Capstone Project - FINAL REPORT

26 April, 2020

AIML Online May19 Group7A

Team

Amit Kumar Gupta

Deepali Chandratre Godbole

Leela Desai

Micheal Philomine Raja

Poonam Kushwaha

Uma Satya Vani Nagandala

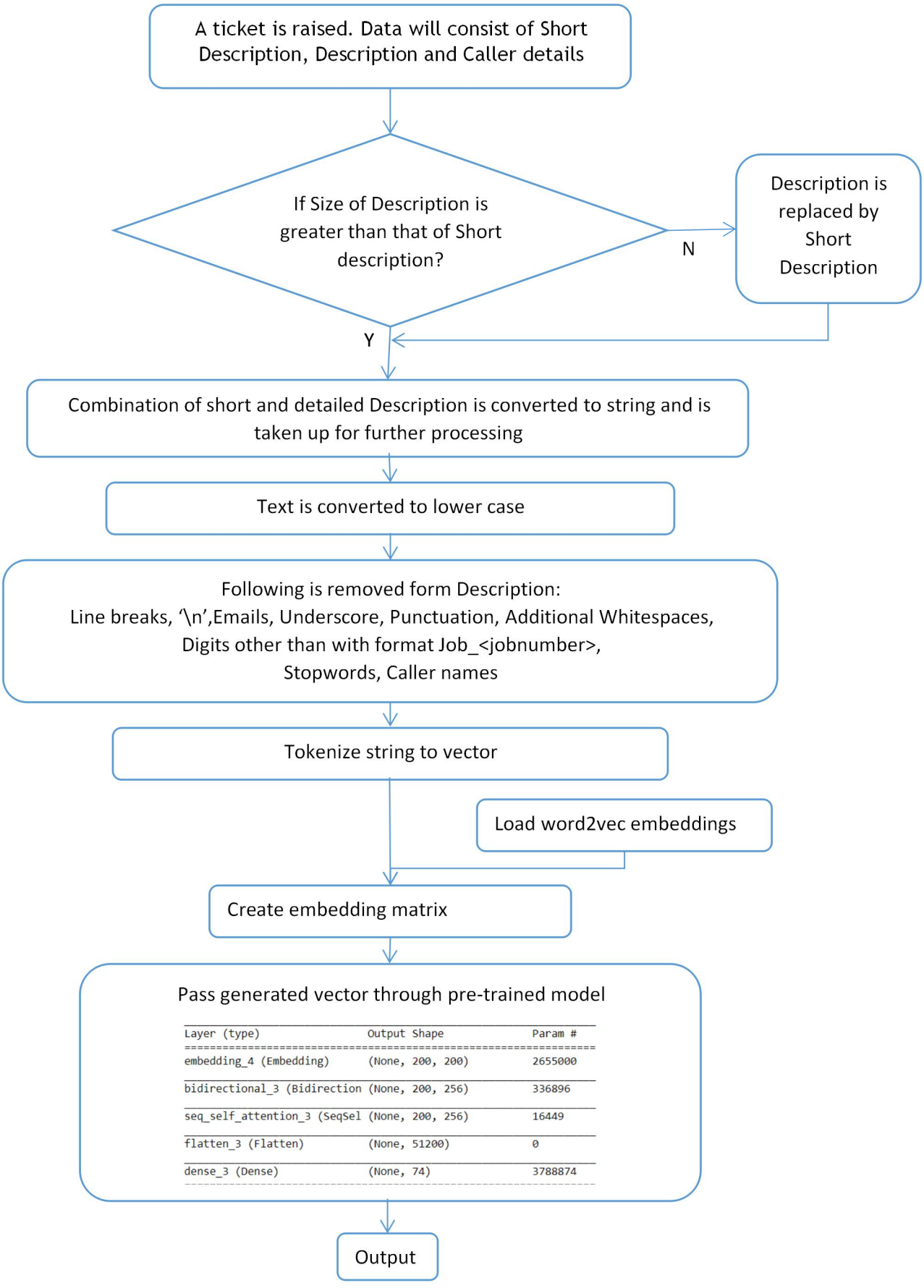
Contents

1. Abstract

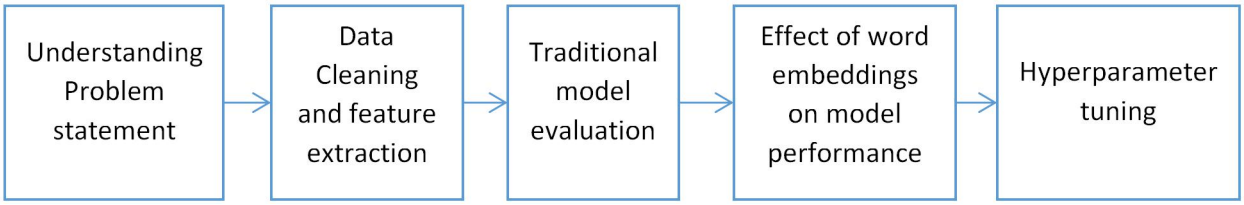
A ticketing system is a customer service tool that helps companies manage their service and support cases. It is a tool that allows IT support to be organized, focused and efficient. This directly impacts costs and revenues, customer retention, and public brand image.
An incident is something that is unplanned interruption to an IT service or reduction in the quality of an IT service. The main goal of Incident Management process is to provide a quick fix / workarounds or solutions that resolves the interruption and restores the service to its full capacity to ensure no business impact. It goes through a lifecycle of Identifying and recording the issue to management and resolution of the same. Assigning the incidents to the appropriate person or unit in the support team has critical importance to provide improved user satisfaction with minimal wastage of time and resources.
1. An incident management system guided by powerful AI techniques that can classify incidents to right functional groups can help organizations to reduce the resolving time of the issue and yield more productive tasks. In this capstone project, the goal is to build a classifier that can classify the tickets by analyzing text in ticket raised. Data provided for building model for ticketing system consists of 8500 entries with 4 columns. They are Short description, Description, Caller name and Assignment_group. Short and Detailed description were combined together, cleaned and tokenized to form Input Variable. Assignment Group is our target variable. Upon undergoing a series of steps that are detailed in document for model building, we were able to achieve an accuracy of 70.4%.

2 Overview of Final Process:

A ticket after being raised will go through the flow as shown in flowchart below:



Our approach consisted of 5 broad steps:



1. Exploratory data analysis: Data was explored and various visualizations were created using word cloud to get a sense of terminology and to identify stopwords and special characters that might need cleaning. There were 8500 entries divided among 74 assignment groups. Column with Caller names did not provide any specific insight.

2. Data Cleaning and feature vector generation: Upon identification of stopwords, punctuations, special characters and missing fields, data was appropriately cleaned and word frequency and TF-IDF was determined to produce feature vector for model fitting.

3. Traditional model evaluation: Input vector was fit into traditional models like Naive Bayes, logistic Regression, SVM, Random Forest, XG boost, KNN and simple Neural Network. Random forest and Neural Network gave better accuracy than others.Complex network consisting of Dense, Batch Normalization and Dropout layer was constructed for modelfitting. The ones that performed better than others were further tuned through their hyperparameters.

Out[48]:

	Model	unigram-tfidf Accuracy	bigram-tfidf Accuracy	trigram-tfidf Accuracy
0	Naive Bayes	51.292	50.000	49.815
1	Linear-Logistic Regression	53.506	51.292	51.107
2	SVM Classifier	52.952	51.292	51.107
3	RandomForest \n Esti = 600,Cri=gini,max_depth=...	54.982	56.273	55.535
4	XGBoost	54.244	53.506	54.059
5	KNN Classifier (k=5)	52.214	50.738	50.185
6	Simple NN	53.875	55.351	54.613

4. Word embeddings boost to model performance: Different Embeddings namely ELMO, BERT, FastText, Word2Vec and GloVe were explored to further enhance the model performance. Our main focus was on Fast Text, Word2Vec and GloVe. And of the three Word2Vec provided with the best performance.

5. Hyperparameter Tuning: Models with word enbeddings as input were tested with LSTM and Bidirectional LSTM layer. Bidirectional LSTM was found to yield better performance. At the end of this step, we were able to achieve 61.8% accuracy. After further hyperparameter tuning and inserting appropriate callbacks, the performance improved to 70.4%

Embedding	**LSTM**	**BiLSTM**	**BiLSTM + Attention layer + Calbacks**
Glove	57.6	60.2	68.7
Word2vec	59.3	61.8	70.4
Fastext	54.2	57.6	67.5

3 Walk through the solution

3.1 Exploratory Data Analysis

3.1.1 Data Exploration

Data for building model for ticketing system consists of 8500 entries with 4 columns. Following is the list these 4 columns.

2. 'Short description': Short summary of ticket

3. 'Description': Description of the issue faced for which the ticket is raised. Since, this column has more information that 'Short description'column, this will be our input variable. There are 7817 unique values on the description. This signifies there might be some duplicates in the column. There are cases where length of Description is smaller than that of 'Short Description'column. In such cases, Short description will be considered instead of Description.

4. 'Caller': Name of the person who raised the ticket. There are 2950 unique callers. This column does not provide any special information in differentiating the Assignment group allotment. Hence, this column can be removed for dataset.

5. 'Assignment_group': Group name to which the ticket is assigned to.There are total of 74 groups out of which maximum entries belong to GRP_0. out of 8500, 3976 entries belong to GRP_0. This column will be our target variable.

Following is an image summarizing the data for building classifier:

```
In [9]: data_df.shape
Out[9]: (8500, 4)

In [10]: data_df.describe()
```

Out[10]:

	Short description	Description	Caller	Assignment group
count	8492	8499	8500	8500
unique	7481	7817	2950	74
top	password reset	the	bpctwhsn kzqsbmtp	GRP_0
freq	38	56	810	3976

First 5 entries of data can be seen in image below:

	Short description	Description	Caller	Assignment group
0	login issue	-verified user details.(employee# & manager name)\r\n-checked the user name in ad and reset the password.\r\n-advised the user to login and check.\r\n-caller confirmed that he was able to login.\r\n-issue resolved.	spxjnwir pjlcoqds	GRP_0
1	outlook	\r\n\r\nreceived from: hmjdrvpb.komuaywn@gmail.com\r\n\r\nhello team,\r\n\r\nmy meetings/skype meetings etc are not appearing in my outlook calendar, can somebody please advise how to correct this?\r\n\r\nkind	hmjdrvpb komuaywn	GRP_0
2	cant log in to vpn	\r\n\r\nreceived from: eylqgodm.ybqkwiam@gmail.com\r\n\r\nhi\r\n\r\ni cannot log on to vpn\r\n\r\nbest	eylqgodm ybqkwiam	GRP_0
3	unable to access hr_tool page	unable to access hr_tool page	xbkucsvz gcpydteq	GRP_0
4	skype error	skype error	owlgqjme qhcozdfx	GRP_0
5	unable to log in to engineering tool and skype	unable to log in to engineering tool and skype	eflahbxn ltdgrvkz	GRP_0

To get a visual understanding of data, Word Cloud was constructed for ‘Description’ column.

First look of Description column:

Upon data exploration, following observations were recorded:

1. There are os related new line and line termination tags.

2. Few description have header - received from : - which doesn't provide much information on classification. Few description also have footer note - Thanks/regards followed by name - which doesn't provide much information on classification.

3. There are few encoded words in description which could be name of the persons that are encrypted owing to PII governance which needs to be handled in pre-processing.

4. There are also few system drive path, software versioning number and ip addresses.

5. Some description might have attached with evidence photos which resulted in cid: tags in the description footer.

6. ‘_’ was used to join main functions like job_scheduler, hr_payroll. Therefore, upon removal of ‘_’,bigrams and trigrams will be required to improve model performance.

7. Some of the entries mention job-numbers in their description that are important in classificaiton of group. Therefore, we have to be careful not to remove these numbers while cleaning digits.

8. ‘no’, ‘not’ and similar negative words are not to be removed from data

9. There are email references that do not add value to classification

10. Column ‘caller’ does not provide any important information towards classification. Also, they need to be added to stopwords as well.

11. Punctuation and special characters needs to be removed from data.

12. There are cases where Description is either not present or Short description is more informative. In such cases, short description will have to be considered

13. There are few entries without any english words

14. NA will have to be dealt with.

15. There are few groups that have just one entry. It will be difficult to train a model in presence of such low entries. Hence, groups less than 6 entries are clubbed together to form just one group. There were 44 such entries. And all these are clubbed together under one group GRP_COMMON. After this step, we have a total of 54 groups.

In [17]:
```
#list the description which are less than 5 groupiing
temp_df = desc_df.groupby('assignment_group').filter(lambda x: len(x) < 6)
temp_df.describe()
```

Out[17]:

	short_description	description	assignment_group
count	44	44	44
unique	44	43	19
top	update cutview to lauacyltoe hxgaycze version	update cutview to lauacyltoe hxgaycze version and adjust to current ms office installation	GRP_43
freq	1	2	5

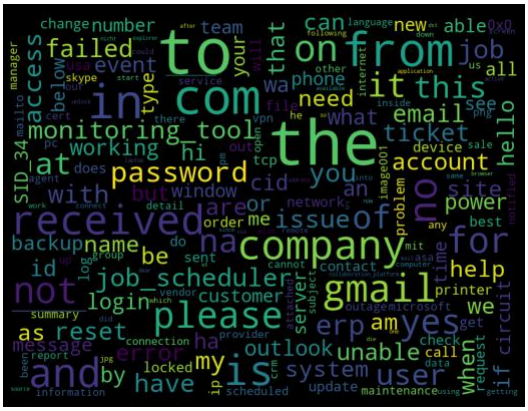
In [0]:
```
#desc_df['assignment_group'] = 'GRP_COMMON'
desc_df['assignment_group'] = desc_df.assignment_group.apply(lambda v: 'GRP_COMMON' if (desc_df.assignment_group.value_counts()
```

3.1.2 Data Cleaning

Following are the preprocessing steps taken on data:

1. Wherever length of Short description was smaller than that of Description, Description was modified to include content from Short Description.
2. All Characters were converted to lower case.
3. Escape characters like new line were removed.
4. Stopwords were updated to include received from header & footer tags and image attachment reference in footer.
5. email id references were removed.
6. Punctuation characters were removed.
7. Stop words were removed.
8. Digits other than Job_<job number> were removed.
9. Caller names were removed.

Word cloud for Description column upon cleaning:



Before Cleaning

After Cleaning

This visual representation provides with better understanding of vocabulary we are dealing with. We also know that out of these 54 groups, GRP_0 has maximum entries. Hence, this representation will have maximum words from GRP_0. Therefore, it is important to see word cloud for individual groups as well.

Following is the list of top 10 Assignment groups:

Out[19]:

	assignment_group
GRP_0	3976
GRP_8	661
GRP_24	289
GRP_12	257
GRP_9	252
GRP_2	241
GRP_19	215
GRP_3	200
GRP_6	184
GRP_13	145

To have further understanding of vocabulary, word clouds for these 10 assignment groups were generated (See Appendix).

Following is an observation summary of top ten groups:

Group	Observations
GRP_0	Word cloud looks lot similar to the word cloud of Description. Most of the issues look like they arise from inability to access an account.
GRP_8	Most of issues are concentrated around job scheduler and failed job.
GRP_24	Issues consist of installation and setup related problems
GRP_12	Maximum tickets describe server access and disk space to be the issue.
GRP_9	This looks similar to group 8 with key words being job scheduler and failed job. Job numbers become important here. Also, most of the issues apear to be sales related.
GRP_2	Keywords here are 'sid', 'hrp', 'erp'. Therefore , we have to be careful with removal of non-english words. Also, embeddings to be selected needs to have this vocabulary.
GRP_19	Most tickets are regarding laptop or printer not working
GRP_3	We can see some non-english but frequently used acronyms like tcp, sep, src and dst. We have to be careful with them while data cleaning as well as choosing an embedding source.
GRP_6	This also is similar to GRP_8 and GRP_9. Job numbers become important here as well. The tickets here appear to be raised by supply chain team.
GRP_13	Main keywords are order, billing, and shipment. This group looks to catering to order and billing related issues.

3.1.3 N gram analysis

Bigrams and trigrams provides more sensible keywords that would help us in enhancing overall performance of model.

Word cloud for unigram:
Total no of uni-grams from corpus without stopwords : 14520
Size of uni-gram Vocabulary upon cleaning : 3049
Size of updated uni-gram Tokens based on TF-IDF frequency : 2659

Word Cloud for unigram + bigrams:

Total no of bi-grams from corpus without stopwords : 61580
Size of bi-gram Vocabulary upon cleaning : 30368
Size of updated bi-gram Tokens based on TF-IDF frequency: 26692

Word Cloud for unigrams+ bigrams+ trigrams:

Total no of tri-grams from corpus without stopwords: 75257
Size of tri-gram Vocabulary upon cleaning : 38857
Size of updated tri-gram Tokens based on TF-IDF frequency : 36155

After Generation of Ngrams, TF-IDF was calculated and on its basis, high and low frequency words were removed. Data was further vectorized and input matrix consisting of unigrams, bigrams and trigrams was constructed. This was provided as input to models.

```
In [0]: unigram_vocal = get_top_n_ngram(cleansed_data_df['description'],(1,1), None, 'tfidf')
        bigram_vocal = get_top_n_ngram(cleansed_data_df['description'],(2,2), None, 'tfidf')
        trigram_vocal = get_top_n_ngram(cleansed_data_df['description'],(3,3), None, 'tfidf')
```

```
In [0]: unigram_vec = TfidfVectorizer(ngram_range=(1,1)).fit(cleansed_data_df['description'])
        bigram_vec = TfidfVectorizer(ngram_range=(2,2)).fit(cleansed_data_df['description'])
        trigram_vec = TfidfVectorizer(ngram_range=(3,3)).fit(cleansed_data_df['description'])
```

```
In [206]: print('Size of uni-gram Vocabulary : ', unigram_vocal.shape[0])
          print('Size of bi-gram Vocabulary : ', bigram_vocal.shape[0])
          print('Size of tri-gram Vocabulary : ', trigram_vocal.shape[0])

          Size of uni-gram Vocabulary :  3049
          Size of bi-gram Vocabulary :  30368
          Size of tri-gram Vocabulary :  38857
```

3.2 Modelling

After cleansing the data,before moving to Complex models, we first started with traditional machine learning models namely, Naïve Bayes, Logistic Regression, SVM Classifier, Random Forest, XGBoost and KNN.

For modeling we have taken 90% train and 10% test data for modeling.

```
y, le = encode_target(y)
```

In [0]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.1, stratify=y, random_state = 10)
```

Since it is a classification problem we have used the aforementioned algorithms.

Modeling was done on uni-gram, bi-gram and tri-grams. Below is the vocab size for all the three.

In [19]:
```
print('unigram vocab : ', unigram_vocab.shape)
print('bigram vocab : ', bigram_vocab.shape)
print('trigram vocab : ', trigram_vocab.shape)
```

```
unigram vocab :  (2659, 2)
bigram vocab :  (26692, 2)
trigram vocab :  (36155, 2)
```

Then we calculated the Tf-Idf Vectorizer for Uni-gram, bi-gram and tri-gram.

In [23]:
```
unigram_tokenizer = TfidfVectorizer(ngram_range=(1,1), max_features = unigram_max_features, vocabulary=uni
gram_vocab['words']).fit(X)
X_train_unigram_seq = unigram_tokenizer.transform(X_train)
X_test_unigram_seq = unigram_tokenizer.transform(X_test)
print('Vocabulary size of unigram :', len(unigram_tokenizer.vocabulary_))

bigram_tokenizer = TfidfVectorizer(ngram_range=(1,2), max_features = bigram_max_features, vocabulary=unigr
am_bigram_vocab['words']).fit(X)
X_train_bigram_seq = bigram_tokenizer.transform(X_train)
X_test_bigram_seq = bigram_tokenizer.transform(X_test)
print('Vocabulary size of bigram :', len(bigram_tokenizer.vocabulary_))

trigram_tokenizer = TfidfVectorizer(ngram_range=(1,3), max_features = trigram_max_features, vocabulary=uni
gram_bigram_trigram_vocab['words']).fit(X)
X_train_trigram_seq = trigram_tokenizer.transform(X_train)
X_test_trigram_seq = trigram_tokenizer.transform(X_test)
print('Vocabulary size of trigram :', len(trigram_tokenizer.vocabulary_))
```

```
Vocabulary size of unigram : 2659
Vocabulary size of bigram : 29351
Vocabulary size of trigram : 65506
```

We then modeled the uni-gram, bi-grams and tri-grams vectors.

Below are the models used and their corresponding accuracy scores.

3.1.1 Traditional Models

3.1.1.1 Naïve Bayes

Naïve Bayes is a fast algorithm for classification problem. This algorithm is a good fit for real-time prediction, multi-class prediction, recommendation system, text classification, and sentiment analysis use cases. Naive Bayes Algorithm can be built using Gaussian, Multinomial and Bernoulli distribution. This algorithm is scalable and easy to implement for the large data set.

Due to its better performance with multi-class problems and its independence rule, Naive Bayes algorithm perform better or have a higher success rate in text classification, Therefore, it is used in Sentiment Analysis and Spam filtering.

Naive Bayes :

In [31]:
```
# Naive Bayes on tfidf Vectors
accuracy1 = train_model(naive_bayes.MultinomialNB(), X_train_unigram_seq, y_train, X_test_unigram_seq, y_t
est)
accuracy2 = train_model(naive_bayes.MultinomialNB(), X_train_bigram_seq,  y_train, X_test_bigram_seq,  y_t
est)
accuracy3 = train_model(naive_bayes.MultinomialNB(), X_train_trigram_seq, y_train, X_test_trigram_seq, y_t
est)

accuracy_df = accuracy_df.append(pd.Series(['Naive Bayes', accuracy1,accuracy2,accuracy3],index=accuracy_d
f.columns), ignore_index=True)
accuracy_df
```

Out[31]:

	Model	unigram-tfidf Accuracy	bigram-tfidf Accuracy	trigram-tfidf Accuracy
0	Naive Bayes	51.292	50.0	49.815

3.1.1.2 Logistic Regression

Logistic Regression is widely used technique because it is very efficient, does not require too many computational resources, it's highly interpretable, it doesn't require input features to be scaled, it doesn't require any tuning, it's easy to regularize, and it outputs well-calibrated predicted probabilities. Another advantage of Logistic Regression is that it is incredibly easy to implement and very efficient to train.

Linear Classifier :

In [32]:
```
# Logistic Regression on tfidf Vectors
accuracy1 = train_model(linear_model.LogisticRegression(), X_train_unigram_seq, y_train, X_test_unigram_se
q, y_test)
accuracy2 = train_model(linear_model.LogisticRegression(), X_train_bigram_seq,  y_train, X_test_bigram_se
q,  y_test)
accuracy3 = train_model(linear_model.LogisticRegression(), X_train_trigram_seq, y_train, X_test_trigram_se
q, y_test)

accuracy_df = accuracy_df.append(pd.Series(['Linear-Logistic Regression', accuracy1,accuracy2,accuracy3],i
ndex=accuracy_df.columns), ignore_index=True)
accuracy_df
```

Out[32]:

	Model	unigram-tfidf Accuracy	bigram-tfidf Accuracy	trigram-tfidf Accuracy
0	Naive Bayes	51.292	50.000	49.815
1	Linear-Logistic Regression	53.506	51.292	51.107

3.1.1.3 SVM Classifier

SVM separates data points using a hyperplane with the largest amount of margin. That's why an SVM classifier is also known as a discriminative classifier. SVM finds an optimal hyperplane which helps in classifying new data points. The benefit is that it can capture much more complex relationships between data points without having to perform difficult transformations on your own.

SVM's can model non-linear decision boundaries, and there are many kernels to choose from. They are also fairly robust against overfitting, especially in high-dimensional space.

Support Vector Machines :

In [33]:
```
# SVM on tfidf Vectors
accuracy1 = train_model(svm.SVC(), X_train_unigram_seq, y_train, X_test_unigram_seq, y_test)
accuracy2 = train_model(svm.SVC(), X_train_bigram_seq,  y_train, X_test_bigram_seq,  y_test)
accuracy3 = train_model(svm.SVC(), X_train_trigram_seq, y_train, X_test_trigram_seq, y_test)

accuracy_df = accuracy_df.append(pd.Series(['SVM Classifier', accuracy1,accuracy2,accuracy3],index=accurac
y_df.columns), ignore_index=True)
accuracy_df
```

Out[33]:

	Model	unigram-tfidf Accuracy	bigram-tfidf Accuracy	trigram-tfidf Accuracy
0	Naive Bayes	51.292	50.000	49.815
1	Linear-Logistic Regression	53.506	51.292	51.107
2	SVM Classifier	52.952	51.292	51.107

3.1.1.4 Random Forest

Random forest is considered as a highly accurate and robust method because of the number of decision trees participating in the process. It does not suffer from the overfitting problem. The main reason is that it takes the average of all the predictions, which cancels out the biases.

Random forests can also handle missing values. There are two ways to handle these: using median values to replace continuous variables, and computing the proximity-weighted average of missing values.

In [38]:
```
# RandomForest on tfidf Vectors
n_estimators = 600
criterion = 'gini'
max_depth = 100
min_samples_split = 6

unigram_rf_model = randomforest_model(n_estimators,criterion,100,6)
bigram_rf_model = randomforest_model(n_estimators,criterion,None,10)
trigram_rf_model = randomforest_model(n_estimators,criterion,None,10)

accuracy1 = train_model(unigram_rf_model, X_train_unigram_seq, y_train, X_test_unigram_seq, y_test)
accuracy2 = train_model(bigram_rf_model, X_train_bigram_seq,  y_train, X_test_bigram_seq,  y_test)
accuracy3 = train_model(trigram_rf_model, X_train_trigram_seq, y_train, X_test_trigram_seq, y_test)

accuracy_df = accuracy_df.append(pd.Series(['RandomForest: Esti={},Cri={},max_depth={})'.format(n_estimato
rs,criterion,max_depth),
                                            accuracy1,accuracy2,accuracy3],index=accuracy_df.columns), igno
re_index=True)

accuracy_df
```

Out[38]:

	Model	unigram-tfidf Accuracy	bigram-tfidf Accuracy	trigram-tfidf Accuracy
0	Naive Bayes	51.292	50.000	49.815
1	Linear-Logistic Regression	53.506	51.292	51.107
2	SVM Classifier	52.952	51.292	51.107
3	RandomForest \n Esti = 600,Cri=gini,max_depth=...	54.982	56.273	55.535

3.1.1.5 XGBoost

The XGBoost library implements the gradient boosting decision tree algorithm. XGBoost goes by lots of different names such as gradient boosting, multiple additive regression trees, stochastic gradient boosting or gradient boosting machines.

XGBoost is an implementation of gradient boosted decision trees designed for speed and performance. Generally, XGBoost is fast when compared to other implementations of gradient boosting. XGBoost dominates structured or tabular datasets on classification and regression predictive modeling problems.

XGBoost Classifier :

In [0]:
```
# XGBoost on tfidf Vectors
accuracy1 = train_model(xgboost.XGBClassifier(), X_train_unigram_seq, y_train, X_test_unigram_seq, y_test)
accuracy2 = train_model(xgboost.XGBClassifier(), X_train_bigram_seq, y_train, X_test_bigram_seq, y_test)
accuracy3 = train_model(xgboost.XGBClassifier(), X_train_trigram_seq, y_train, X_test_trigram_seq, y_test)
```

In [40]:
```
accuracy_df = accuracy_df.append(pd.Series(['XGBoost', accuracy1,accuracy2,accuracy3],index=accuracy_df.co
lumns), ignore_index=True)
accuracy_df
```

Out[40]:

	Model	unigram-tfidf Accuracy	bigram-tfidf Accuracy	trigram-tfidf Accuracy
0	Naive Bayes	51.292	50.000	49.815
1	Linear-Logistic Regression	53.506	51.292	51.107
2	SVM Classifier	52.952	51.292	51.107
3	RandomForest \n Esti = 600,Cri=gini,max_depth=...	54.982	56.273	55.535
4	XGBoost	54.244	53.506	54.059

3.1.1.6 KNN

KNN uses lazy training which means all computation is deferred till prediction. This works very well if we have good training data. Naïve Bayes is a quick classifier and K-NN should be preferred when the data-set is relatively small. For this particular data set the KNN was not performing well.

KNN Classifier:

In [0]:
```
# KNN Classifier on tfidf Vectors
accuracy1 = train_model(neighbors.KNeighborsClassifier(n_neighbors=5, algorithm='brute'), X_train_unigram_
seq, y_train, X_test_unigram_seq, y_test)
accuracy2 = train_model(neighbors.KNeighborsClassifier(n_neighbors=5, algorithm='brute'), X_train_bigram_s
eq, y_train, X_test_bigram_seq, y_test)
accuracy3 = train_model(neighbors.KNeighborsClassifier(n_neighbors=5, algorithm='brute'), X_train_trigram_
seq, y_train, X_test_trigram_seq, y_test)
```

In [42]:
```
accuracy_df = accuracy_df.append(pd.Series(['KNN Classifier (k=5)', accuracy1,accuracy2,accuracy3],index=a
ccuracy_df.columns), ignore_index=True)
accuracy_df
```

Out[42]:

	Model	unigram-tfidf Accuracy	bigram-tfidf Accuracy	trigram-tfidf Accuracy
0	Naive Bayes	51.292	50.000	49.815
1	Linear-Logistic Regression	53.506	51.292	51.107
2	SVM Classifier	52.952	51.292	51.107
3	RandomForest \n Esti = 600,Cri=gini,max_depth=...	54.982	56.273	55.535
4	XGBoost	54.244	53.506	54.059
5	KNN Classifier (k=5)	52.214	50.738	50.185

Conclusion: Naïve Bayes performed the worst Random Forest performed the best amongst all classifier.

3.2.2 Deep learning Models

3.2.2.1 Simple NN

After Machine learning models, we tried deep learning models. Initially we tried simple neural network.

In [0]:
```
def create_nn_model(input_size, output_size):
    model = models.Sequential()

    model.add(layers.Dense(512,input_dim=input_size, kernel_initializer='random_uniform', kernel_regulariz
er=regularizers.l2(Lambda)))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('relu'))
    model.add(layers.Dropout(0.5))

    model.add(layers.Dense(256,kernel_initializer='random_uniform',kernel_regularizer=regularizers.l2(Lamb
da)))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('relu'))
    model.add(layers.Dropout(0.5))

    model.add(layers.Dense(128,kernel_initializer='random_uniform',kernel_regularizer=regularizers.l2(Lamb
da)))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('relu'))
    model.add(layers.Dropout(0.3))

    model.add(layers.Dense(64, kernel_initializer='random_uniform',kernel_regularizer=regularizers.l2(Lamb
da)))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('relu'))

    model.add(layers.Dense(output_size, activation='softmax'))
    return model
```

In [0]:
```
def train_deep_model(model, feature_train, target_train, feature_validate, target_validate, epochs, batch_
size, lr, Lambda, callbacks):
  model.compile(optimizer=optimizers.Adam(learning_rate=lr),
                loss='categorical_crossentropy', metrics=['accuracy'])
  history = model.fit(feature_train, target_train, validation_split=0.2,
                      epochs=epochs, batch_size=batch_size, verbose=1,
                      callbacks = callbacks)
  predict = model.predict_classes(feature_validate)
  return (metrics.accuracy_score(predict, target_validate)*100).round(3)
```

In [0]:
```
y_train_cate = keras.utils.to_categorical(y_train)
y_test_cate = keras.utils.to_categorical(y_test)
```

In [47]:
```
# Simple NN with own embedding on tfidf Vectors
epochs = 50
batch_size = 128
lr = 1e-2
Lambda = 1e-5
stop = EarlyStopping(monitor="loss", patience=10, mode="min")
reduce_lr = ReduceLROnPlateau(monitor="loss", factor=0.1, patience=3, min_lr=1e-8, verbose=1, mode="min")
callback = [reduce_lr,stop]

model1 = create_nn_model(X_train_unigram_seq.shape[1], y_train_cate.shape[1])
accuracy1 = train_deep_model(model1, X_train_unigram_seq, y_train_cate, X_test_unigram_seq, y_test, epoch
s, batch_size, lr, Lambda, callback)

model2 = create_nn_model(X_train_bigram_seq.shape[1], y_train_cate.shape[1])
accuracy2 = train_deep_model(model2, X_train_bigram_seq, y_train_cate, X_test_bigram_seq, y_test, epochs,
batch_size, lr, Lambda, callback)

model3 = create_nn_model(X_train_trigram_seq.shape[1], y_train_cate.shape[1])
accuracy3 = train_deep_model(model3, X_train_trigram_seq, y_train_cate, X_test_trigram_seq, y_test, epoch
s, batch_size, lr, Lambda, callback)
```

```
Train on 3901 samples, validate on 976 samples
Epoch 1/50
3901/3901 [==============================] - 2s 397us/step - loss: 2.7734 - accuracy: 0.4371 - val_loss:
3.0713 - val_accuracy: 0.5000
Epoch 2/50
3901/3901 [==============================] - 1s 189us/step - loss: 2.3058 - accuracy: 0.5040 - val_loss:
2.6218 - val_accuracy: 0.5000
```

The final accuracy score of the models are below:

Out[48]:

	Model	unigram-tfidf Accuracy	bigram-tfidf Accuracy	trigram-tfidf Accuracy
0	Naive Bayes	51.292	50.000	49.815
1	Linear-Logistic Regression	53.506	51.292	51.107
2	SVM Classifier	52.952	51.292	51.107
3	RandomForest \n Esti = 600,Cri=gini,max_depth=...	54.982	56.273	55.535
4	XGBoost	54.244	53.506	54.059
5	KNN Classifier (k=5)	52.214	50.738	50.185
6	Simple NN	53.875	55.351	54.613

Conclusion : Random Forest performed the best amongst the basic Deep Learning and Machine Learning models.

3.2.3 Hyper parameter tuning

As mentioned in above section, we removed all the stop words and cleansed the data and combined the rows where there were few data sets and then did modeling on that. For better performance of the model we tested for uni-gram, bi-gram and tri-gram vectors. However, the results were not great. The accuracy score was ~54%.

Out[48]:

	Model	unigram-tfidf Accuracy	bigram-tfidf Accuracy	trigram-tfidf Accuracy
0	Naive Bayes	51.292	50.000	49.815
1	Linear-Logistic Regression	53.506	51.292	51.107
2	SVM Classifier	52.952	51.292	51.107
3	RandomForest \n Esti = 600,Cri=gini,max_depth=...	54.982	56.273	55.535
4	XGBoost	54.244	53.506	54.059
5	KNN Classifier (k=5)	52.214	50.738	50.185
6	Simple NN	53.875	55.351	54.613

Random Forest and Simple Neural Network has better accuracy score in bi-gram and tri-grams. However, the others performed well in uni-grams.

3.2.3.1 Random Forest Tuning

We performed hyper-parameter tuning for Random Forest to find best parameters.

In [0]:
```
from sklearn.model_selection import RandomizedSearchCV
# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 10)]
# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(100, 1000, num = 11)]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [6, 10, 15]

# Create the random grid
random_grid = {'n_estimators': n_estimators,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split}

# Use the random grid to search for best hyperparameters
rf = ensemble.RandomForestClassifier()
# Random search of parameters, using 3 fold cross validation,
rf_random = RandomizedSearchCV(estimator = rf, param_distributions = random_grid, n_iter = 50, cv = 3, ver
bose=2, random_state=42, n_jobs = 2)
```

3.2.3.2 Deep learning model

Initially the neural network score was 50.526, 52.105 and 53.333 for uni-gram, bi-gram and tri-grams. To improve the model score we added more Dense Layer, performed batch normalization, added Dropout Layer, used callback and optimizer regularization loss.

In [0]:
```
def create_nn_model(input_size, output_size):
    model = models.Sequential()

    model.add(layers.Dense(512,input_dim=input_size, kernel_initializer='random_uniform', kernel_regulariz
er=regularizers.l2(Lambda)))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('relu'))
    model.add(layers.Dropout(0.5))

    model.add(layers.Dense(256,kernel_initializer='random_uniform',kernel_regularizer=regularizers.l2(Lamb
da)))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('relu'))
    model.add(layers.Dropout(0.5))

    model.add(layers.Dense(128,kernel_initializer='random_uniform',kernel_regularizer=regularizers.l2(Lamb
da)))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('relu'))
    model.add(layers.Dropout(0.3))

    model.add(layers.Dense(64, kernel_initializer='random_uniform',kernel_regularizer=regularizers.l2(Lamb
da)))
    model.add(layers.BatchNormalization())
    model.add(layers.Activation('relu'))

    model.add(layers.Dense(output_size, activation='softmax'))
    return model
```

In [0]:
```
def train_deep_model(model, feature_train, target_train, feature_validate, target_validate, epochs, batch_
size, lr, Lambda, callbacks):
  model.compile(optimizer=optimizers.Adam(learning_rate=lr),
                loss='categorical_crossentropy', metrics=['accuracy'])
  history = model.fit(feature_train, target_train, validation_split=0.2,
                      epochs=epochs, batch_size=batch_size, verbose=1,
                      callbacks = callbacks)
  predict = model.predict_classes(feature_validate)
  return (metrics.accuracy_score(predict, target_validate)*100).round(3)
```

In [0]:
```
y_train_cate = keras.utils.to_categorical(y_train)
y_test_cate = keras.utils.to_categorical(y_test)
```

In [47]:
```
# Simple NN with own embedding on tfidf Vectors
epochs = 50
batch_size = 128
lr = 1e-2
Lambda = 1e-5
stop = EarlyStopping(monitor="loss", patience=10, mode="min")
reduce_lr = ReduceLROnPlateau(monitor="loss", factor=0.1, patience=3, min_lr=1e-8, verbose=1, mode="min")
callback = [reduce_lr,stop]

model1 = create_nn_model(X_train_unigram_seq.shape[1], y_train_cate.shape[1])
accuracy1 = train_deep_model(model1, X_train_unigram_seq, y_train_cate, X_test_unigram_seq, y_test, epoch
s, batch_size, lr, Lambda, callback)

model2 = create_nn_model(X_train_bigram_seq.shape[1], y_train_cate.shape[1])
accuracy2 = train_deep_model(model2, X_train_bigram_seq, y_train_cate, X_test_bigram_seq, y_test, epochs,
batch_size, lr, Lambda, callback)

model3 = create_nn_model(X_train_trigram_seq.shape[1], y_train_cate.shape[1])
accuracy3 = train_deep_model(model3, X_train_trigram_seq, y_train_cate, X_test_trigram_seq, y_test, epoch
s, batch_size, lr, Lambda, callback)
```

```
Train on 3901 samples, validate on 976 samples
Epoch 1/50
3901/3901 [==============================] - 2s 397us/step - loss: 2.7734 - accuracy: 0.4371 - val_loss:
3.0713 - val_accuracy: 0.5000
Epoch 2/50
3901/3901 [==============================] - 1s 189us/step - loss: 2.3058 - accuracy: 0.5040 - val_loss:
2.6218 - val_accuracy: 0.5000
Epoch 3/50
3901/3901 [==============================] - 1s 191us/step - loss: 2.0874 - accuracy: 0.5345 - val_loss:
```

We ran the model for 50 epochs. Below are the accuracy score after performing the above mentioned steps.

Out[48]:

	Model	unigram-tfidf Accuracy	bigram-tfidf Accuracy	trigram-tfidf Accuracy
0	Naive Bayes	51.292	50.000	49.815
1	Linear-Logistic Regression	53.506	51.292	51.107
2	SVM Classifier	52.952	51.292	51.107
3	RandomForest \n Esti = 600,Cri=gini,max_depth=...	54.982	56.273	55.535
4	XGBoost	54.244	53.506	54.059
5	KNN Classifier (k=5)	52.214	50.738	50.185
6	Simple NN	53.875	55.351	54.613

3.3 Embeddings

There are different types of Embeddings available. Following is a list of a select few.

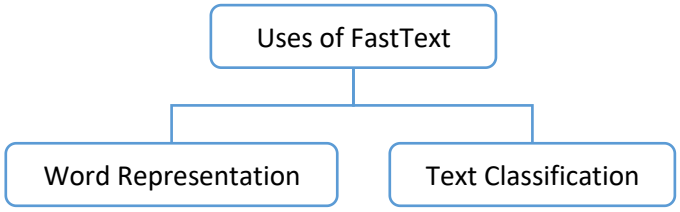
1. ELMO
2. BERT
3. Fast Text
4. Word2Vec
5. GloVe

Model Name	Context Sensitive embeddings	Learnt representations
Word2vec	No	Words
Glove	No	Words
ELMo	Yes	Words
BERT	Yes	Subwords

Out of Above mentioned models, our main focus was on Fast Text, Word2Vec and Glove.

3.3.1 Fast Text:

FastText is a library created by the Facebook Research Team(FLAIR) for efficient learning of word representations and sentence classification.



Word vectors treats every single word as the smallest unit whose vector representation is to be found but FastText assumes a word to be formed by a n-grams of character, for example, sunny is composed of [sun, sunn,sunny],[sunny,unny,nny] etc, where n could range from 1 to the length of the word. This new representation of word by fastText provides the following benefits over word2vec or glove.

It can give the vector representations for the words not present in the dictionary (OOV words) since these can also be broken down into character n-grams. word2vec and glove both fail to provide any vector representations for words not in the dictionary. character n-grams embeddings tend to perform superior to word2vec and glove on smaller datasets.

```
max_features = 200000 #17780 #using all unique words
embedding_dim = 300
num_classes = 74
batch_size = 128

embedding_path = "/content/gdrive/My Drive/Colab Notebooks/wiki-news-300d-1M.vec"

def get_coefs(word,*arr): return word, np.asarray(arr, dtype='float32')
embedding_index = dict(get_coefs(*o.strip().split(" ")) for o in open(embedding_path))

word_index = tokenizer.word_index
nb_words = len(word_index)
embedding_matrix_fast = np.zeros((nb_words + 1, embedding_dim))
for word, i in word_index.items():
    embedding_vector = embedding_index.get(word)
    if embedding_vector is not None: embedding_matrix_fast[i] = embedding_vector
```

Model was constructed using these embeddings:

```
Model: "sequential_13"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_9 (Embedding)      (None, 1000, 300)         2209200
_________________________________________________________________
lstm_10 (LSTM)               (None, 1000, 128)         219648
_________________________________________________________________
global_max_pooling1d_10 (Glo (None, 128)               0
_________________________________________________________________
dense_15 (Dense)             (None, 128)               16512
_________________________________________________________________
dense_16 (Dense)             (None, 74)                9546
=================================================================
Total params: 2,454,906
Trainable params: 245,706
Non-trainable params: 2,209,200
```

Model was trained for 5 Epochs.

```
Epoch 00005: val_acc improved from 0.53298 to 0.54225, saving model to best_model.h5
```

```
Out[257]: <keras.callbacks.callbacks.History at 0x7fae6d04cef0>
```

Output: Validation Accuracy was found to be 54.2%.

3.3.2 Word2Vec:

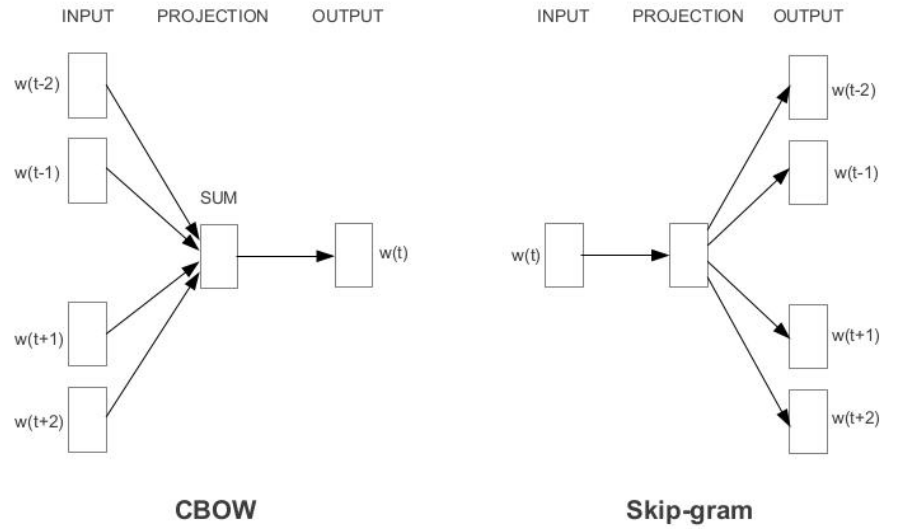
Word2Vec is one of the most popular pretrained word embeddings developed by Google. Word2Vec is trained on the Google News dataset (about 100 billion words). The architecture of Word2Vec is really simple. It's a feed-forward neural network with just one hidden layer. Hence, it is sometimes referred to as a **Shallow Neural Network architecture**.

Word2Vec is classified into two approaches:

- Continuous Bag-of-Words (CBOW)
- Skip-gram model

Continuous Bag-of-Words (CBOW) model learns the focus word given the neighbouring words whereas the Skip-gram model learns the neighbouring words given the focus word. That's why, Continuous Bag Of Words and Skip-gram are inverses of each other.

CBOW accepts multiple words as input and produces a single word as output whereas Skip-gram accepts a single word as input and produces multiple words as output.



CBOW being probabilistic is nature, it is supposed to perform superior to deterministic methods(generally). It is low on memory. It does not need to have huge RAM requirements like that of co-occurrence matrix where it needs to store three huge matrices.

Skip-gram with negative sub-sampling outperforms every other method generally

In [0]:
```
from gensim.models.keyedvectors import KeyedVectors
PROJECT_DIR = "/content/gdrive/My Drive/Colab Notebooks/"
EMBEDDING_FILE = 'SO_vectors_200.bin'
EmbeddingFile = PROJECT_DIR+EMBEDDING_FILE
word2vec = KeyedVectors.load_word2vec_format(EmbeddingFile, binary=True)
MAX_SEQUENCE_LENGTH = 30
MAX_NB_WORDS = 200000
```

In [0]:
```
# create a weight matrix for words in training docs
EMBEDDING_DIM = 200
embedding_matrix_word2vec = np.zeros((size_of_vocabulary, EMBEDDING_DIM))

for word, i in tokenizer.word_index.items():
    if word in word2vec.vocab:
        embedding_matrix_word2vec[i] = word2vec.word_vec(word)
```

Model similar to FastText was constructed using this embedding layer. Model was run for 5 epochs.

```
In [157]: y_train_vec),batch_size=128,epochs=5,validation_data=(np.array(x_val_seq_vec),np.array(y_valid_vec)),verbose=1,callbacks=[es,mc])
```

```
Train on 5695 samples, validate on 2805 samples
Epoch 1/5
5695/5695 [==============================] - 243s 43ms/step - loss: 2.9175 - acc: 0.4609 - val_loss: 2.2160 - val_acc: 0.5344

Epoch 00001: val_acc improved from -inf to 0.53440, saving model to best_model.h5
Epoch 2/5
5695/5695 [==============================] - 240s 42ms/step - loss: 2.1077 - acc: 0.5349 - val_loss: 1.9817 - val_acc: 0.5497

Epoch 00002: val_acc improved from 0.53440 to 0.54973, saving model to best_model.h5
Epoch 3/5
5695/5695 [==============================] - 239s 42ms/step - loss: 1.9206 - acc: 0.5521 - val_loss: 1.8709 - val_acc: 0.5729

Epoch 00003: val_acc improved from 0.54973 to 0.57291, saving model to best_model.h5
Epoch 4/5
5695/5695 [==============================] - 239s 42ms/step - loss: 1.7931 - acc: 0.5712 - val_loss: 1.7868 - val_acc: 0.5847

Epoch 00004: val_acc improved from 0.57291 to 0.58467, saving model to best_model.h5
Epoch 5/5
5695/5695 [==============================] - 238s 42ms/step - loss: 1.6979 - acc: 0.5868 - val_loss: 1.7213 - val_acc: 0.5939

Epoch 00005: val_acc improved from 0.58467 to 0.59394, saving model to best_model.h5
```

Output: Maximum validation accuracy was found to be 59.4%

3.3.3 GloVe:

The basic idea behind the GloVe word embedding is to derive the relationship between the words from Global Statistics. One of the simplest ways is to look at the co-occurrence matrix. A co-occurrence matrix tells us how often a particular pair of words occur together. Each value in a co-occurrence matrix is a count of a pair of words occurring together. A co-occurrence matrix tells us how often a particular pair of words occur together. Each value in a co-occurrence matrix is a count of a pair of words occurring together.

We derive the relationship between the words using simple statistics. This the idea behind the GloVe pretrained word embedding.

GloVe learns to encode the information of the probability ratio in the form of word vectors. The most general form of the model is given by:

$$F(w_i, w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}$$

```
In [146]: # load the whole embedding into memory
embeddings_index = dict()
f = open('../content/gdrive/My Drive/Colab Notebooks/glove.6B.300d.txt')

for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs

f.close()
print('Loaded %s word vectors.' % len(embeddings_index))
```

```
Loaded 400000 word vectors.
```

Model was constructed using LSTM layer

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_5 (Embedding)      (None, 1000, 300)         2210100
_________________________________________________________________
lstm_4 (LSTM)                (None, 1000, 128)         219648
_________________________________________________________________
global_max_pooling1d_4 (Glob (None, 128)               0
_________________________________________________________________
dense_7 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_8 (Dense)              (None, 74)                9546
=================================================================
Total params: 2,455,806
Trainable params: 245,706
Non-trainable params: 2,210,100
```

The model was run for 5 Epochs.

```
Epoch 00005: val_acc improved from 0.56827 to 0.57219, saving model to best_model.h5
```

Output : Validation Accuracy was found to be 57.2%

All the three Embeddings were incorporated in model using BiLSTM layers as well and Accuracy was recorded. Following table gives a comparison of accuracy achieved with these models.

Embedding	**LSTM**	**BiLSTM**
Glove	57.6	60.2
Word2vec	59.3	61.8
Fastext	54.2	57.6

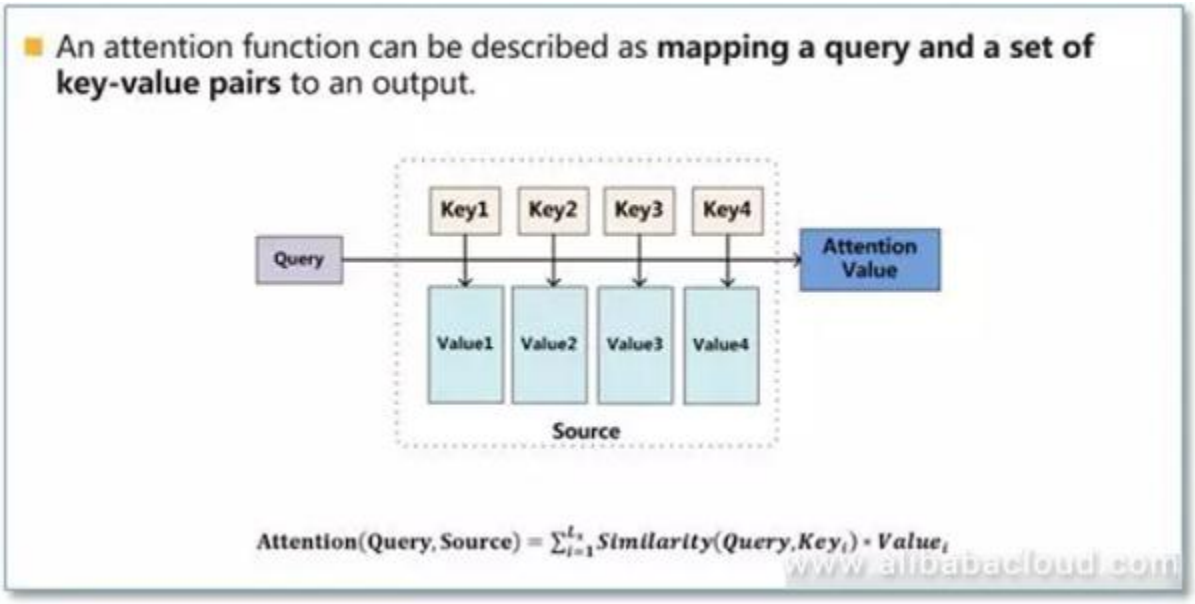
Conclusion:

Word2vec , Glove and Fastext are more about word embeddings. So the above table shows the overall accuracy of the different pretrained embeddings with two models LSTM and BiLSTM. The use of BiLSTM shows good results.

3.4 Additional Steps

In order to improve model performance, 2 main steps were taken.

3.4.1 Self Attention layer



Calculating attention comes primarily in three steps. First, we take the query and each key and compute the similarity between the two to obtain a weight. Frequently used similarity functions include dot product, splice, detector, etc. The second step is typically to use a softmax function to normalize these weights, and finally to weight these weights in conjunction with the corresponding values and obtain the final Attention.

3.4.2 Callbacks

You define and use a callback when you want to automate some tasks after every training/epoch that help you have controls over the training process. This includes stopping training when you reach a certain accuracy/loss score, saving your model as a checkpoint after each successful epoch, adjusting the learning rates over time, and more. The loss function chosen to be optimized for your model is calculated at the end of each epoch. To callbacks, this is made available via the name "loss."
If a validation dataset is specified to the fit() function via validation_data or validation_split arguments, then the loss on the validation dataset will be made available via the name "val_loss."
.

3.4.2.1 ReduceLROnPlateau

Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This callback monitors a quantity and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced.
Function used in our model:
rlr = ReduceLROnPlateau(monitor="val_loss", factor=0.1, patience=3, verbose=0, mode="auto", min_delta=0.0001, cooldown=0)

Arguments
monitor="val_loss" : (Validation loss) is the quantitythat will be monitored here.
factor=0.1 :factor by which the learning rate will be reduced.
patience=3 : number of epochs with no improvement after which learning rate will be reduced.
verbose=0: quiet
mode="auto": in 'auto' mode, the direction is automatically inferred from the name of the monitored quantity.
min_delta=0.0001: threshold for measuring the new optimum, to only focus on significant changes.
cooldown=0: number of epochs to wait before resuming normal operation after lr has been reduced.

3.4.2.2 EarlyStopping:

It stops training when a monitored metric has stopped improving. Overfitting is a nightmare for Machine Learning practitioners. One way to avoid overfitting is to terminate the process early.
Assuming the goal of a training is to minimize the loss. With this, the metric to be monitored would be 'loss', and mode would be 'min'. A model.fit() training loop will check at end of every epoch whether the loss is no longer decreasing, considering the min_delta and patience if applicable. Once it's found no longer decreasing, model.stop_training is marked True and the training terminates.

Function used in our model:
es = EarlyStopping(monitor='val_loss', min_delta=0.00001, patience=15, verbose=1, mode='auto', baseline=None, restore_best_weights=False)

Arguments
monitor="val_loss" : (Validation loss) is the quantitythat will be monitored here.
min_delta=0.00001: Minimum change in the monitored quantity to qualify as an improvement
patience=15: Number of epochs with no improvement after which training will be stopped.
verbose=1: verbosity mode.
mode='auto': in "auto" mode, the direction is automatically inferred from the name of the monitored quantity.
restore_best_weights=False: The model weights obtained at the last step of training are used.

3.4.2.3 Model Checkpoint

Callback to save the Keras model or model weights at some frequency.
ModelCheckpoint callback is used in conjunction with training using model.fit() to save a model or weights (in a checkpoint file) at some interval, so the model or weights can be loaded later to continue the training from the state saved.

Function used in our model:

mc = ModelCheckpoint('best_model.h5', monitor='val_acc', mode='max', save_best_only=True,verbose=1)

Attributes:
filepath: string or Path
monitor="val_loss" : (Validation loss) is the quantitythat will be monitored here.
Verbose = 1: verbosity mode.
save_best_only = True: the latest best model according to the quantity monitored will not be overwritten.

4 Final Model

4.1 Data Preprocessing

In orignal data set, we had 4 columns: Short description,Description, Caller and assignment group.
3 new columns were added. Description_new , Descriptin_new1 and SD-DD
Description_new and Description_new1 are 2 cleaned version of Description. SD-DD is combintion of both Short Description and Description_new1 . SD-DD column was considered as training input going forward

```
In [0]: cleansed_data["SD - DD"] = cleansed_data['Short description'].str.cat(cleansed_data["Description"], sep= ' - ')
```

```
In [90]: print(cleansed_data.columns)
         cleansed_data.head(10)

         Index(['Short description', 'Description', 'Caller', 'Assignment group',
                'Description_new', 'Description_new1', 'SD - DD'],
               dtype='object')
```

Label encoder to convert group names to number
oneHotencoder = OneHotEncoder()#reshape the 1-D country array to 2-D as fit_transform expects 2-D and finally fit the object

```
In [94]: oneHotencoder = OneHotEncoder()

         #reshape the 1-D country array to 2-D as fit_transform expects 2-D and finally fit the object
         X = oneHotencoder.fit_transform(cleansed_data['Assignment group'].values.reshape(-1,1)).toarray()
```

Text samples in Input data was further vectorized into a 2D integer tensor.
Max words = 10000
Max sequence length = 150

In [0]:
```
from keras.preprocessing import text
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
```

In [0]:
```
max_words = 10000
# finally, vectorize the text samples into a 2D integer tensor
tokenizer = text.Tokenizer(num_words=max_words, char_level=False)
tokenizer.fit_on_texts(x_train_keras)
sequences_train = tokenizer.texts_to_sequences(x_train_keras)
sequences_valid = tokenizer.texts_to_sequences(x_valid_keras)
sequences_test = tokenizer.texts_to_sequences(x_test_keras)
```

In [43]:
```
MAX_SEQUENCE_LENGTH = 150

# pad sequences with 0s
X_train = pad_sequences(sequences_train, maxlen=MAX_SEQUENCE_LENGTH)
X_test = pad_sequences(sequences_test, maxlen=MAX_SEQUENCE_LENGTH)
X_valid = pad_sequences(sequences_valid, maxlen=MAX_SEQUENCE_LENGTH)
print('Shape of data tensor:', X_train.shape)
print('Shape of data test tensor:', X_test.shape)
print('Shape of data validation tensor:', X_valid.shape)
```

```
Shape of data tensor: (6141, 150)
Shape of data test tensor: (1275, 150)
Shape of data validation tensor: (1084, 150)
```

4.2 Glove Model:

In [47]:
```
# load the whole embedding into memory
embeddings_index = dict()
f = open('../content/drive/My Drive/Colab Notebooks/glove.6B.100d.txt')

for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs

f.close()
print('Loaded %s word vectors.' % len(embeddings_index))
```

```
Loaded 400000 word vectors.
```

In [48]:
```
size_of_vocabulary=len(tokenizer.word_index) + 1 #+1 for padding
print(size_of_vocabulary)
```

```
13115
```

```
In [0]: model=Sequential()
        #embedding layer
        #model.add(Input(shape=(150,)))
        model.add(Embedding(size_of_vocabulary,100,weights=[embedding_matrix],input_length=150,trainable=True))

        #Bilstm layer
        model.add(Bidirectional(LSTM(128,return_sequences=True,dropout=0.2,recurrent_dropout=0.2)))
        model.add(SeqSelfAttention(units = 32, attention_activation='relu'))

        #Global Maxpooling
        model.add(Flatten())

        #Dense Layer
        #model.add(Dense(256,activation='relu'))
        #model.add(Dense(128,activation='relu'))
        model.add(Dense(74,activation='softmax'))
        model.compile(loss='categorical_crossentropy',
                      optimizer='adam',
                      metrics=['acc'])

        #Adding callbacks
        #es = EarlyStopping(monitor='val_loss', mode='min', verbose=1,patience=3)
        es = EarlyStopping(monitor='val_loss', min_delta=0.00001, patience=15, verbose=1, mode='auto', baseline=None, restore_best_weigh
        rlr = ReduceLROnPlateau(monitor="val_loss", factor=0.1, patience=3, verbose=0, mode="auto", min_delta=0.0001, cooldown=0)
        mc = ModelCheckpoint('best_model.h5', monitor='val_acc', mode='max', save_best_only=True,verbose=1)
```

Validation Accuracy Attained: 68.7%

4.3 Word2vec Model

```
from gensim.models.keyedvectors import KeyedVectors
PROJECT_DIR = "/content/drive/My Drive/Colab Notebooks/"
EMBEDDING_FILE = 'SO_vectors_200.bin'
EmbeddingFile = PROJECT_DIR+EMBEDDING_FILE
word2vec = KeyedVectors.load_word2vec_format(EmbeddingFile, binary=True)
MAX_SEQUENCE_LENGTH = 30
MAX_NB_WORDS = 200000
```

```
model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_4 (Embedding)      (None, 200, 200)          2655000
_________________________________________________________________
bidirectional_3 (Bidirection (None, 200, 256)          336896
_________________________________________________________________
seq_self_attention_3 (SeqSel (None, 200, 256)          16449
_________________________________________________________________
flatten_3 (Flatten)          (None, 51200)             0
_________________________________________________________________
dense_3 (Dense)              (None, 74)                3788874
=================================================================
Total params: 6,797,219
Trainable params: 6,797,219
Non-trainable params: 0
_________________________________________________________________
```

Validation Accuracy: 70.4%

4.4 Fast Text Model

```
In [0]: max_features = 200000 #17780 #using all unique words
        embedding_dim = 300
        num_classes = 74
        batch_size = 32

        embedding_path = "/content/drive/My Drive/Colab Notebooks/wiki-news-300d-1M.vec"

        def get_coefs(word,*arr): return word, np.asarray(arr, dtype='float32')
        embedding_index = dict(get_coefs(*o.strip().split(" ")) for o in open(embedding_path))

        word_index = tokenizer.word_index
        nb_words = len(word_index)
        embedding_matrix_fast = np.zeros((nb_words + 1, embedding_dim))
        for word, i in word_index.items():
            embedding_vector = embedding_index.get(word)
            if embedding_vector is not None: embedding_matrix_fast[i] = embedding_vector
```

Model:

```
In [0]: model=Sequential()
        #embedding layer
        #model.add(Input(shape=(150,)))
        model.add(Embedding(size_of_vocabulary,300,weights=[embedding_matrix_fast],input_length=300,trainable=True))

        #Bilstm layer
        model.add(Bidirectional(LSTM(128,return_sequences=True,dropout=0.2,recurrent_dropout=0.2)))
        model.add(SeqSelfAttention(units = 32, attention_activation='relu'))

        #Global Maxpooling
        model.add(Flatten())

        #Dense Layer
        #model.add(Dense(256,activation='relu'))
        #model.add(Dense(128,activation='relu'))
        model.add(Dense(74,activation='softmax'))
        model.compile(loss='categorical_crossentropy',
                      optimizer='adam',
                      metrics=['acc'])
```

Validation Accuracy: 67.5%

4.5 Summary

Embedding	LSTM	BiLSTM	BiLSTM + Attention layer + Calbacks
Glove	57.6	60.2	68.7
Word2vec	59.3	61.8	70.4
Fastext	54.2	57.6	67.5

Validation Accuracy was considered to be the parameter to compare performance among models. And as per the table provided above, Word2Vec embedding layer along with BiLSTM, and Attention layer yielded the best model performance.

5 Implications

The model performance that we could achieve was 70.4%. Hence, inspite of the fact that it cannot replace the existing model where tickets are manually allotted to Assignment groups to be worked on, it can definitely help in faster decision making. This model can be implemented in two ways. One where it provides the person in charge with suggestions which would help the person in faster decision making. And other application could be where model directly forwards the ticket to recommended group. And if the group replies with a query, it can then be resolved by the personel.

With my experience, I would opt for the first application.

Along with deployment of this model, there needs to be a feedback mechanism where the personel overseeing the task corrects the incorrect suggestions and retrains the mode on regular basis.

6 Limitations

As we can see that the confidence level is 70%, there are chances that it would provide you with incorrect suggestions. This deficiency in the accuracy is mainly due to insufficient training data.

There are 25 groups that have less than 10 entries. Model for these groups may not have been proper

```
In [0]: labelencoder = LabelEncoder()
        cleansed_data['Assignment group'] = labelencoder.fit_transform(cleansed_data['Assignment group'])

In [0]: target_count = cleansed_data['Assignment group'].value_counts()

In [93]:
        (target_count<=10).value_counts()

Out[93]: False    49
         True     25
         Name: Assignment group, dtype: int64
```

Hence, in order to improve the accuracy, we would need more data to train the model. Especially for these 25 groups .

Also in future, if new problem resolving groups are formed, model will not be able to work for them. Hence, model will have to retrained for this scenario.

Model is highly dependent on problem description provided by user. If user fails to communicate the issue correctly possibly due to lack of proficiency in English, model would spit out an incorrect recommendation. Model should be retrained on a regular basis with newly generated data to account for incorrect suggestions and improving the overall efficiency of the system.

7 Closing Reflections

The process has been very instrumental for us in making us understand and learn real life application of Natural language processing. It has put us face to face with issues that we would face while implementation of NLP in real life. It has also given us an experience of the issues that we would face while working as a group. We have learned that sometimes, when we over clean the data, it would bring down the overall efficiency of the model. There is no direct formula or solution for solving a problem statement. They are unique and without proper understanding of data, It is difficult build a right model. The process helped us understand different embeddings that could be used to build a model. And all have their shortcomings. Hence, which embedding would work for us entirely depends on the data and problem statements. This has sharpened our skills and helped us in understanding importance and role of various concepts and hyperparameters.

8 Code Submission

As a part of submission along with Interim report we are submitting 4 python notebooks.

1. CapstoneProject_preprocessing.ipynb : This consists of data visualization, data cleaning and generation of feature vectors. This cleansed data is saved as cleansed_data.csv

2. Capstone_Modelling.ipynb: It reads cleansed_data.csv , runs TP-IDF vectorization and runs all models mentioned in Modelling section of this document.

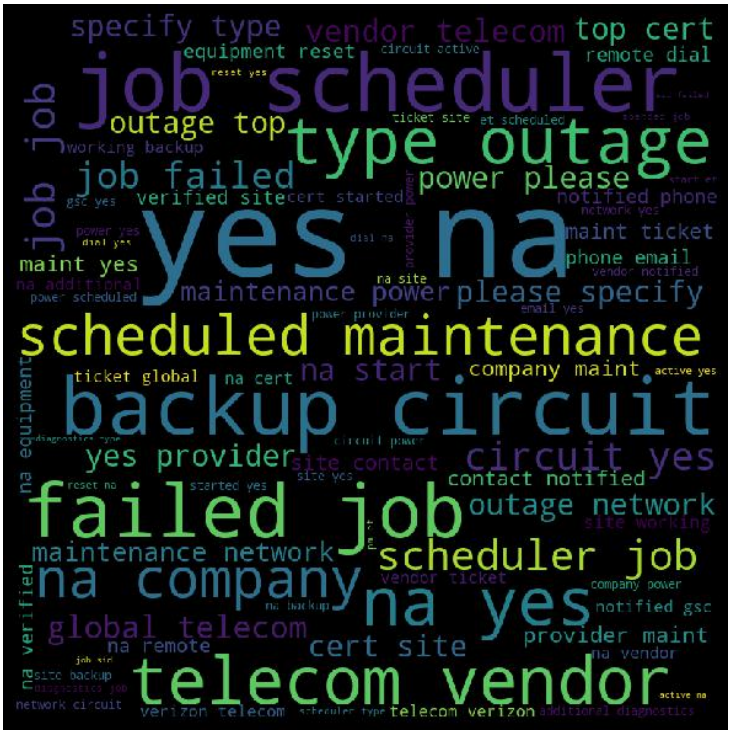
3. Embeddings_selfAttentionlayer.ipynb: This is a standalone document. It consists of data preprocessing and models with final model and embeddings .

9 Appendix

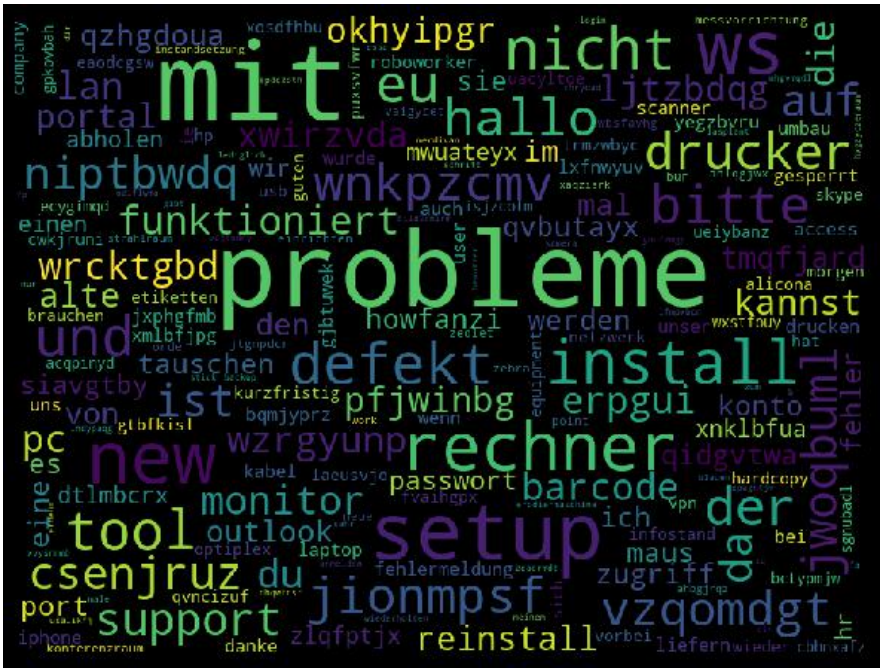
Wordclouds for top 10 groups:

Word Cloud of GRP_0: We can see that this looks lot similar to the word cloud of Description. Most of the issues look like they arise from inability to access an account.

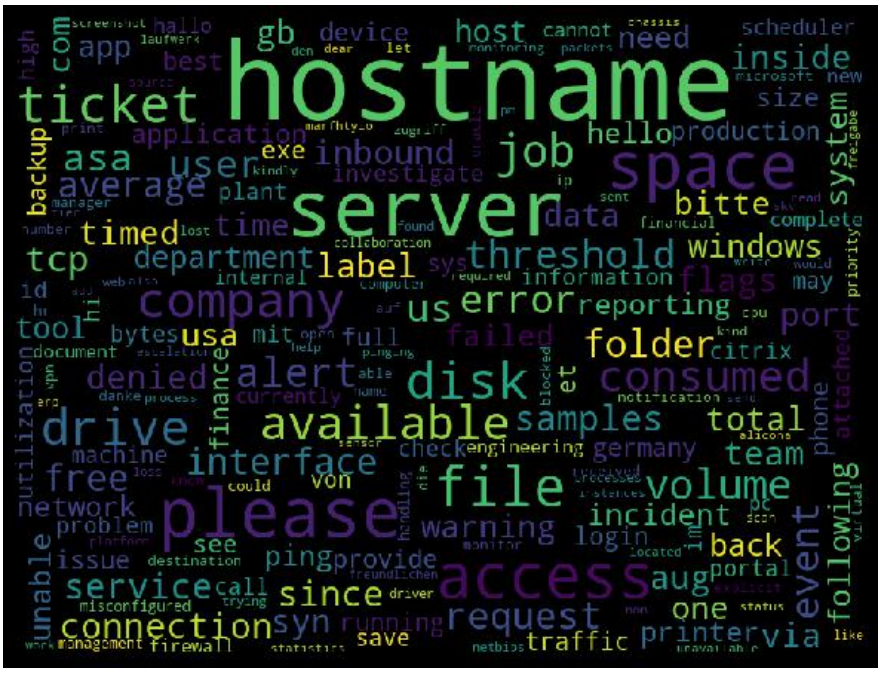
Word Cloud for GRP_8: most of issues are concentrated around job scheduler and failed job.



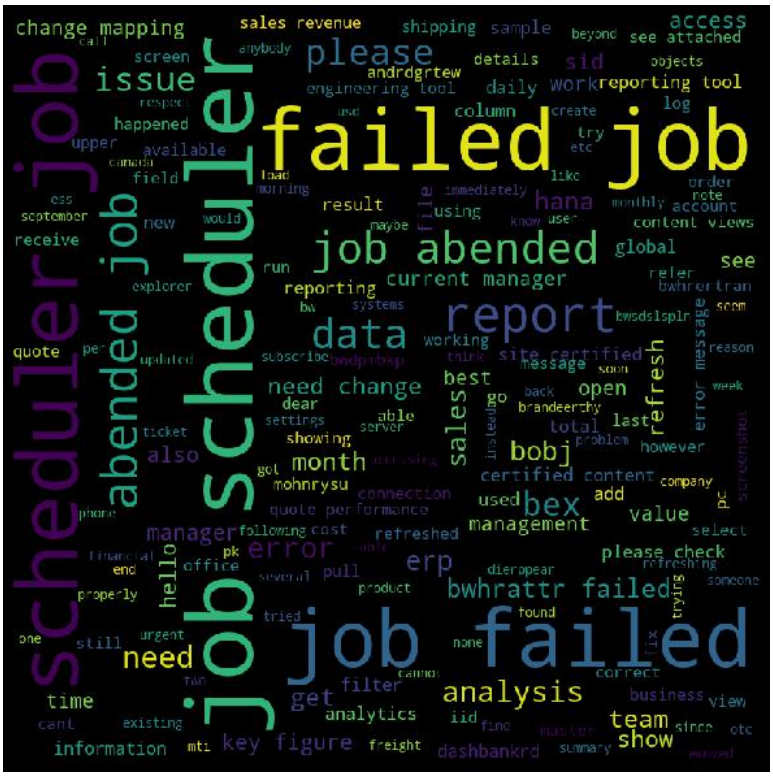
Word cloud for GRP_24: Issues consist of installation and setup related problems



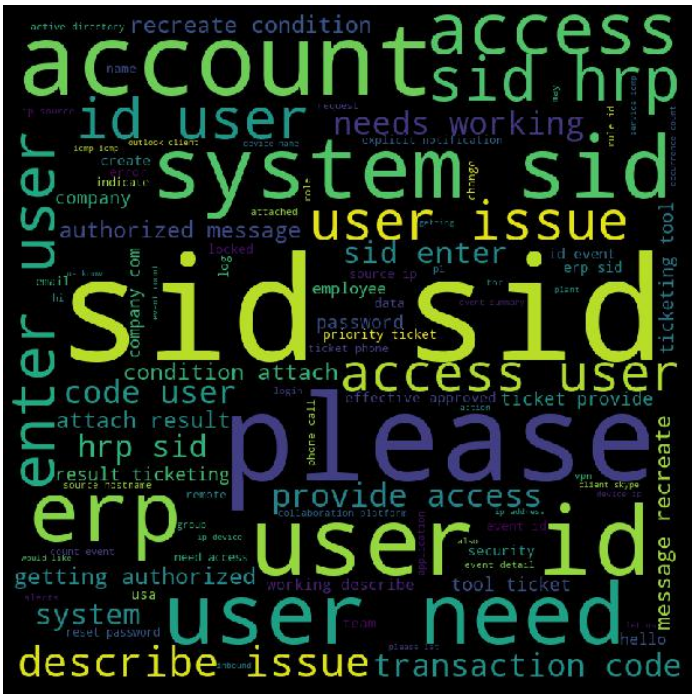
Word cloud for GRP_12: Maximum tickets describe server access and disk space to be the issue.



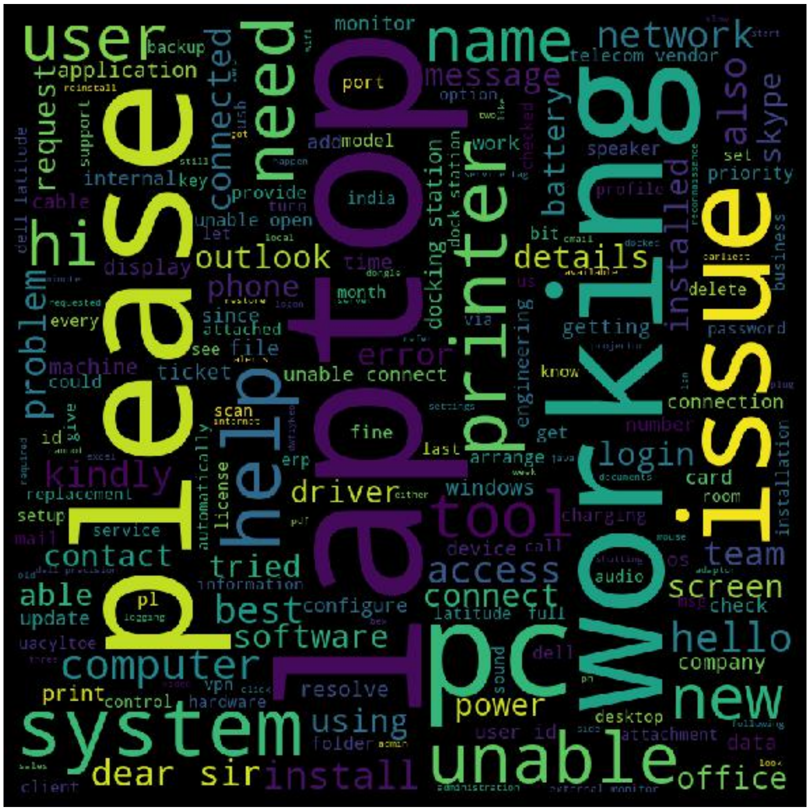
Word Cloud for GRP_9: This looks similar to group 8 with key words being job scheduler and failed job. Job numbers become important here. Also, most of the issues apear to be sales related.



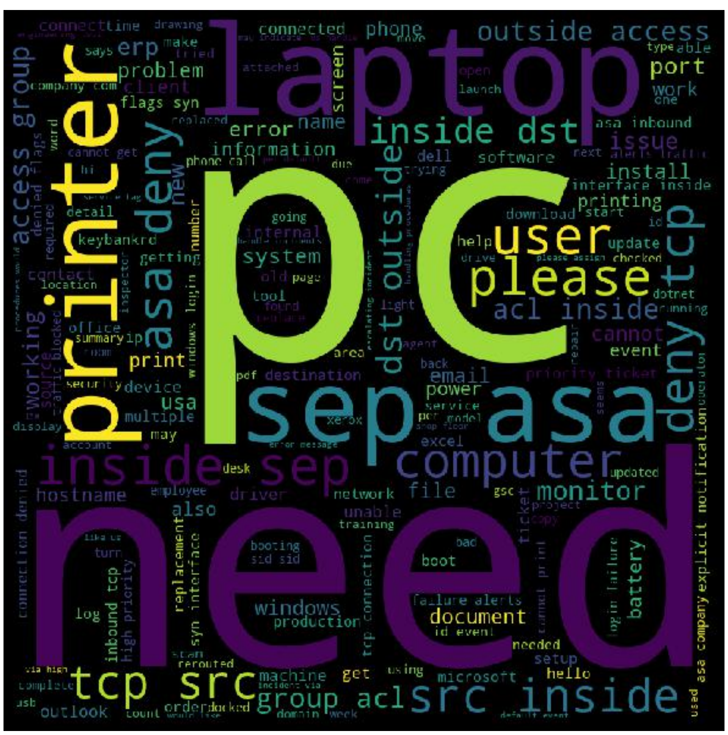
GRP_2: Keywords here are ‘sid’, ‘hrp’, ‘erp’. Therefore , we have to be careful with removal of non-english words. Also, embeddings to be selected needs to have this vocabulary.



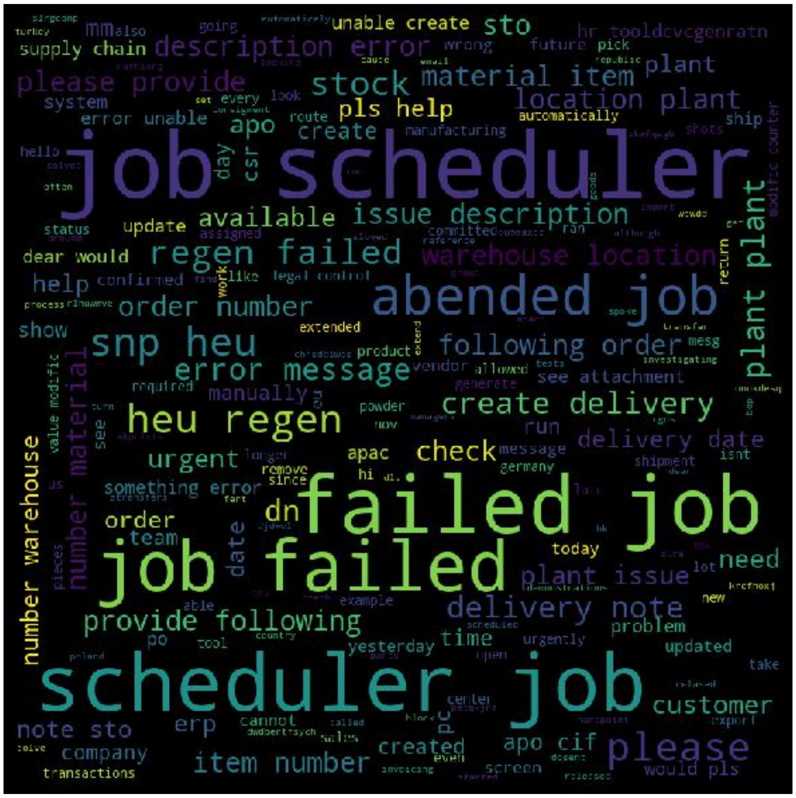
Word cloud for GRP_19: Most tickets are regarding laptop or printer not working.



Word cloud for GRP_3: We can see some non-english but frequently used acronyms like tcp, sep, src and dst. We have to be careful with them while data cleaning as well as closing an embedding source.



Word cloud for GRP_6: This also is similar to GRP_8 and GRP_9. Job numbers become important here as well. The tickets here appear to be raised by supply chain team.



Word cloud for GRP_13: Main keywords are order, billing, and shipment. This group looks to catering to order and billing related issues.