribution after Augmentation



As the graphs indicate, we were able to significantly increase the representation of the minority groups. The initial results are promising. A sample classification report below shows that there is test data in every group:

	precision	recall	f1-score	support
0	0.63	0.71	0.67	919
1	0.85	0.78	0.81	64
2	0.94	0.88	0.91	193
3	0.96	0.87	0.91	78
4	0.84	0.84	0.84	594
5	0.95	0.93	0.94	327
6	0.91	0.84	0.87	242
7	1.00	0.87	0.93	84

8	0.92	0.93	0.92	224
9	0.99	0.95	0.97	73
10	0.97	0.94	0.96	186
11	0.84	0.84	0.84	447
12	0.86	0.90	0.88	585
13	1.00	0.95	0.97	78
14	0.95	0.93	0.94	61
15	0.99	0.88	0.93	75
16	0.88	0.91	0.89	55
17	0.80	0.90	0.85	433
18	0.92	0.97	0.95	265
19	0.94	0.81	0.87	113
20	0.90	0.84	0.87	45
21	0.96	0.78	0.86	105
22	0.97	0.89	0.93	236
23	0.83	0.86	0.84	469
24	0.94	0.82	0.88	83
25	0.83	0.74	0.78	148
26	1.00	0.80	0.89	10
27	0.92	0.87	0.90	269
28	0.85	0.85	0.85	142
29	1.00	1.00	1.00	2
30	0.89	0.63	0.74	27
31	1.00	0.91	0.96	35
32	1.00	1.00	1.00	10
33	0.67	0.59	0.62	34
34	0.92	0.86	0.89	237
35	0.96	0.93	0.94	97
36	0.96	0.95	0.96	79

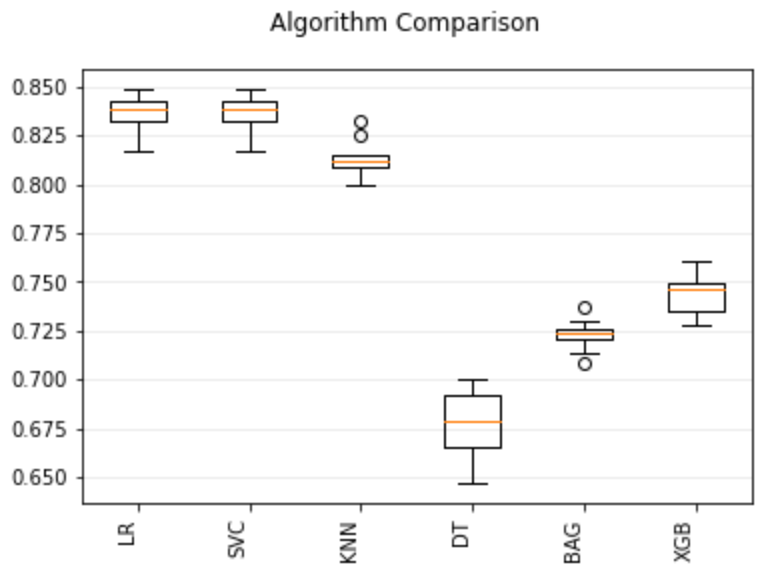
37	0.95	0.93	0.94	87
38	1.00	0.92	0.96	13
39	1.00	0.92	0.96	26
40	0.96	0.82	0.89	66
41	1.00	1.00	1.00	16
42	0.69	0.48	0.56	42
43	0.96	0.89	0.93	57
44	1.00	0.92	0.96	12
45	0.62	0.61	0.61	155
46	0.85	0.74	0.79	31
47	1.00	0.94	0.97	16
48	1.00	0.88	0.93	16
49	0.89	0.70	0.78	23
50	1.00	1.00	1.00	2
51	0.81	1.00	0.90	13
52	0.75	0.60	0.67	5
53	1.00	0.40	0.57	5
54	1.00	0.75	0.86	4
55	1.00	0.93	0.96	14
56	0.68	0.81	0.74	241
57	0.58	0.32	0.41	22
58	1.00	1.00	1.00	1
59	0.95	0.76	0.84	46
60	1.00	0.75	0.86	4
61	1.00	1.00	1.00	2
62	0.90	0.95	0.93	20
63	0.80	1.00	0.89	4
64	1.00	1.00	1.00	1
65	1.00	0.40	0.57	5

66	1.00	0.50	0.67	6
67	0.96	0.95	0.96	151
68	1.00	1.00	1.00	2
69	1.00	0.67	0.80	3
70	0.00	0.00	0.00	1
71	1.00	1.00	1.00	2
72	0.78	0.85	0.82	852
73	0.80	0.64	0.71	250
accuracy			0.84	9340
macro avg	0.90	0.82	0.85	9340
weighted avg	0.85	0.84	0.84	9340

	Algorithm	Accuracy	f1	Precision Score	Recall Score
1	SVC 30% Test Data after Data Augmentation Word Embedding Round 2	91.43%	91.44%	94.37%	89.24%
2	Stacker 30% Test Data after Data Augmentation Word Embedding Round 2	90.50%	90.46%	93.74%	88.29%
3	KNN 30% Test Data after Data Augmentation Word Embedding Round 2	88.81%	88.70%	89.36%	90.07%
4	XGBoost Classifier 30% Test Data after Data Augmentation Word Embedding Round 2	84.06%	84.16%	90.65%	83.31%
5	Bagging Classifier 30% Test Data after Data Augmentation Word Embedding Round 2	83.94%	83.94%	92.17%	84.95%
6	DT 30% Test Data after Data Augmentation Word Embedding Round 2	78.37%	78.35%	82.03%	79.56%
7	LR 30% Test Data after Data Augmentation Word Embedding Round 2	76.48%	76.10%	78.98%	76.77%

We were able to increase the Accuracy of Logistics Regression from 62% to 76.48% after a couple of rounds of Data Augmentation. SVC topped the charts with 91.43 %.

We also ran the ensembles but their results were not too impressive. We will need to look into the reasons. Attached box plot shows the different Machine Learning Algorithms. The SVC & LR score the highest



Milestone - 1

a. EDA - Explore and understand the dataset provided
   i. Visualizing different patterns
   ii. Visualizing different text features
   iii. Identify data discrepancies
b. Text preprocessing
   i. Dealing with missing values
   ii. Text Translation
c. Explore ways to augment the dataset provided without losing the relationship to the target label
d. Creating word vocabulary from the corpus of report text data
e. Creating tokens as required

Our Approach for Milestone-1

EDA:

- There are 8500 records in the dataset
- Each Dataset contains 4 columns
- The column 'Caller' seems to contain only junk. So we dropped the column
- We identified junk characters using ftfy library
- There were very few columns with Nulls

An example of data that looked like junk at an initial glance. But on applying the ‘fixes text for you - ftfy’ library, we discovered that the data is actually Ticket Description in Chinese Simplifed

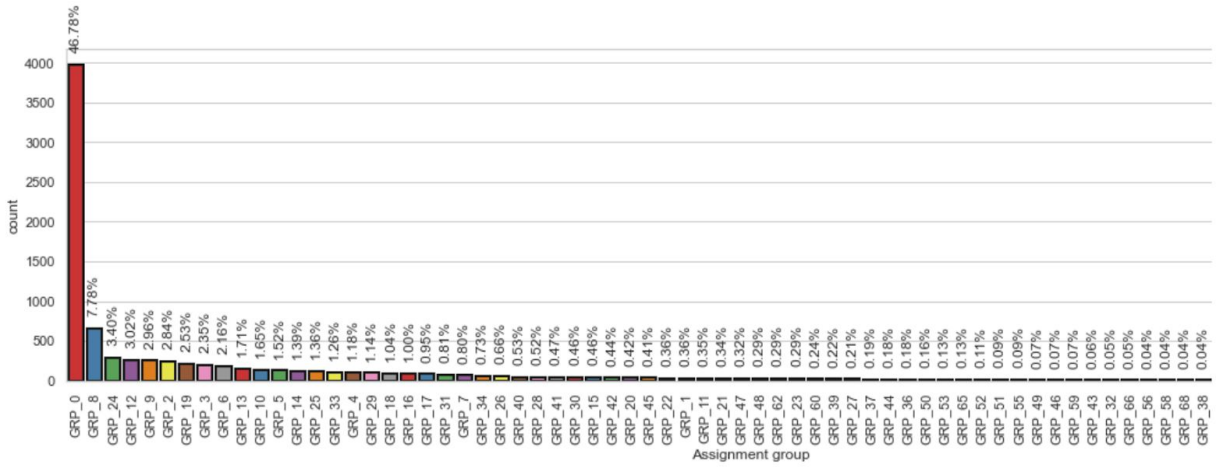
Junk text: **ç”µè„‘å¡ä¸”è¿žä¸ä¸Šå†…ç½‘**

Fixed text: **电脑卡且连不上内网**

We discovered this kind of brokenness is called Mojibake

- when someone has encoded Unicode with one standard and decoded it with a different one. This often shows up as characters that turn into nonsense sequences (called “mojibake”)

Fig 1. Distribution of Tickets amongst various groups:



As the figure 1 shows, Group 0 has the most number of tickets, followed by Group 8. The brief (problem statement) provided indicated that about 54% are resolved by L1/L2 groups. Since we also find from the above graph GRP_0 + GRP_8 also equals 54% roughly, we guess GRP_0 is L1 and GRP_8 is probably L2. The remaining groups are probably Functional/L3 teams. (The word cloud for GRP_8 also indicate this group could be related to monitoring tool OR job scheduler)

Some other interesting Figures are below. Fig2 shows the Assignment groups with the highest number of tickets. Figure 3 shows the groups with the lowest number of tickets. Some groups have only 1 ticket. We can choose to discard these groups. BUT, we have chosen a different approach - which we will describe more in the Data Augmentation section. We have done our best NOT to delete any data in the dataset provided. Figure 4 shows a pie chart % distribution by assignment group. Figures 6a, 6b, 6b show the Word Cloud.

Fig 2.

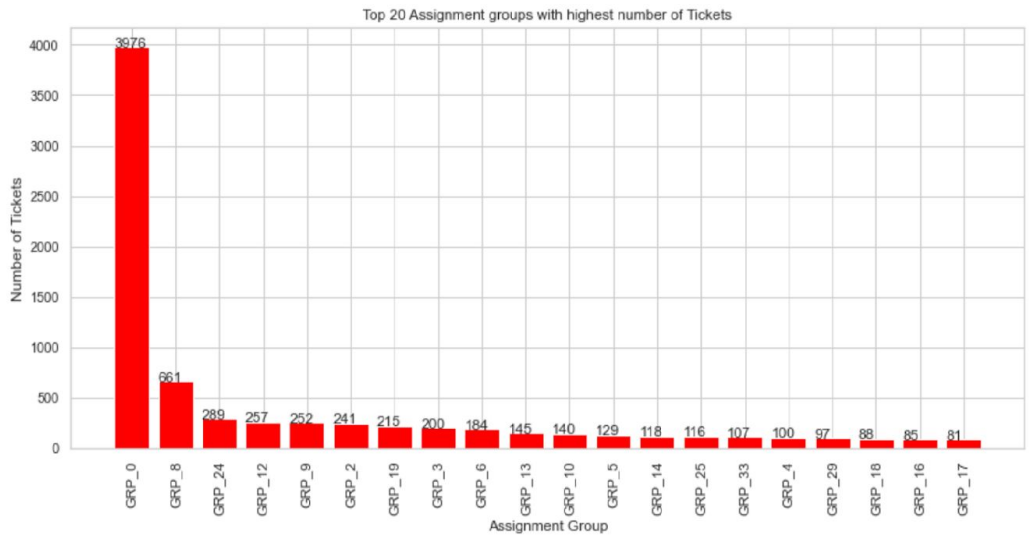


Fig 3.

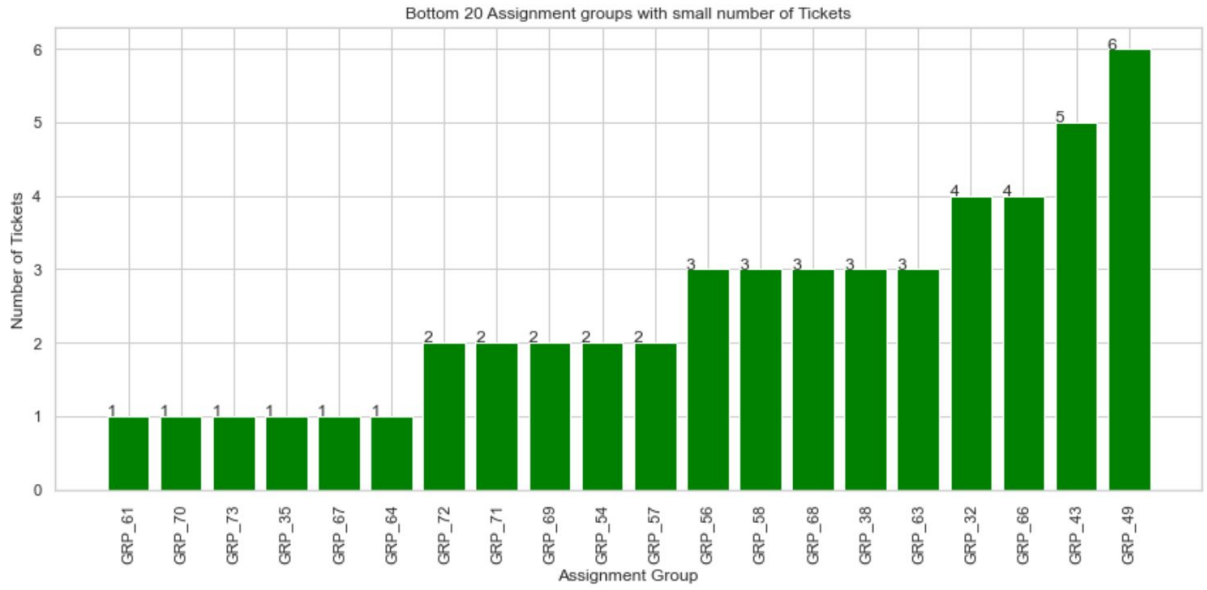


Fig 4.

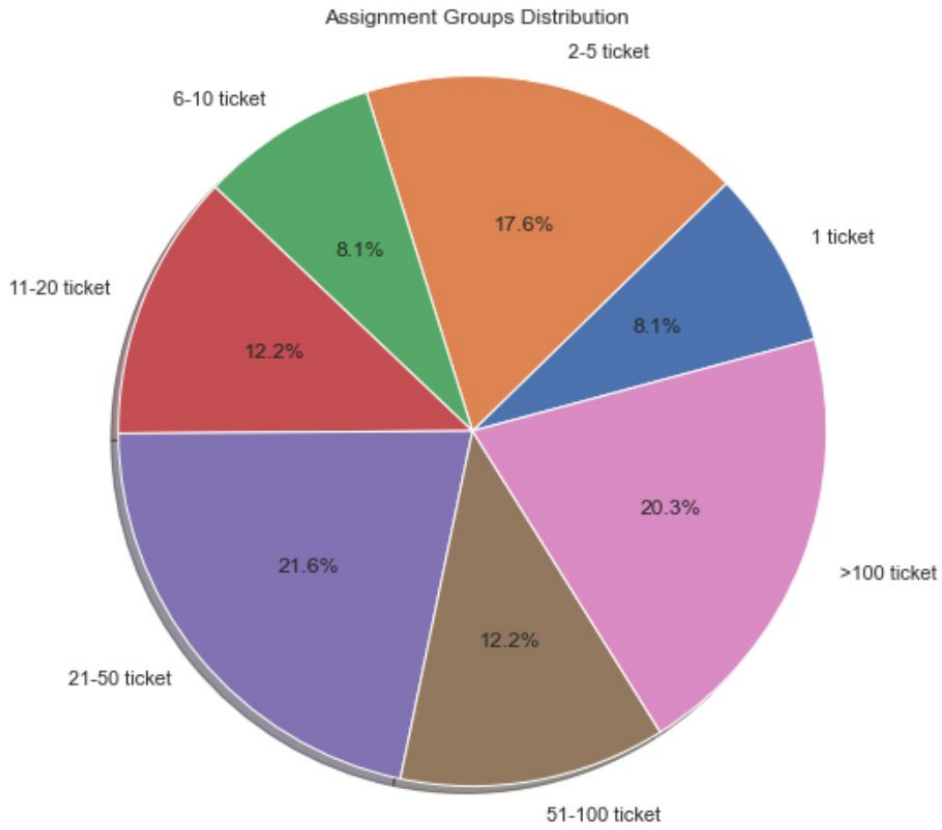


Fig 6a - Word Clouds for GRP_0

- These ticket descriptions indicate issues typically resolved by Help Desk using Standard Operating Procedures

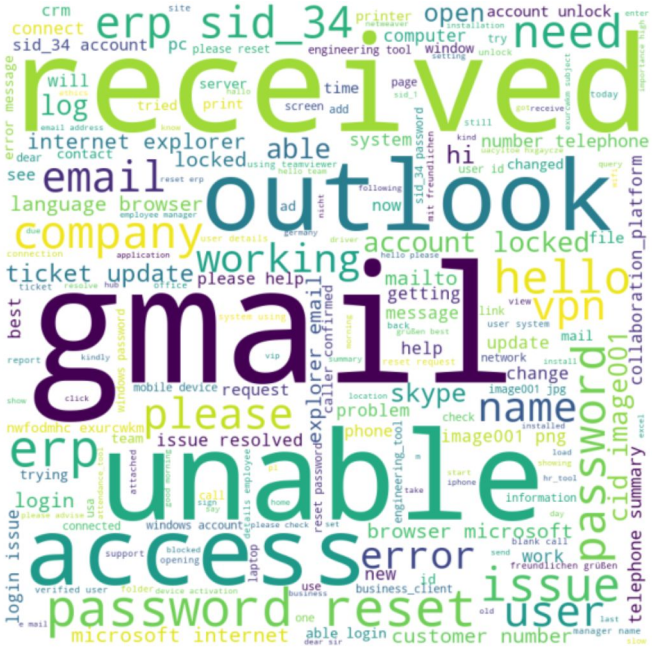


Fig 6b - Word Clouds for GRP_8

The words indicate issues reported by Monitoring tool OR tools like Job Scheduler

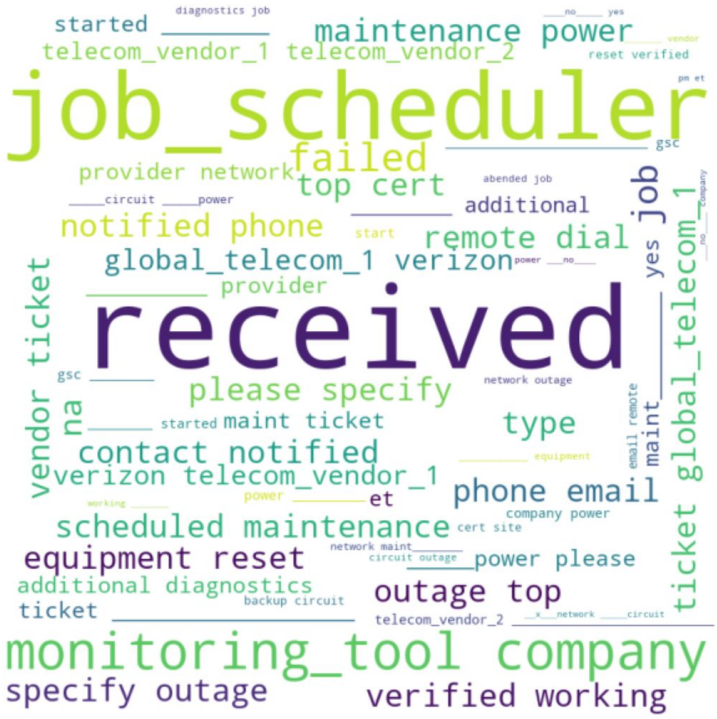
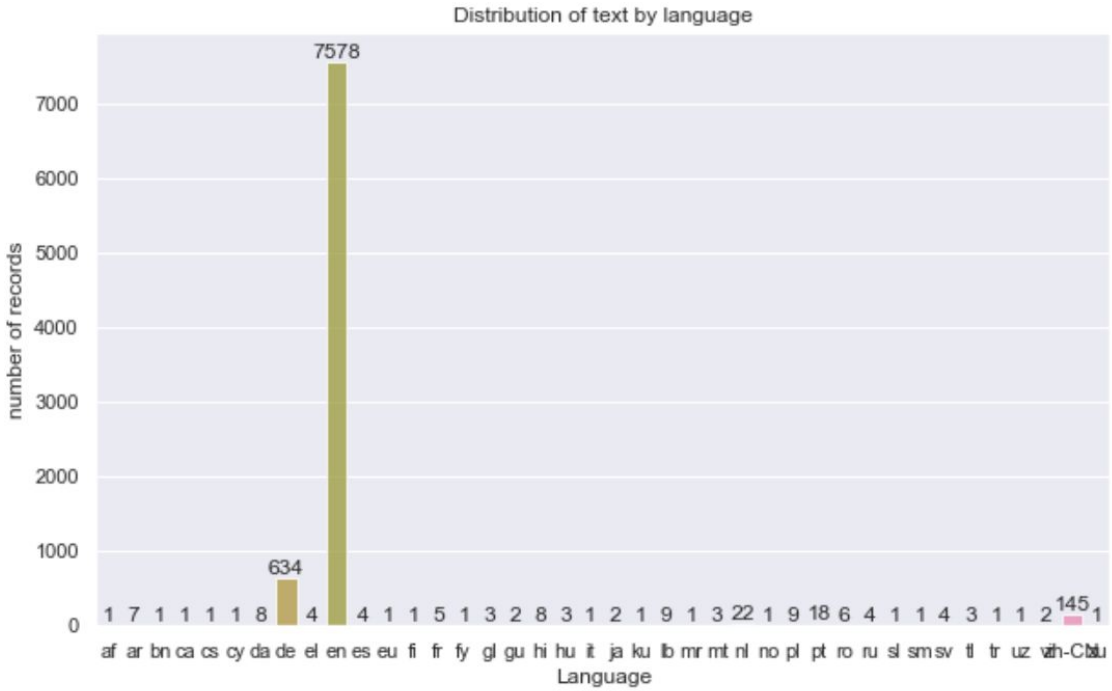


Fig 6c Word Cloud for GRP_24

GRP_24 Word Cloud led us into looking into Tickets with languages other than English. We used the Google Translation Engine to look into this further

As Fig 7 shows, most of the Tickets are in English followed by German and Simplified Chinese.



We also found in some tickets, the Short Description and Description were in different languages. An example below where the Description is in english and Shirt Description is in Portuguese

Fig 8.

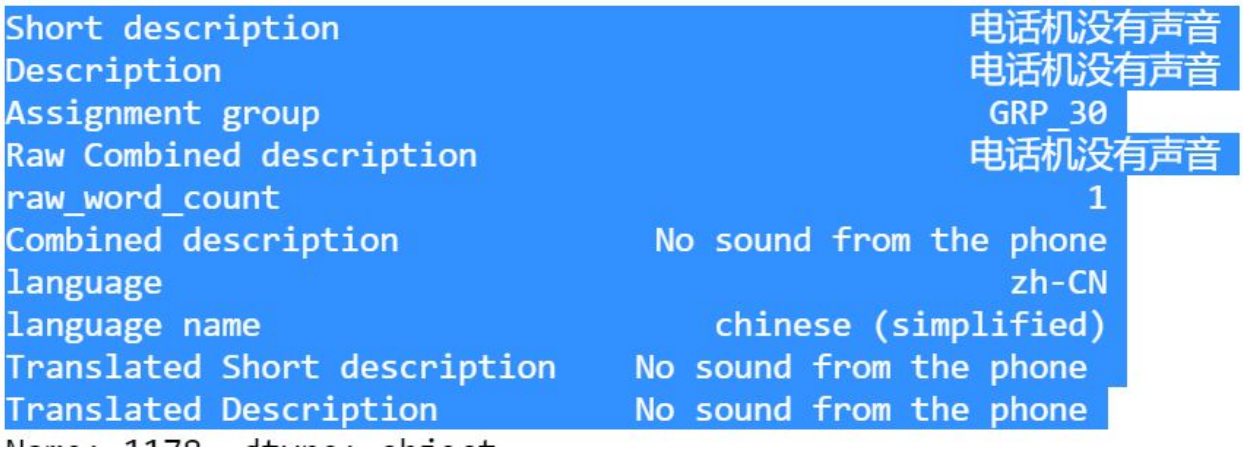
	Short description	Description	Assignment group	Raw Combined description	raw_word_count	Combined description	language	language name	Translated Short description	
8498	machine não está funcionando	i am unable to access the machine utilities to...	GRP_62	machine não está funcionando i am unable to ac...	18	machine is not working i am unable to access t...	pt	portuguese	machine is not working	i am ur

For more details on the EDA, please review this notebook:
capstone_nlp_merged_EDA_Preprocessing_Translation_v13.ipynb

Our approach to Pre Processing

So we decided to translate the Short Description and Description fields separately first and then combine them for further preprocessing.

Fig 8 shows the Translated Short Description and Description field of a sample.



Short description	电话机没有声音
Description	电话机没有声音
Assignment group	GRP_30
Raw Combined description	电话机没有声音
raw_word_count	1
Combined description	No sound from the phone
language	zh-CN
language name	chinese (simplified)
Translated Short description	No sound from the phone
Translated Description	No sound from the phone

For translation we used the Google Translator API in multithreading mode which performed the translation in a few seconds (all 8500 records)

We also found a couple of records where the Translator got confused so we fixed the data manually.

Fig 9 - In the sample below, the Translator mistook the sentences as Greek. We had to manually fix the data.

	Short description	Description	Assignment group	Raw Combined description	raw_word_count	Combined description	language	language name	Translated Short description	Translated Description
8043	setup new ws \xaqzisrk ahbgjrqz	setup new ws \xaqzisrk ahbgjrqz	GRP_24	setup new ws \xaqzisrk ahbgjrqz	5		el	greek	σετύπ νέω ως \ χάκζησρκ αχβγξρκζ	σετύπ νέω ως \ χάκζησρκ αχβγξρκζ
8072	setup new ws \pnwbkitv phbnwmkl	setup new ws \pnwbkitv phbnwmkl	GRP_24	setup new ws \pnwbkitv phbnwmkl	5		el	greek	σετύπ νέω ως \ πνωβκίτυ φβνωμκλ	σετύπ νέω ως \ πνωβκίτυ φβνωμκλ

We also fixed about 17 other records manually .

Using the ftfy library, we fixed the junk characters as noted above.

There were a couple of null records with null values for Short Description and Description. We fixed those to empty spaces.

After the translation, we combined the Short Description and Description fields and then applied the logic to remove special characters, numbers and things like email addresses .

For more details on the Preprocessing, please review this notebook:
capstone_nlp_merged_EDA_Preprocessing_Translation_v13.ipynb

Dataset for Deep Learning

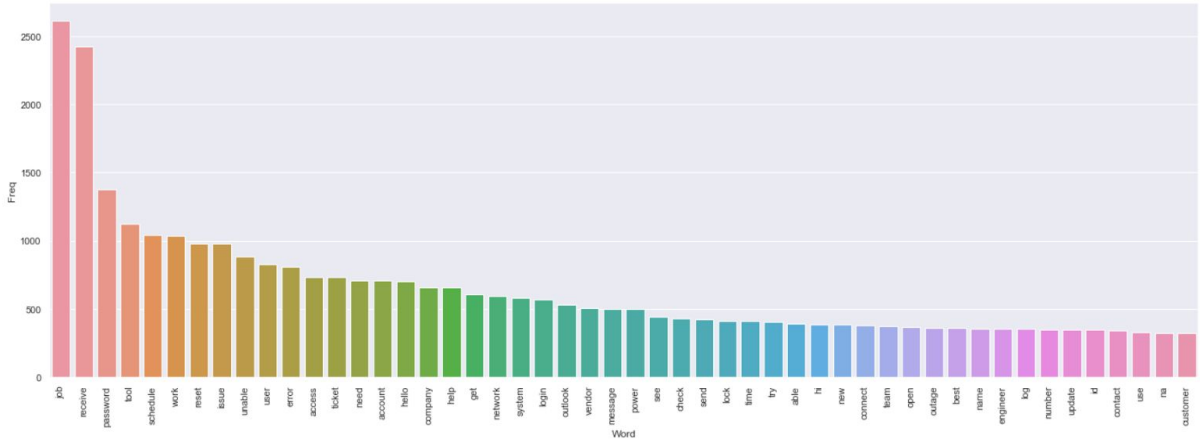
The preprocess is still NOT complete, but, at this point, we created the dataset for Deep Learning. The reason we did this is because we wanted to retain the stop words and ‘un lemmatized’ words so the Deep Learning Algorithms will be able to process them in a context sensitive manner.

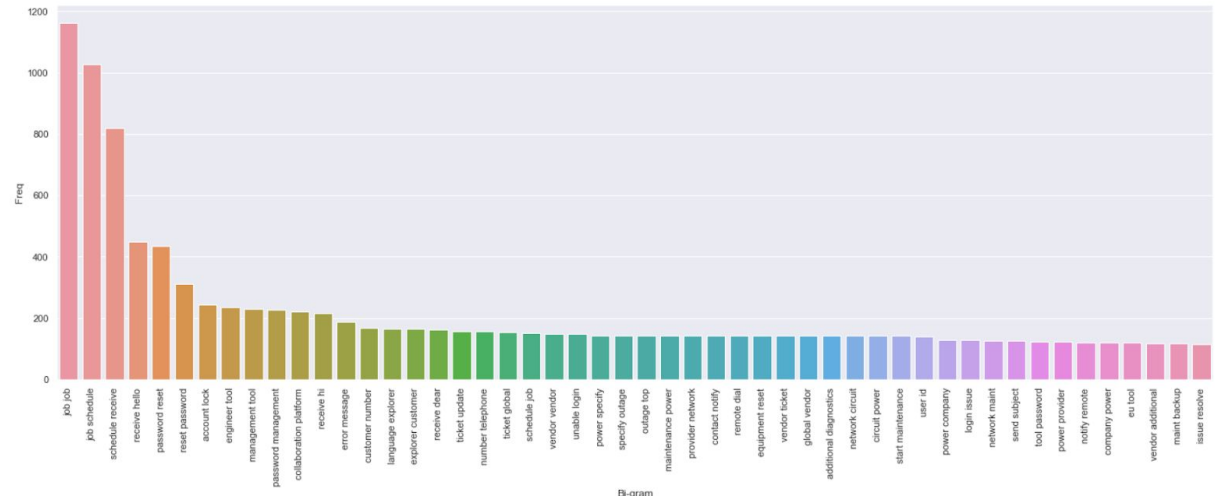
EDA (cont’d)

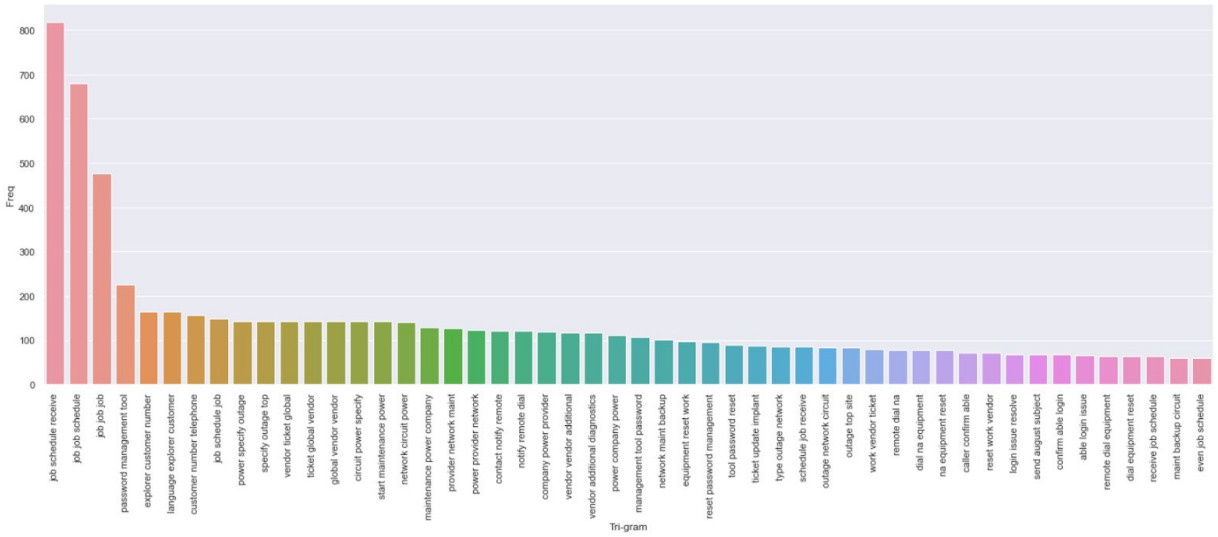
We then used the NLTK library for stopwords in English. We also used punkt and wordnet for lemmatization .

Total Corpus Word Count before lemmatization: 137508
Total Corpus Word Count after lemmatization: 82245

Fig 10. **Top 20 uni-grams, bi-grams & tri-grams**







Document Clustering

We also attempted to use the K-Means algorithm to do Document clustering.

Why is document clustering an important tool in NLP?

Document Clustering is a process of grouping similar items together. Each group, also called as a cluster, contains items that are similar to each other. Clustering algorithms are unsupervised learning algorithms i.e. we do not need to have labelled datasets. There are many clustering algorithms for clustering including KMeans, DBSCAN, Spectral clustering, hierarchical clustering etc and they have their own advantages and disadvantages. The choice of the algorithm mainly depends on whether or not you already know how many clusters to create. Some algorithms such as KMeans need you to specify the number of clusters to create whereas DBSCAN does

not need you to specify. Another consideration is whether you need the trained model to be able to predict clusters for unseen dataset. KMeans can be used to predict the clusters for new dataset whereas DBSCAN cannot be used for new dataset.

First we decided on the optimal k - we found the optimal k to be 4 as shown by the ‘elbow method’

These are the clusters:

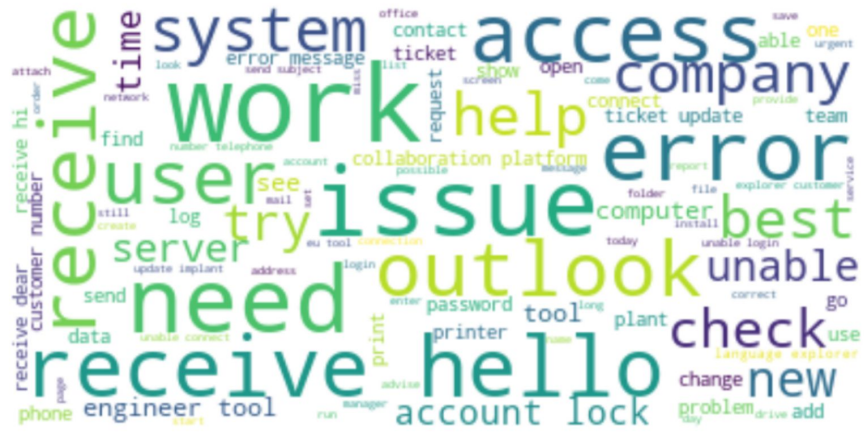
Cluster:0 contains words like outlook, unable, etc

Cluster: 1 contains words like vendor, outage, etc

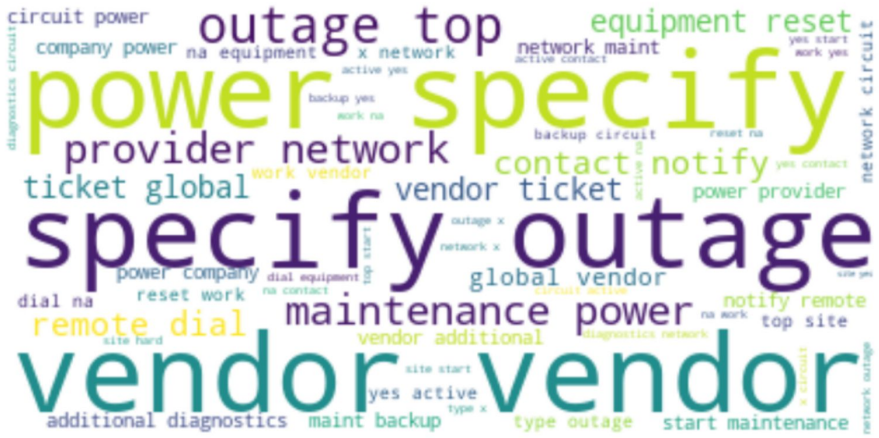
Cluster: 2 contains words related password management and reset

Cluster: 3 contains words related to jon schedule

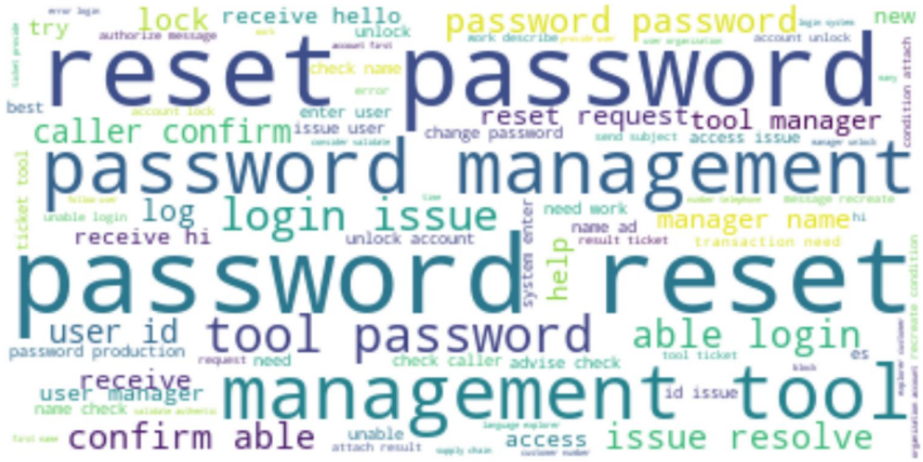
Cluster: 0



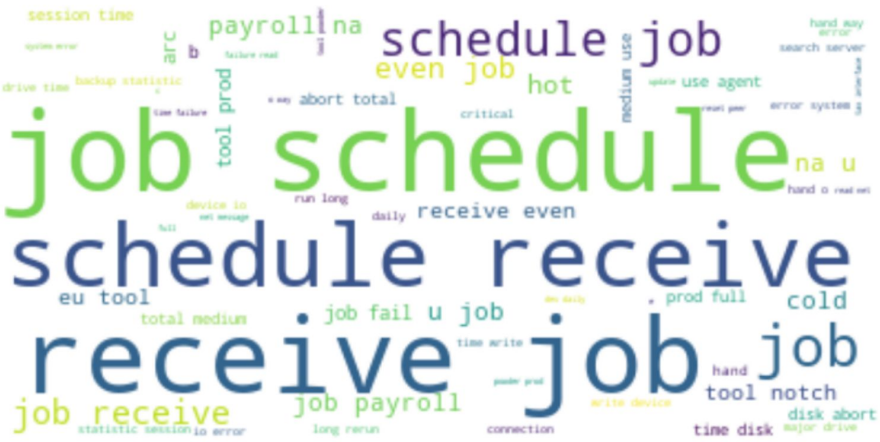
Cluster: 1



Cluster: 2



Cluster: 3



Topic Modeling (aka Tagging)

Topic modeling is an interesting problem in NLP applications where we want to get an idea of what topics we have in our dataset. A topic is nothing more than a collection of words that describe the overall theme. For example, in case of news articles, we might think of topics as politics, sports etc. but topic modeling won't directly give you names of the topics but rather a set of most probable words that might describe a topic. It is up to us to determine what topic the set of words might refer to

In our dataset, we identified these words highlighted that seem to suggest the most important topics are job schedule, password resets, account locks, login issues, vendor issues.

0 job schedule receive hot cold payroll na notch prod tool eu fail arc hand way

1 password reset management tool request user manager unlock production change receive es need login id

2 account lock unlock ad user lockout id check login issue hello frequent need team receive

3 unable login outlook issue connect tool access user work error open engineer receive help log

4 ticket update implant vendor power network outage circuit na yes work site global company active

At a later point, we may use these techniques to generate more training data if needed.

Pre Processing (cont'd)

We used the WordNetLemmatizer to lemmatize the corpus

After lemmatization and word - gram analysis we created the dataset required for the Machine Learning Models.

We also applied spell check and Contraction logic (for example: change "there'd've" to "there would have")

Finally we build the processed dataset required for Machine Learning Models.

For more details on the Preprocessing, please review this notebook:
capstone_nlp_merged_EDA_Preprocessing_Translation_v13.ipynb

Our final numbers after preprocessing

```
In [133]: 1 mydata.isna().apply(pd.value_counts)
```

Out[133]:

	Short description	Description	Assignment group	Raw Combined description	raw_word_count	Combined description	language	language name	Translated Short description	Translated Description	CombinedWordCount	D
False	8500	8500	8500	8500	8500	8500	8500	8500	8500	8500	8500	

After preprocessing, we ran an initial Model with Logistic Regression. It scored about 62% in accuracy and an f1 score of 57%

Fig 11.

	Algorithm	Accuracy	f1	Precision Score	Recall Score
1	LR 30% Raw Test Data after Cleansing	62.12%	56.91%	61.34%	67.72%