Executive Summary

Word embeddings are an extremely powerful concept in natural language processing, where they can be used to make mathematical representations of words in relation to other words in a vector form. They can be used directly in downstream natural language processing tasks as they encode this representation of the words, which is analogous to the meaning of the word. **The client** is developing a natural language processing interface called **ChatBot Application**, that aims to intuitively model an optimisation problem and retrieve the results. ChatBot Application's natural language processing engine makes use of word embeddings for processing the natural language input by a user.

Word embeddings are developed by training the vector representation of words on a large text corpus, where the final vectors produced are dense representations of the respective words, and the words are added to the model's vocabulary. This causes a limitation for word vectors to arise, as word embeddings are only available for words in the training text vocabulary. These words that are not part of the training text data are called Out of Vocabulary words. These words thus do not have an embedding and cause errors in the natural language engine as their representations are not available or is some generic embedding.

In this project, we deal with this problem of Out of Vocabulary words, by developing a model for producing an embedding by using the context of the word. The model is developed by leveraging tools in machine learning and artificial intelligence. Given an input sentence which contains a word that does not have a word embedding, the proposed model predicts new words in place of the out of vocabulary word, whose vector representations are used to produce an embedding for the out-of-vocabulary word. These predictions make use of the context of the out-of-vocabulary word through a probability distribution of words in the context sequence. This technique of generating word embeddings using probability distribution of words in a sequence is called language modelling.

Thus, the model developed in this thesis produces a meaningful embedding of words not in vocabulary. This is similar to how a human would infer the meaning of a word that is not in their vocabulary by understanding the context around the unknown word.

The model developed is efficient and can be used as an add-on with **ChatBot Application**'s existing natural language processing engine. It improves **ChatBot Application**'s performance by predicting a meaningful word embeddings for out-of-vocabulary words, when they are come across by the natural language processing engine.

Contents

1 Background

1.1.1 Problem Description

Word embeddings are an extremely powerful concept in natural language processing. They can be used directly in downstream NLP tasks since they encode the relationships between words (analogous to meaning) in the representations of the words themselves. This dense representation, when trained on large corpuses, can make the training of other NLP models where data is more limited much more effective. ChatBot Application's natural language interface makes use of word embeddings to understand words in conversation by the user and the ChatBot Application Agent.

A limitation of word embeddings is they are learned from the training data and therefore words must have been in the training data in order to have an embedding. The words that have embeddings after training consist of the words in vocabulary, and the words not in vocabulary are called out-of-vocabulary words.

1.1.2 Purpose of Project

The goal of this project is to create a model which, given an out-vocabulary word and the context in which it appears, produces an embedding which would be representative of that word, had it been in the training

data. Thus, the model efficiently predicts word embeddings for out-of-vocabulary words on the fly, based on the context words around it. The model is built on open-source libraries using the Python programming language.

1.1.3 Importance of the Project

ChatBot Application makes use of word embeddings for its natural language processing engine which aims to extract the intent and entities from input sentences. Thus, in conversational text, the natural language is bound to come across out-of-vocabulary words which are not understood by the NLP engine. This can cause problems for ChatBot Application with regards to intent and entity recognition for framing the optimisation problem from natural language. Thus, the proposed model aims to tackle this problem which results in the overall increase in accuracy of the NLP engine. The client will use the model to be incorporated into a production setting so as to enhance ChatBot Application.

2 Literature Review

This chapter summarizes the relevant research on natural language processing using deep learning, word embeddings and handling of out-of-vocabulary words.

2.1 Natural Language Processing

Natural Language Processing (NLP) is a branch of artificial intelligence that helps computers understand, interpret and manipulate human language through the study of interactions between computers and human natural languages (Kiser 2016). By utilizing NLP, developers can organize and structure language data to perform tasks such as automatic summarization, translation, named entity recognition, relationship extraction, sentiment analysis, speech recognition, and topic segmentation (Kiser 2016).

2.2 Deep Learning

Deep learning is a subfield of machine learning. Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction (Yann LeCun 2015). Thus, in contrast to machine learning, deep learning methods attempts to automatically learn good features or representations from raw inputs.

The main advantages of deep learning is that learned features are easy to adapt and fast to learn as opposed to manually designed features that often tend to be over-specific and require a long time to design and validate. Deep learning is a very flexible and almost universal learnable framework for representing linguistic information. It can also learn in both unsupervised and supervised contexts.

2.3 Neural Networks

With respect to Natural Language Processing, this report focuses on Neural Networks. Although human language is a symbolic and categorical signalling system, the brain encoding appears to be a continuous pattern of activation, and the symbols are transmitted via continuous signals of sound and vision (Socher n.d.). Thus, neural networks are best suited to explore this continuous encoding of natural language, as they are inspired by biological neural networks, and learn to perform tasks by examples. The neural network consists of simple elements called artificial neurons which receive input, change their internal state according to that input using an activation function and then produces an output depending on the input and the activation. There are a series of optimisation steps called forward and backward propagation where the output is optimised with the help of an error function.

2.4 Word Embeddings

Word embeddings in linguistics, aims to quantify and categorise similarities in the semantics between words based on their distribution with relation to context words in large samples of language data. In natural language processing techniques in the present context, word embeddings are vector representations of the words based on the occurrences of these words in the training data set. There is a lot of research being done and different techniques tried for word representations in a vector space, such as n-gram model, which is a probabilistic Markov model, and more recent techniques including the use of a neural network architecture. Different algorithms for word vector representations are discussed in the following sections.

2.5 Feedforward Neural Net Language Model (NNLM)

The probabilistic feedforward neural network language model has been proposed by Bengio et. al. where it aims to learn the joint probability function of sequences of words in a language. "*It consists of four layers input, projection, hidden and output layers. At the input layer, N previous words are encoded using one-hot encoding of dimension V, where V is size of the vocabulary. The input layer is then projected to a projection layer P that has dimensionality N × D, using a shared projection matrix. As only N inputs are active at any given time, composition of the projection layer is a relatively cheap operation. The values in the projection layer are dense. The hidden layer is used to compute probability distribution over all the words in the vocabulary, resulting in an output layer with dimensionality V*" (Tomas Mikolov 2013) .

2.6 Word2Vec

Word2vec model is the implementation of two log-linear neural network language model architectures published by Thomas Mikolov et. al. for learning distributed representations of words that try to minimize computational complexity (Tomas Mikolov 2013). The two architectures consist of shallow 2-layer neural network where vector representations depend on the context words. The model is trained on two steps: first, continuous word vectors are learned using simple model, and then an N-gram probabilistic feedforward neural network language model (Y. Bengio 2003) is trained on top of these distributed representations of words (Tomas Mikolov 2013). The two model architectures for learning word vectors are given below:

2.6.1 Skip-Gram Model

Skip-gram is to learn word vector representations that are good at predicting its context in the same sentence.

$$\text{maximize J} = \log \text{p}(w_{c-m}, \dots w_{c-1}, w_{c+1} \dots, w_{c+m} | w_c)$$

where w_c represents the centre word and $w_{c-m}, \dots w_{c-1}, w_{c+1} \dots, w_{c+m}$ represents the words surrounding the centre word. Given a sequence of training words $w_1, w_2, \dots, w_T$, the objective of the skip-gram model is to maximize the average log-likelihood:

$$= \sum_{t=1}^{T} . \sum_{j=-k}^{j=k} \log \mathrm{p}(w_{t+j}|w_t)$$

where k is the size of the training window and T is the total number of words. For the log probability term, every word w is associated with two vectors u_w and v_w which are vector representations of w as word and context respectively. (Tomas Mikolov 2013)The probability of correctly predicting word w_i given word w_j is determined by the softmax model, which is

$$\log p(w_i|w_j) = \frac{\exp \mathrm{s}(u_{w_i}, u_{w_j})}{\sum_{l=1}^{V} \exp \mathrm{s}(u_l, u_{w_j})}$$

Where V is the vocabulary size.Here $\mathrm{s}(u_{w_i}, u_{w_j})$ is called the scoring function where the score can be computed as the scalar product between word and context vectors (u_{w_i}, u_{w_j}) .The speed of training the Skip-Gram model can be accelerated using a hierarchical SoftMax function (Tomas Mikolov 2013). The vectors are tuned to maximise the average log-likelihood.

2.6.2 Continuous-Bag-of-Words (CBOW)

In the Continuous Bag of Words architecture, which is very similar to the Skip-Gram model except that instead of predicting the context words from the centre word, the centre word is predicted from the context words:

$$\text{maximize J} = \log \mathrm{p}(w_c|w_{c-m}, \dots w_{c-1}, w_{c+1} \dots, w_{c+m})$$

where w_c represents the centre word and $w_{c-m}, \dots w_{c-1}, w_{c+1} \dots, w_{c+m}$ represents the words surrounding the centre word (Tomas Mikolov 2013).

2.7 Global Vectors for Word Representation (GloVe)

Global Vectors for word Representation is a recent method for vector representation of words in a corpus that aims to improve on word2vec proposed by Jeffery Pennington et. al. by capturing the co-occurrence counts of words directly. It is a global log-bilinear regression model that combines the advantages of the two major model families in the literature: global matrix factorization and local context window methods (Jeffrey Pennington GloVe: Global Vectors for Word Representation). The model is efficient in leveraging statistical information gained from the corpus by training only on the nonzero elements in a word-word cooccurrence matrix (Jeffrey Pennington GloVe: Global Vectors for Word Representation). The model is efficient and produces meaningful vectors of the words as the co-occurrences of the words gives rise to more meaningful representation of words in vector space. “The relationship of words can be examined by studying the ratio of their co-occurrence probabilities with various probe words” (Jeffrey Pennington GloVe: Global Vectors for Word Representation).

Probability and Ratio	K=solid	K=gas	K=water	K=fashion
P(k|ice)	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
P(k|steam)	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
P(k|ice)/P(k|steam)	8.9	8.5×10^{-2}	1.36	0.96

Table 1: *"Co-occurrence probabilities for target words ice and steam with selected context words from a 6 billion token corpus. Only in the ratio does noise from non-discriminative words like water and fashion cancel out, so that large values (much greater than 1) correlate well with properties specific to ice, and small values (much less than 1) correlate well with properties specific of steam"* (Jeffrey Pennington GloVe: Global Vectors for Word Representation).

The above table demonstrates the co-occurrence matrix shown in the literature by Pennington et al. The vectors perform very well in analogous tasks such as similarity and named entity recognition.

2.8 Sub-word Model - Fasttext

FastText is an open-source, Natural Processing Language (NLP) library created by Facebook AI Research that allows users to efficiently learn word representations. The library was developed using models described in the paper 'Enriching Word Vectors with Sub-word Information, Piotr Bojanowski et. al.' The models described in the sections above learn vector representations of the word with respect to its context but ignore the morphology of the word itself as its assigns distinct vectors to each word. This is a limitation for languages with large vocabularies and rarer words. The paper proposes a new approach based on the skipgram model, where each word is represented as a bag of character n-grams. "A vector representation is associated to each character n-gram; words being represented as the sum of these representations. The method is fast, allowing to train models on large corpora quickly and allows to compute word representations for words that did not appear in the training data" (Bojanowski∗ 2016).

In the sub-word model used by Fasttext, it uses the skip-gram model described by Mikolov et al. in the above section, but instead focuses on the internal structure of the words. Each word w is represented as a bag of character n-gram. It also includes the word w itself in the set of its n-grams, to learn a representation for each word in addition to character n-grams.

For example, taking the word *what* and n = 3 as an example, it will be represented by the character n-grams: <wh, wha, hat, at> and the special sequence: <what>. Note that the '<' and '>' are included as prefix and suffixes for distinguishing from other character sequences. Thus, mathematically, the objective is similar to the Skip-Gram model where given a sequence of training words $w_1, w_2, \ldots, w_T$, the objective is to maximize the average log-likelihood:

$$\sum_{t=1}^{T}. \sum_{j=-k}^{j=k} \log \mathrm{p}(w_{t+j}|w_t)$$

where k is the size of the training window and T is the total number of words.

$$\log p(w_i|w_j) = \frac{\exp \mathrm{s}(u_{w_i}, u_{w_j})}{\sum_{l=1}^{V} \exp \mathrm{s}(u_{l}, u_{w_j})}$$

Where V is the vocabulary size. The difference in this sub-word model is that the scoring function $\mathrm{s}(u_{w_i}, u_{w_j})$ is different. Given a word w, let us denote the set of n-grams appearing in w by $G_w \subset \{1, \ldots, G\}$. A vector representation z_g is associated to each n-gram g. The sum of the vector representations of a words n-grams represents the word (Bojanowski∗ 2016). Thus, the scoring function is:

$$\mathrm{s}\left(u_{w_i}, u_{w_j}\right) = \sum_{g \in G_w} z'_g . v_j$$

where j represents the context. This model allows for sharing the vector representations based on sub-words across words, thus allowing to learn reliable representation for words that are out-of-vocabulary (Bojanowski∗ 2016).

3 Approach

This chapter presents the approach and the set up for the model to handle out-of-vocabulary words and the justification of the model chosen.

3.1 Out-of-Vocabulary Words

Out-of-vocabulary (OOV) words are words that appear in the testing text data but not in the recognition vocabulary where the text data is trained on. They are usually content words such as names and locations which are critical to the success to use case scenarios where natural language processing is applied. For ChatBot Application, the existing language processing engine uses a closed-vocabulary model that only recognises words in the model's vocabulary in the training data. It is not practical to list all words in a natural language for ChatBot Application's current model's vocabulary. When the user enters a sentence containing an out of vocabulary word in it, the existing Natural Language Processing Engine gives a null embedding to the word and tags it as an 'UNKOWN' word, for the words classification tag. This causes trouble for the downstream tasks for which ChatBot Application is intended to be used for. Thus, the OOV word's null embedding does not give it any position in the vector space and is overlooked, without any guesses on its embedding being made by the existing model. This problem is addressed using a language model using word sequencing on an existing word embedding model.

3.2 Overview of Model for Predicting Word Embeddings for OOV words

The model for predicting word embeddings for OOV words in a sentence is designed to make use of the context words surrounding the OOV word. A high-level summary of the approach is described below:

- Load a pre-trained word embedding model that contains a vocabulary of English words with mapped vector representations of their respective words. These vector representations are dense and have a meaningful representation of the word in the vector space.
- Given a sample sentence, the words in the sentence are tokenised and mapped to their word embeddings based on the vocabulary of the pre-trained model
- If a word is not in the vocabulary and thus does not contain a word embedding, the context of the word is input into a language model to predict the most probable word in place of the OOV word. This language model is designed using a recurrent neural network for text prediction given the context sequence of words.
    - Language modelling involves predicting the next word in a sequence of words. This prediction uses the estimated probability of a word. Recurrent neural are well suited for this task of predicting the probabilities of the next sequence unit based on the context.
- A weighted average of the predicted words' vector embeddings are taken to assign the OOV words word embedding. The assumption is that, given a large enough training data, the language

model will predict words that closely resemble the meaning or intent of the OOV word. This new word embedding is assigned to the OOV word and a position is assigned for it in the vocabulary.

3.3 Pre-trained Word Embedding Model

ChatBot Application provides a chatbot interface to for a user to input text data in natural language, which can be used on downstream tasks. To understand a given input text by the user, the input text is tokenised and then the individual tokens are mapped to an existing word embedding model pre-trained on a large vocabulary of words. These embeddings make it possible for ChatBot Application to understand the input text and correctly classify the intent and tags based on what the user is looking for.

For this project, a pre-trained model for word embeddings is used for to represent words in a vector space. A pre-trained model is a model that is already trained on a large text corpus, and thus have a large vocabulary and their respective embeddings. A pre-trained model is used because of the large computing power required for training a meaningful corpus. The corpus can be trained on either the word2vec model or GloVe model described in the sections (3.6) and (3.7) respectively.

3.3.1 GloVe algorithm

For this project, the GloVe (Global Vectors) algorithm is used for the initial pre-trained model for ChatBot Application. A basic overview of this model by Stanford's NLP team (Jefferey Pennington et al.) is discussed in the literature review section (3.7). The GloVe algorithm used in the pre-trained word embedding model used is described below:

- A co-occurrence matrix X is used to collect word co-occurrence statistics, where each element, X_{ij} of the matrix is used to represents how often word i appears in context of word j. A window of n words can be taken for the number of context words, with less weight being given for more distant words.
- For w_i as the vector for the centre word and w_j as the vector for the context word, a constraint is defined:$w_i'.w_j + b_i + b_j = \log(X_{ij})$
  Where $b_i \; and \; b_j$ are scalar biases for the centre and context words respectively.
- Define a cost function:

$$J = \sum_{i=1}^{V} . \sum_{j=1}^{V} f(X_{ij}) . \left(w_i' . w_j + b_i + b_j - \log(X_{ij})\right)^2$$

  Here f is a weighting function which helps us to prevent learning only from extremely common word pairs (Selivanov 2017).

Thus, in implementing GloVe vector model on a large text corpus, the word vectors can be used for meaningful representations of the words and this can be understood from mathematics of computational linguistics using the vectors built using a co-occurrence matrix. For example:

$$vector('King') - vector('Man') + vector('woman') = vector('Queen')$$

This is a very useful way to understand the words for analogous tasks and we can see that they perform well based on the co-occurrence matrix. Thus, the GloVe model is used as the basis for building the original vocabulary for this project. The model for handling the out-of-vocabulary words is discussed in the following sections.

3.4 Algorithms used in Model

A language model is a model of the probability of word sequences. A language model can be used to generate sequences. The probability of a sequence of m words $\{w_1, w_2, \dots, w_m\}$ is denoted as P$(w_1, w_2, \dots, w_m)$.

$$P(w_1, w_2, \dots, w_m) = \prod_{i=1}^{i=m} P(w_i | w_1, \dots, w_{i-1})$$

Traditionally, the most well-known approach to language modelling relies on n-grams. The limitation of n-gram language models is that they only explicitly model the probability of a sequence of n words. In contrast, Recurrent Neural Networks can model longer sequences and thus typically are better at predicting which words will appear in a sequence. Thus, a language modelling technique can be used to predict the vectors of the OOV words and thus give it a meaningful word embedding, based on its context.

3.4.1 Recurrent Neural Networks (RNN)

Recurrent Neural Networks are a general framework for modelling sequence data and are particularly useful for natural language processing tasks. At a high level, RNN encode sequences via a set of parameters (weights) that are optimized to predict some output variable. The advantage of RNN in language modelling is that, it can condition the model on all previous words in a corpus instead of a finite window.

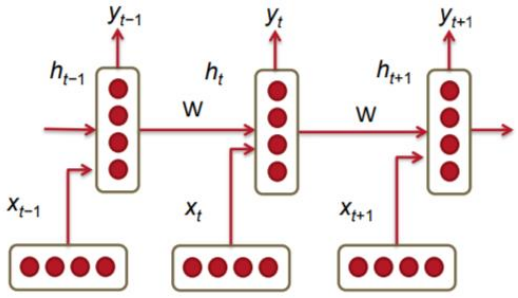

Figure 1 Timesteps in a Recurrent Neural Network (RNN) *(Socher n.d.)*

The above figure, shows the basic architecture of an RNN where each box represents a hidden layer consisting of neurons that perform a linear matrix optimisation on the inputs followed by a non-linear operation (e.g. tanh()). For each timestep, two inputs are taken h_t and x_t. h_t is the output of the previous step and x_t is the vector of the next word in the text corpus. The hidden layer then produces an output

y which is a prediction and the output features h_t. Below we give a detailed description of each parameter in the network:

- $x_1, x_2 \ldots x_t, x_{t+1} .. x_T$ are the word vectors corresponding the corpus with T words. In our case, the word vectors are obtained using the GloVe algorithm.
- $h_t = \mathrm{H}(W^{(hh)} h_{t-1} + W^{(hx)} x_t$ +b) ; This is the output features h_t and its formula where

$$x_t \in \mathbb{R}^{\mathrm{d}}: \text{input word vector at time t}$$
$$W^{hx} \in \mathbb{R}^{\mathrm{D_h} * d}: \text{Weight matrix for pruning the word vectors } \mathrm{x_t}$$
$$W^{hh} \in \mathbb{R}^{D_h * D_h}: \text{Weight matrix for pruning the previous step's feature output } \mathrm{h_{t-1}}$$
$$h_{t-1} \in \mathbb{R}^{D_h}: \text{Output of non} - \text{linear funtion at previous output. } \mathrm{h_0} \text{ is initialised}$$
$$H(): \text{non} - \text{linearity function, in our case an LSTM cell}$$

- $y_t = softmax(W^{(s)} h_t)$: y_t is the next predicted word based on the probability distribution over the vocabulary at each time-step t, given h_{t-1} and x_t. Also b is a bias. $W^{(s)} \in \mathbb{R}^{|V| * D_h}$ and $y \in \mathbb{R}^{|V|}$ where $|V|$ is the vocabulary. (Socher n.d.).

The weights W are updated through backpropagation through time, using a loss function to calculate the loss and update the weights at each timestep. The loss function used for our model is a cross entropy loss. Cross-entropy loss increases as the predicted probability diverges from the actual label and is well suited for our task of predicting the words in a sequence. The mathematical expression for the loss function is:

$$J^t(\theta) = -\sum_{J=1}^{|v|} y_{t,j} * log(\hat{y}_{t,j})$$

where $\hat{y}_{t,j}$ is the predicted value and $\mathrm{y_{t,j}}$ is the true value of the probabilty distribution of the word based on its context.

3.4.2 The LSTM Cell (Long Short-Term Memory Cell)

During the back-propagation phase, the contribution of gradient values, which are the values used to update weights in an RNN based on the loss function, gradually vanishes as they propagate to earlier time-steps in an RNN. This is due to recursive multiplication in the RNN for these gradients as the words are farther away in the context. The weights and the activation functions (or more precisely, their derivatives) affect the magnitude of the gradients. If either of these factors is smaller than 1, then the gradients may decrease exponentially in time; if larger than 1, then the gradient values increase exponentially. The gradients for words farther away are thus minimal or negligible and their weights are not update. Thus, the context words that are further away from the centre word are not considered due to their incorrect gradients.

This problem is tackled by replacing the regular RNN cell that computes h_t and y_t with an input, forget, and output gate, as well as a cell state. Each of these gates have their own set of weight values and the gradients are computed and these weights are updates. This allows backpropagation through the cell. Thus, the equation for calculating h_t is replaced from the one discussed above (Has¸im Sak 2014).

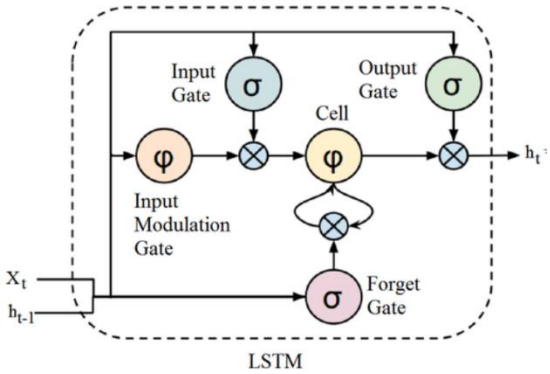

Figure 2 LSTM Cell and its gates (Jeff Donahue 2014)

The LSTM has the ability to remove or add information to the cell state, using the gates. At the 'forget gate' layer, the LSTM decides what information is thrown away from the cell state using a sigmoid layer.

Forget gate: $f_t = \sigma(W_f.[h_{t-1}, x_t] + b_f)$
Where σ() is the sigmoid function that outputs a number between 0 and 1. h_{t-1}, x_t are the same as described for RNNs above, and b is the bias. (Jeff Donahue 2014)

The next gate is the 'input gate'. It consist of two layers, a sigmoid layer which decides on which values to update and then a tanh() layer that creates a new vector of candidate values that are added to the state.

Input Gate : $i_t = \sigma(W_i.[h_{t-1}, x_t] + b_i)$
$q_t = \tanh(W_c.[h_{t-1}, x_t] + b_C)$

Here, q_t is the vector of candidate values and i_t is the sigmoid layer. (Jeff Donahue 2014)

The old cell state C_{t-1} is updated into the new cell state C_t using the following formula:

$$C_t = f_t * C_{t-1} + i_t * q_t$$

Thus, in the above equation, $f_t * C_{t-1}$ is used to forget unrequired values from the previous cell state and $i_t * q_t$ is the update to the state value (Jeff Donahue 2014).

$$o_t = \sigma(W_o.[h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

Finally, we have an output gate that gives us the h_t value, buy first a sigmoid layer and then a tanh() layer. Here, b is the bias, and we get the final output h_t (Has¸im Sak 2014).

3.4.3 Bi-directional RNNs with LSTM cells

For our model we use a bi-directional LSTM cell so that we can make use of not just the context of the words that come before the target word, but also of the words that come later, as a unidirectional RNN is only able to make use of its previous context. "*Bidirectional RNNs (BRNNs) do this by processing the data in both directions with two separate hidden layers, which are then feed forwards to the same output layer"* (Zhou Yu 2015)

$$\vec{h}_t = \mathrm{H}\left(W^{(\vec{h}\vec{h})}\vec{h}_{t-1} + W^{(\vec{h}x)}x_t + b_{\vec{h}}\right)$$
$$h_t^{\leftarrow} = \mathrm{H}\left(W^{(h^{\leftarrow}h^{\leftarrow})}h_{t-1}^{\leftarrow} + W^{(h^{\leftarrow}x)}x_t + b_{h^{\leftarrow}}\right)$$
$$y_t = softmax(W^{(\vec{h}y)}\vec{h}_t + W^{(h^{\leftarrow}y)}h_{t-1}^{\leftarrow} + b_y)$$

Here, H is the function for abstracting the LSTM cell function. B remains as the bias. The main changes is the inclusion of forward directional $\vec{h}_t$ and backward directional $h_t^{\leftarrow}$, and as result the new output y, which are computed using H() by iterating the backward layer from t = T to 1, the forward layer from t =1 to T and then updating the output layer (Zhou Yu 2015).

3.5 Model Selection

The model is built to produce embeddings for Out-of-Vocabulary(OOV) words depending on the OOV word's context. This is done using language model built using a Bi-directional Recurrent Neural Network with a Long-Short Term Memory cell. This language model is used for predicting the most probable word embedding for the OOV word based on its context, by predicting words in place of the OOV word and then taking a weighted average of their mapped word embeddings. This gives a word embedding for the OOV word which is reliable in terms of usability for entity recognition tasks and a meaningful representation for it in the vector space.

The Model for predicting OOV word embeddings consist of two segments: The first step is where the Model is prepared by tokenising, training and saving a model based on a training corpus, and the second step, which consists of the Prepared model being used to predict embeddings. The first step is called the Preparation Step and the second step is called the Embedding Prediction step.

3.5.1 Model Preparation Step

The details of the preparation for the OOV word embedding model is discussed in this section.

Model Preparation: Preparation of OOV word embedding prediction model

1. Input Text Corpus for Training
   1.1. Pre-process the training corpus to be used for line sequencing

2. Text Data Sequencing Function
   2.1. Tokenise the Input Text Data
   2.2. Retrieve Vocabulary of Input Data
   2.3. Encode the tokenised sequences of text into sequences of integers
   2.4. Prepare forward and backward sequences of text data
      2.4.1.Pad the forward and reverse sequences of text
      2.4.2.Split the sequences into input and output elements
   2.5. Return Input and Output elements of forward and reversed sequences

3. Define Prediction Model
   3.1. Create 3-layer Forward Sequential Recurrent Neural Network Model
      3.1.1.Add Embedding layer to vectorise the text data to a set dimension
      3.1.2.Add Bidirectional Long-Short Term Memory layer with set number of neurons
      3.1.3. Add Dense layer with SoftMax activation function
   3.2. Create the same 3-layered model with same properties for reverse sequence

4. Train the Forward and Reverse Sequenced Models
   4.1. Compile Model using Sparse Categorical Cross entropy as the Loss Function and 'Adam' optimiser
   4.2. Fit the model with set number of Batches and Epochs
   4.3. Save the trained Model

In Step 1 of the model preparation, shown above, a large corpus is pre-processed to be used for Step 2, where this corpus is tokenised, and the text is encoded as integers.

The tokenizer in Step 2 is used to fit the source text to develop the mapping from words to unique integers. Then these integers can be used to sequence lines of text. The size of the vocabulary of the text corpus is retrieved using the tokenizer to define the word embedding layer of the model inn Step 3.1.1. In Step 2.4, we prepare forward and backward sequences of the text data, where both the directions of the sequences are used for predicting embedding based on the previous and later words from the OOV word. The sequences are then split into input (X) and output elements (y) for each forward and reverse sequences to be used for training them on the RNN LSTM model prediction model. The forward and backward sequences are padded to have all sequences of the same length.

In Step 3, we define a model for each forward and backward sequences with 3 layers. The first embedding layer creates a real valued vector for each sequence. The second layer is the Bi-directional LSTM layer with a set number of units, which can be pruned to best fit the training corpus. The output layer is a dense layer comprised of one neuron for each word in the vocabulary. This layer uses a SoftMax function to ensure that the output is normalised to return a probability.

In Step 4, the encoded text for forward and backward sequences are compiled fit on respective RNNs. Since the Network is technically used to predict the probability distribution of the vocabulary based on the sequence provided, we use a sparse categorical cross entropy loss function to update weights on the network. The Adam optimiser is an efficient implementation of the gradient decent algorithm that is used to track the accuracy of each epoch of the training. The models is then saved to be used to return probability distributions of the vocabulary depending on forward or backward sequenced models.

3.5.2 OOV Embedding Prediction Step

After preparation and saving of the RNN models and the networks weights, we can use them for predicting the embeddings of OOV words in our pre-trained model. The details are discussed below:

OOV Word Embedding Prediction: Prediction of OOV word embedding

---

1. Load Models
   1.1. Load the pre-trained model trained on GloVe, from where OOV words are detected
   1.2. Load the forward and reverse sequenced models prepared using the LSTM RNN trained in the Model Preparation Step. Also Load tokenizer and sequence length used in this step.

2. Sequence Generating Function
   2.1. Takes an input seed text and model (either/or reverse or forward sequenced model)
   2.2. Predicts the probability distribution of words
   2.3. Returns predicted vocabulary

3. Input Sample Text

4. Function to set Embedding for OOV
   4.1. Checks for OOV words in Sample text
   4.2. If OOV word detected, embedding assigned to OOV word based on the context before and after the word from Generate Sequence Function (Step 2)
   4.3. Embedding for OOV updated and added to the pretrained model's (GloVe) vocabulary

---

The OOV Word Embedding Prediction step is shorter than the Model preparation step. Step 1, consists of loading all the models and parameters required to run the Embedding Prediction functions. In Step 2, the Generate Sequence function is used as a function that is called by Step 4's Set Embedding function to be

able to predict the most probable words that occur in place of the OOV word in the Sample text. These predictions are used to map to GloVe vectors of the predicted words and the Step 4 function defines a weighted average of the predicted words' embeddings. This Embedding is assigned to the vocabulary of our pre-trained model. This method of assigning embeddings is used so that, the OOV words will have a reasonable position in the vector space based on its context, even though it was initially not assigned an embedding.

4 Modelling

The model described in Section (4), Approach, is built using Python 3 and open-source libraries on Python. In this section, we take you through the model implementation and the analysis of the model itself.

4.1 Data Selection

The data used for training the RNN language model is the ROCStories Corpora available from the University of Rochester computer science department's website (http://cs.rochester.edu/nlp/rocstories/). The corpus contains 50,000 five-sentence common-sense stories. This corpus is unique in as it captures a rich set of causal and temporal commonsense relations between daily events (Nasrin Mostafazadeh 2016). This is useful for ChatBot Application's initial use case of using natural language for day to day location related tasks. This data allows the model to be versatile enough for a broad use case and is trained to reasonably predict OOV word embeddings for common queries on ChatBot Application.

4.2 Data Pre-processing

The ROCStories corpora was pre-processed on Python 3 using the Pandas Library. Pandas is an open source, python data science library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. The Corpora was available in .csv format, which was loaded as a data-frame using pandas, after which unwanted attributes of the data-frame were dropped. All the text in the Corpora was merged and loaded to a new .csv file which contained only the text data without any indices.

4.3 Model Implementation on Python

The model is built on Jupyter Notebook environment using Python 3 programming language. The model is built using open-source libraries for Python 3 and the pre-processed data. The pre-processed data is converted into tokenised integer sequences of text which is used to train on a Bi-directional Recurrent Neural Network with Long-short term memory cell. This RNN is the basis for the word predictions used which is in turn used to take a weighted average of their embeddings. The final embedding of the Out-of-

vocabulary word is obtained and is updated to the original pre-trained model used in production for ChatBot Application. The detailed walk-through is given in the sections below.

4.3.1 Python Libraries Used

A total of four libraries were used for building the complete model. The python libraries used and their features are described in this below:

4.3.1.1 SpaCy

SpaCy is an industrial-strength open source natural processing engine built for python for large-scale information extraction tasks. The version used was spacy 2.0 which features neural models for tagging, parsing and entity recognition of text. The library is used mainly for tokenisation of input text and to load pre-trained GloVe trained word vectors. Spacy is able to assign word vectors and context-specific token vectors while also being able use Part-Of-Speech tagging, dependency parsing and named entity recognition of text data. In short, Spacy is a full-fledged natural language processing tool. The built-in vocabulary of the GloVe model is referenced using SpaCy's Vocab function.

4.3.1.2 NumPy

NumPy is a fundamental package used for scientific computing on Python. It is used for vector manipulation tasks and for defining data-types such as arrays while building the model. NumPy is used for building the embedding structure.

4.3.1.3 Pickle

Pickle is a module used to implement serialising and de-serialising algorithm on a python object structure. The python object hierarchy is converted into a byte stream, which can then be 'unpickled' which is the inverse operation. The objects serialised in our code includes the tokenizer and the maximum sequence length of the text data.

4.3.1.4 Keras

Keras is a Python high-level neural networks API, that lets you quickly put together neural network models with a minimal amount of code. It is run on top of TensorFlow without needing to go into details of either of these underlying frameworks. It provides implementations of several of the layer architectures, objective functions, and optimization algorithms needed for building a model. For this project we use the Sequential structure of Keras which is a linear stack of layers. The layers used for the model include Embedding layer, Bi-directional Long-short term memory layer and a dense layer. Keras is also used for the sequencing the training text data.

4.3.2 Text Data Sequencing Function

The text data sequencing function (*data_sequencing*()) is the function used in the preparation of the Pre-processed text data so that it can be formatted as forward and backward integer sequences to be input for training.

Initially, the Tokenizer from Keras is used to fit the tokenizer on the text corpus. This allows us to encode the corpus into sequences of integers, where each integer is a mapping of a unique word in the corpus. The vocabulary size is retrieved using the length of the index of the tokenizer.

In this project, we split up the source text sentence-by-sentence and then break each line down into a series of words that build up. This allows the model to use the total context of a sentence when predicting the embedding for the OOV word. Note that, in this sentence by sentence representation, each sentence is required to be padded so that all the sentences meet a fixed length. This is done to meet the requirements of Keras API. Two types of sequences are built using the encoding, a forward sequence and a reverse sequence. The forward sequence is to predict the embedding for an OOV word based on the context before the OOV word, and the reverse sequence allows us to predict embeddings based on the context after the OOV word in a sample input sentence. After sequencing and padding the sentences, both types of sequences are split into input and output elements. This is done to provide training data where the input elements of the training (X) is the context words and the output (y) is the predicted word.

The function takes in the data as an input and returns the input and output elements of the forward and reverse sequences and also the maximum length of a sequence and the total vocabulary size of the text corpus.

4.3.3 Prediction Model Preparation and Training

The prediction model is defined as a stack of sequential layers using Keras, Sequential API. Using the Sequential API, the Recurrent Neural Network is built using three layers, an Embedding layer, a bi-directional LSTM layer and finally a dense layer. Two models are made, one for the forward sequences and the other for the reverse sequences. Both the models have the same definitions as described below. The only difference in these models is the input parameters for training.

The embedding layer takes the input integer sequences as input and produces distributed vector representations. The vector dimension defined for this model is a 100-dimensional vector. This can be changed depending on the vocabulary size. The embedding layer also takes in the maximum length of the input sequences found in the data_sequencing() function. This is required as a requirement for the Dense layer.

The second layer is the Bidirectional LSTM layer is the central component of the model. It observes each word in the story, it integrates the word embedding representation with what it's observed so far to compute a representation (hidden state) of the story at that timepoint. 100 LSTM units are defined for this model and the bi-directional aspect creates two copies of the hidden layer, one fit in the input sequences as-is and one on a reversed copy of the input sequence. This results in a total of 200 LSTM units. The output values from these LSTMs will be concatenated. The activation function used in these LSTM cells is the hyperbolic tangent.

The dense output layer outputs a probability for each word in the sequence, where each probability indicates the chance of that word being the next word in the sequence. The 'SoftMax' activation is what

transforms the values of this layer into scores from 0 to 1 that can be treated as probabilities. The Dense layer produces the probability scores for one particular timepoint (word).

The term "layer" is just an abstraction, when really all these layers are just matrices. The "weights" that connect the layers are also matrices. The process of training a neural network is a series of matrix multiplications. The weight matrices are the values that are adjusted during training in order for the model to learn to predict the next word.

The next step of the code is to compile and fit the network. In the compilation of the model for both the forward and reverse sequence-based model, we use a sparse categorical cross entropy loss function on which we update the weights using an optimiser called the 'Adam' optimiser which is an efficient gradient descent optimisation algorithm. Both the models use batch processing using a batch size of 100. A total of 200 epochs or iterations are used for the training. An increase in the number of epochs allows for increased accuracy of the model at a cost of time required for training. The forward sequencing model took 2 hours and 14 minutes for training and reached an accuracy of 86.15%. The reverse sequenced model took 2 hours and 34 minutes for training and reached an accuracy of 92.87%.

4.3.4 Sequence Generating Function

The sequence generating function (*generate_seq(model, tokenizer, max_length, seed_text))* is the function used for generating the word predictions for the sentence that contains an OOV word. It takes in the trained model, tokenizer, the maximum sequence length and the seed text as inputs and returns the predicted words based on the seed text. The seed text here is the context of the word, and is either the context words before the OOV word in the forward sequence model, or the context words after the OOV in a reverse sequence model. The reverse sequence first checks if the context is empty and if is not it undergoes a for loop which is used to predict the words based on the context.

The predict function available in Keras is used to predict the probabilities of the integer indices of the words, depending on the context words, which are tokenised using the same tokenizer used in training. The seed text is also padded with the same sequence length used in training for consistency and it is requirement for Keras prediction. The predicted probabilities of the indices is used to map to their respective words. Note that, a list of stopwords are excluded from the predictions to avoid insignificant predictions. The stopwords include common words such as 'and', 'the', 'a' and similar words that are filtered out. Finally, the function returns the list of predicted words as a string.

4.3.5 Loading Models and Setting Word Embeddings for OOV words

After training of the models and saving them, the models are loaded along with the tokenizer, stop-words and the maximum sequence length parameters. This is also where we load the GloVe word vectors are also loaded using SpaCy's loading function. The model loaded is the English model trained on OntoNotes, with GloVe vectors trained on the Common Crawl dataset. The Common Crawl dataset is a publicly available dataset that is a collection of text collected by scraping text data from websites on the internet. The GloVe vectors contain 20,000 unique, 300-dimensional word vectors that are pre-trained, so we can directly use them in production.

The function to set embeddings for OOV Words *(set_embedding_for_oov())* takes an input sentence and assigns a word embedding for an OOV word if it does contain one. The input sentence is first tokenised using SpaCy's natural language processing engine loaded on GloVe vectors. Thus the tokenised input sentence is mapped to their GloVe vectors. SpaCy's *token.is.oov* function allows us to check if a word in the sentence is Out-Of-Vocabulary. OOV words in SpaCy's model have a null embedding. If a token is OOV, then the context of the word in the sentence is taken and processed into the *generate_seq()* function to produce the most probable words given the context. The context before the OOV word is processed in the forward sequenced model and the context after the OOV word is processed in the reverse sequenced model. A null embedding vector that can be updated, is created to assign it to the OOV word. The predictions returned from the *generate_seq()* function are then taken and their embeddings from the GloVe model are taken, from the most probable word to the least (given a specified number of predictions, here we use 5) with decreasing weightage. The embedding is also weighted depending on the context window, where, the more weightage is given to the larger context (either the context size before the OOV word or after the OOV word). Thus, we have a final word embedding that is representative of the most probable vector in the word vector space for that context.

This embedding is then added to the original GloVe vector model using *nlp.vocab.set_vector,* which is function in SpaCy, that allows to update the embedding to the OOV word, and the OOV word is then added to SpaCy's vocabulary.

5 Analysis

For the Analysis of the OOV word embedding prediction model, we first do an analysis of the model based on the cosine similarities of the word embeddings of the OOV words in the GloVe vector space.

5.1 Cosine Similarity in SpaCy

The cosine similarity between two vectors is a measure that calculates the cosine of the angle between them. The cosine of the angle is a good metric to measure the orientation and not the magnitude of the two vectors compared. Thus, it shows how related two words are based on the angle between the vectors. A high similarity means that the vectors are oriented closely in the same direction and the angle is close to 0 degrees, and thus the cosine of the angle is near 1. Unrelated scores are nearly orthogonal and the angle between them is near 90 degrees resulting in a cosine of 0. Vectors oriented in the opposite direction have an angle between them that is near 180 degrees and have a cosine of -1. In SpaCy we compare the 300-dimensional word vectors and calculate the dot products divided by their vector norms to get the cosine similarity of two words:

$$\underline{numpy.dot(self.vector, other.vector) \ / \ (self.vector\_norm \ * \ other.vector\_norm)}$$

5.2 Word Embedding Prediction Performance

We use a sample sentence to test the prediction performance of our model.

5.2.1 Test case 1

Test Sentence 1: *"I live in lndn"*

The above test sentence contains an OOV word '*lndn'* which is a misspelt word for '*London'.* We first check the vector embedding for the misspelt word '*lndn'*:

```
In [6]: nlp.vocab.get_vector('lndn')

Out[6]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
               0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

Figure 3 Original Embedding for OOV word - 'lndn'

After passing the test sentence 1 through our model, we check for the resulting word embedding obtained.

```
array([ 1.8482960e+00, -1.3151566e+00,  5.1632800e+00, -2.5023196e+00,
        1.1026979e+01, -1.1978415e+00,  2.6360888e+00, -7.6715689e+00,
        1.8184205e+00,  1.2371289e+01, -1.8969524e+01, -6.9695215e+00,
       -4.0036454e+00, -3.4339972e+00,  2.2859437e+00,  2.0211749e+00,
       -1.6033973e+00,  2.5036875e+01,  7.0728431e+00,  1.6018537e+00,
        7.1240129e+00,  4.5292950e+00, -3.7383690e-02, -7.1894655e+00,
       -1.3374448e+00, -9.9555075e-01, -3.1635969e+00,  6.5285888e+00,
        3.5050064e-01,  6.0480785e+00,  1.6345665e+00,  4.3460326e+00,
        3.8556526e+00,  1.1056219e+01, -5.3953868e-01,  1.8110361e+00,
        7.4895191e-01, -3.0155444e+00, -2.2909701e+00,  1.6532058e+00,
        1.5836163e+00, -3.9497399e+00, -9.4094789e-01,  5.8741717e+00,
       -3.9443329e-01, -3.0442669e+00,  4.3613777e+00, -7.3590183e+00,
        3.3658335e+00,  2.9368665e+00, -7.5039077e+00, -3.0951777e+00,
       -2.7213774e+00, -6.5578384e+00, -5.4292530e-02, -2.7713127e+00,
        6.1229925e+00, -2.6328421e+00,  2.3735766e+00,  4.1633439e-01,
       -3.8031242e+00, -7.8631110e+00,  6.4053221e+00, -8.4930077e+00,
        2.0785544e+00,  5.7985425e+00, -1.8358672e+00,  6.6138186e+00,
       -5.4299636e+00,  3.2754543e+00, -1.5092303e+00,  1.5755850e+00,
       -2.8945620e+00, -1.8537233e+00,  1.0883554e+01, -1.2439799e+00,
       -1.6949245e-01,  8.0974758e-01,  3.5364826e+00,  6.3187199e+00,
        4.9646284e-02,  7.0220070e+00,  5.3657329e-01,  1.1027341e+00,
       -9.6876001e+00, -4.9361672e+00,  1.7104454e+01,  1.0535670e+01,
        7.1097982e-01, -4.5018582e+00,  6.2235584e+00, -3.9154274e+00,
       -9.3574798e-01, -1.0362184e+01,  4.9318762e+00, -1.2941527e+00,
        5.1670575e+00,  2.5040705e+00, -1.5726246e-02,  3.1281366e+00,
        2.5516391e+00,  2.8104773e+00, -8.3167875e-01,  8.2651490e-01,
       -1.9507954e-01, -1.5283947e+01, -2.1904199e+00,  7.6243162e+00,
       -3.2367599e+00, -6.8431282e-01, -3.0784252e+00, -3.2670405e+00,
        2.6581838e+00, -1.5642602e+00,  2.9371276e-01, -6.6515818e-02,
        7.1643782e+00,  3.8122594e+00, -5.7269406e+00, -4.8589315e+00,
       -4.7958355e+00,  1.2670391e+01, -6.6563994e-01, -5.0289164e+00,
        2.3440542e+00,  4.7287350e+00,  5.2303834e+00,  1.7109449e+00,
        4.2487607e+00, -1.2364076e+00,  5.4631014e+00, -1.0596882e+01,
        2.7188144e+00, -5.1134624e+00,  2.7940836e+00, -7.7249022e+00,
       -2.1537201e+00, -3.1962364e+00, -7.4686491e-01, -8.5264759e+00,
       -2.9372513e+01, -1.0098180e+01,  2.2274468e+00, -4.5758657e+00,
       -7.1605575e-01,  4.0997591e+00,  5.9630847e+00,  4.1604185e+00,
        4.5793921e-01, -3.5444698e+00,  3.8990026e+00,  2.9559038e+00,
       -5.6872015e+00,  1.0416037e+00,  6.5723300e+00,  3.2029819e+00,
       -9.5027614e-01,  2.8221412e+00,  8.1247479e-02,  8.3088903e+00,
        6.7680674e+00,  1.2070372e+00,  1.6571019e+00, -6.9955879e-01,
        8.1514578e+00, -3.3837357e+00,  3.5577788e+00, -5.0083113e+00,
       -2.4237785e+00,  7.5021634e+00, -8.3147507e+00, -3.8562052e+00,
       -5.4171901e+00,  6.0213935e-01, -6.5292301e+00, -2.8704681e+00,
        3.6445051e-01,  9.0859165e+00,  8.4192991e+00,  2.0967367e+00,
        4.8895974e+00, -1.0340326e+00, -1.5963291e+00,  2.8389599e+00,
        6.9602623e+00,  2.2386599e+00,  1.6808140e+00,  2.6220601e+00,
       -1.0288388e+00,  5.3553824e+00, -3.1175594e+00,  8.2345498e-01,
        7.2592058e+00, -3.9654045e+00, -2.8189547e+00,  4.2455926e+00,
        8.2732277e+00, -6.8713779e+00,  7.8006711e+00, -7.0584526e+00,
        2.6349974e+00, -5.7567263e+00,  2.7829823e+00,  4.8797774e-01,
        7.6649612e-01, -6.5310297e+00,  8.4381304e+00, -5.5461941e+00,
        1.1035798e+00,  6.0800397e-01,  2.0292904e+00,  1.0137126e+00,
        5.5548573e+00, -4.4183788e+00,  2.1456110e-01, -2.8779254e+00,
       -3.1559083e+00, -8.3407158e-01,  3.5955539e-01,  8.8355923e+00,
        2.6850083e+00,  1.5382688e+00,  4.8037499e-01, -7.1225109e+00,
       -4.2872510e+00,  2.2750561e+00, -5.6218119e+00,  2.8425195e+00,
        1.7085747e+00, -2.3440547e+00,  2.1727529e+00,  2.6691568e+00,
       -5.9693494e+00, -2.9368010e+00, -4.3602972e+00, -1.7149365e+00,
        1.0134034e+00, -8.3472073e-01,  3.5944989e+00,  2.7892847e+00,
        4.7319737e-01,  1.1024659e-01,  5.1601052e+00, -9.9358940e+00,
        1.3250442e+00, -3.7275391e+00,  6.3299775e+00,  4.9339804e-01,
       -7.4476237e+00, -1.2258607e+01,  5.2545152e+00,  4.5401511e+00,
        3.0327568e+00, -8.5919399e+00,  1.1318817e+00, -3.4164495e+00,
       -4.5391655e+00, -5.1926961e+00, -1.3630072e+00, -4.3402243e+00,
        4.3056002e+00,  5.4966784e+00, -4.2552900e+00,  5.3626838e+00,
       -2.9577262e+00, -8.8980751e+00,  6.2287421e+00, -1.8543151e-01,
       -8.1442795e+00,  9.8571596e+00, -1.8691559e+01,  1.9334948e+00,
        2.5822803e-01,  1.4323680e+01,  4.7923651e+00,  5.2803226e+00,
        7.7042909e+00, -4.2261887e+00,  4.2067237e+00,  6.8709960e+00,
        2.1975291e+00, -5.8194790e+00, -9.0879738e-01,  1.0939151e+01,
       -4.6263533e+00,  4.5168977e+00,  4.6697145e+00, -2.6080167e+00,
        2.2466002e+00, -3.0567882e+00, -1.9491187e+00,  7.8061275e+00,
        1.5211755e+00,  3.8697188e+00,  5.0852432e+00, -1.1735033e+00,
        2.5456285e+00, -2.5822015e+00, -8.1861191e+00,  5.9205465e+00],
      dtype=float32)
```

Figure 4 Final OOV word Embedding for 'lndn'

We see that clearly, the model has produced a 300-dimensional word embedding for the misspelt word '*lndn'*. Now we can compare the cosine similarities of the misspelt word with the word '*London',* before and after passing it through the model.

```
In [11]: nlp('lndn').similarity(nlp('London'))
Out[11]: 0.0
```

Figure 5 Cosine similarity of 'lndn' and 'London' before processing through OOV Model

```
In [11]: nlp('lndn').similarity(nlp('London'))
Out[11]: 0.9351419538514082
```

Figure 6 Cosine similarity of 'lndn' and 'London' after processing through OOV Model

We see that the word embedding produced by the OOV model for the misspelt word '*lndn'* is very similar to the word '*London'*, which shows that that the model performs quite well in this case. We can dig a little deeper into the produced embedding and check what words were used in the model to produce the embedding for *'lndn'* from the training data.

```
london
china
dresden
paris
```

Figure 7 Words used to produce the embedding for OOV word 'lndn'

Thus, it makes perfect sense that the word embedding produced is closely related to the word embedding for '*London*', due to weighted average taken from the words in Figure 7.

5.2.2 Test case 2

Test sentence 2: *'I play fidditch at school'*

In this test case, we use an OOV word '*fidditch'* which is a made-up game that is Out-of-Vocabulary. We test the model's context recognition is this test sentence. Based on a normal human's understanding of the context, we can ascribe '*fidditch'* as a sport or a game. The original word embedding for the OOV word is a null embedding.

```
In [24]: nlp.vocab.get_vector('fidditch')

Out[24]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

Figure 8 Original embedding of OOV word – fidditch

After passing it through the OOV model, we get the following word embedding:

In [31]:
```
nlp.vocab.get_vector('fidditch')
```

Out[31]:
```
array([ 6.95829821e+00,  2.58323145e+00,  4.13022232e+00, -2.82207894e+00,
        1.22050238e+01, -1.76323080e+00,  7.34122372e+00, -2.76011181e+00,
       -1.99373531e+00,  5.98183708e+01, -9.07633114e+00, -4.26140964e-01,
        8.36639786e+00,  6.84363127e+00,  5.71905375e+00,  1.00508451e+00,
        1.64460516e+00,  1.97753239e+01, -4.80817366e+00, -3.64908814e-01,
       -4.94153309e+00, -1.01224051e+01,  1.24436235e+01, -3.83875942e+00,
        6.56430244e-01,  1.11855803e+01, -6.92403507e+00,  2.84960055e+00,
       -3.52828813e+00,  2.85260391e+00,  1.64034700e+00, -1.75572252e+00,
       -1.02143236e-01, -4.52884167e-01,  9.89408135e-01, -3.32059598e+00,
        6.54138470e+00, -8.01946259e+00, -4.17164993e+00,  3.92733335e+00,
       -3.85233045e+00,  7.31532288e+00,  4.90277863e+00,  6.72401762e+00,
       -4.65730286e+00,  1.19491875e+00, -4.31131363e+00,  1.44764194e+01,
        3.01500940e+00,  1.13407791e+00, -3.74219060e+00, -2.01545906e+00,
        8.59033298e+00, -6.59692383e+00, -8.24048138e+00, -2.66290164e+00,
        6.07619619e+00,  7.25240946e+00,  3.59682488e+00,  3.44331694e+00,
       -7.10273933e+00,  1.15784378e+01,  3.58241749e+00,  8.87774563e+00,
       -2.20911801e-01, -3.70369434e-01,  4.03996277e+00, -1.38531494e+00,
       -8.29862773e-01, -4.89103031e+00, -3.91250062e+00, -5.22976923e+00,
        4.30083370e+00,  1.26126683e+00, -7.41654015e+00,  2.39310598e+00,
        8.79688978e-01,  8.07505989e+00,  9.00433600e-01,  1.04061108e+01,
       -1.36201692e+00, -8.79403496e+00,  2.35775858e-02,  9.08761120e+00,
        3.69774938e+00,  3.85806823e+00, -5.15039825e+00,  1.36443100e+01,
        3.69924927e+00, -3.73672795e+00, -5.29987812e+00,  2.87808347e+00,
       -5.65528011e+00, -4.82424593e+00, -2.79494315e-01, -1.51647367e+01,
        5.89940906e-01, -1.45358443e+00,  7.38790321e+00,  6.82484865e+00,
       -2.82205433e-01,  9.74566698e-01, -1.03781903e+00, -2.74089384e+00,
       -1.44550467e+00, -1.75624275e+01,  7.78010702e+00,  1.82207510e-01,
        3.63325238e+00, -1.48182368e+00, -3.97746062e+00, -9.57051849e+00,
       -4.68037748e+00, -3.12892103e+00, -1.96901038e-02,  2.29095793e+00,
       -2.13653469e+00,  3.60004878e+00,  1.30778599e+00,  4.51117373e+00,
        3.82346201e+00, -6.67311013e-01, -4.65420485e+00, -3.79636979e+00,
       -1.18041825e+00,  5.96622086e+00, -1.77109277e+00,  3.15216923e+00,
        1.10048676e+01,  2.28549004e+00,  5.56302071e+00,  1.27964246e+00,
        2.77819133e+00, -8.57870293e+00,  3.90102434e+00, -7.46688223e+00,
       -3.51520824e+00, -5.30754423e+00,  3.49084997e+00,  2.20415401e+00,
       -5.30653152e+01, -3.07123780e+00,  1.30861807e+01,  5.13597250e+00,
        3.00419474e+00, -2.40705729e+00,  9.87892437e+00, -2.69774103e+00,
       -4.43922997e+00,  6.10717058e+00, -1.05650415e+01,  3.33584642e+00,
       -1.30113673e+00,  2.93980885e+00, -3.15514541e+00, -2.12158179e+00,
        1.13915004e-01,  4.23778868e+00, -3.56217933e+00, -7.92874479e+00,
        3.14211488e+00, -2.60860801e+00,  9.71025467e+00, -1.53411512e+01,
       -1.37230265e+00,  4.55348730e+00, -3.35510635e+00,  4.38759136e+00,
       -6.25805044e+00, -1.47776246e+00, -1.08239126e+01,  4.97160181e-02,
        1.55002326e-01,  3.31458688e+00,  2.06117153e+00, -5.82048702e+00,
       -5.33283615e+00,  2.77122855e+00,  1.53066504e+00,  3.06864047e+00,
       -4.42576170e+00,  3.66640568e+00,  5.77280331e+00, -2.27866125e+00,
       -7.91539955e+00, -2.95015526e+00, -1.51111317e+00,  1.49531021e+01,
       -9.66643095e-01, -4.91072035e+00,  4.54741955e+00, -2.43272781e+00,
        3.54963279e+00, -6.69958591e+00, -4.89763737e+00,  8.32266331e+00,
        7.57167101e+00, -4.20205021e+00,  2.85678840e+00,  7.42751598e+00,
        1.34764137e+01,  1.72214484e+00,  5.72230816e+00, -2.96877432e+00,
       -1.00928659e+01, -6.22088099e+00,  2.08562660e+00,  9.07909989e-01,
       -3.00422955e+00, -3.29092002e+00, -5.51162291e+00,  2.65437555e+00,
        4.91254902e+00, -3.14290380e+00, -7.07065105e+00,  6.65862179e+00,
       -6.30500674e-01, -3.90444100e-01, -7.79761124e+00, -7.36342907e-01,
        2.35629177e+00,  1.73564339e+00, -6.08027506e+00, -4.91551352e+00,
       -1.36411228e+01,  3.99192929e+00, -1.43584251e+00,  4.08610201e+00,
        1.28846331e+01,  1.50684853e+01, -3.94741297e+00,  2.87995815e+00,
       -4.66738367e+00,  3.21859622e+00, -3.29582238e+00, -3.43916512e+00,
       -3.75561953e+00,  7.89628458e+00, -1.23473716e+00,  9.63781643e+00,
       -3.37609267e+00, -8.34144402e+00,  6.74943018e+00, -4.72857428e+00,
       -4.71439791e+00, -2.07268506e-01, -3.79912138e+00, -2.24355608e-03,
       -1.16510258e+01,  1.04663401e+01,  6.66904402e+00, -3.34221816e+00,
       -1.77800655e+00, -1.55149722e+00,  1.38535345e+00,  1.37720454e+00,
       -1.66190088e+00, -6.23627472e+00, -5.26695204e+00,  1.15914166e+00,
        2.03021584e+01,  2.61268687e+00, -3.59373593e+00, -3.32736671e-01,
       -3.98770618e+00,  5.09707257e-02,  5.90285420e-01,  3.56445909e+00,
       -1.36377543e-01,  1.28130031e+00, -9.58538723e+00, -1.91963863e+00,
       -1.59308887e+00, -2.51103425e+00, -1.99639165e+00,  3.55054617e+00,
        4.97302485e+00, -4.77221459e-02, -5.91505527e+00, -9.94359255e-01,
       -5.71868277e+00, -1.39226246e+00, -1.00212879e+01,  7.35589409e+00,
       -8.34264660e+00, -5.04246712e+00, -5.46081161e+00, -3.28095889e+00,
       -2.15234947e+00, -7.36108780e+00, -9.15419579e+00, -6.40454674e+00,
       -1.90165687e+00, -1.78464451e+01,  3.37746453e+00, -2.91865087e+00,
       -6.12448597e+00, -1.06078596e+01,  5.66841698e+00,  1.57054472e+00],
      dtype=float32)
```

Figure 9 Produced embedding of OOV word – fidditch

We obtain the most similar words to *fidditch* based on the produced embeddings and get the following results:

```
In [32]: most_similar(nlp('fidditch'))

Out[32]: ['SPORT',
          'sportive',
          'Sportsman',
          'SPORTSMAN',
          'sportsman',
          'Sport',
          'sport',
          'SPORTS',
          'sports',
          'Sports']
```

Figure 10 Most similar words to produced embedding of OOV word 'fidditch'

Thus we see that the model has accurately produced a word embedding based on the context and assigned in close to the word vector 'Sport'. Once we take a deeper look into which words were used to produce such an embedding we get the following words:

```
In [45]: set_embedding_for_oov(test2)

         Words predicted from forward sequence model:
         video
         basketball
         two
         months
         going
         Words predicted from reverse sequence model:
         sport
         hockey
         attendance
         race
         continued
```

Figure 11 Print of words used in making the word embedding for OOV word 'fidditch'

We see from the above words displayed that both the forward sequence and reverse sequence models were used to predict words using the context words before and after the OOV word. Their embeddings are weighted to get the word embedding for *fidditch*.Also the cosine similarity of *'fidditch'* and *'sport'* we get the following score:

```
In [46]: nlp('fidditch').similarity(nlp('sport'))

Out[46]: 0.845331215478122
```

Figure 12 Cosine similarity of 'fidditch' and 'sport'

From the produced embedding for *fidditch* , we see that the model does a good job in placing it in reasonable position in the vector space of the existing vocabulary by making use of the context.

5.3 Comparison of OOV Model with Fasttext Model

FastText was introduced and discussed in Section 3.8, where we see that the FastText Model created by Facebook AI Research that allows users to efficiently learn word representations using vector representation associated to each character n-gram. Each word is represented as a bag of character n-grams. We compare our OOV model's word embeddings of OOV words to FastText's OOV word embeddings. Since FastText does not require context and infact relies on the morphology of the word itself, the vector spaces are different as compared to the GloVe vectors we use for our OOV model. Thus, we compare the word embeddings with their own respective model using cosine similarity.

5.3.1 FastText Model Implementation

FastText model was implemented using Gensim, which is an open-source library that provides the implementation of FastText.

At first, the FastText model is trained using the same pre-processed text data corpus from 'ROCstories as discussed in section 5.2. We then remove full stops and turn the corpus to lower case. Gensim's LineText function is used to prepare the sentences for training using the FastText algorithm. The total vocabulary obtained from the model is 42,636 unique words. Using the Gensim wrapper to train the words for Fasttext is very straightforward. The vector dimension of 300 is defined for the embeddings and the model trains using the character n-grams of the words with an iteration count of 10. The model trains very fast on the corpus and takes a total training time of 9 minutes and 14 seconds. This is extremely fast compared to the OOV model used in Section 5. The model is saved after training. The FastText model using Gensim is straightforward and simple to implement.

5.3.2 Word Embedding prediction performance

Since the morphology of the words are considered for FastText, sentences are not considered and only OOV words are directly plugged in to the model to produce an embedding.

5.3.2.1 Test case 1

Using similar test case as used for in section 6.2 for our OOV model, we first use the misspelt word '*lndn'.* First to check if the word is OOV, we use the following command:

```
In [17]: test1 = 'lndn'

In [18]: test1 in model.wv.vocab

Out[18]: False
```

Figure 13 Command to check if word is OOV in Fasttext for word 'lndn'

We see the word is not in the vocabulary. The produced embedding for the word which is computed on the fly is given below:

```
In [19]: model.wv[test1]
```

```
Out[19]: array([-2.79542599e-02, -4.20174263e-02,  1.80512201e-02,  1.79043468e-02,
                -6.68379664e-02,  1.38272017e-01,  1.54858246e-01, -7.67455064e-03,
                 6.37950236e-03,  4.37940173e-02,  6.89615533e-02, -9.60793421e-02,
                 1.12054795e-01,  1.56448811e-01,  9.62309614e-02, -5.11019751e-02,
                 2.52481010e-02,  1.32545233e-01, -1.05296604e-01, -7.78720602e-02,
                -2.17445910e-01,  3.84036936e-02, -1.70118839e-01, -8.25022385e-02,
                 1.22044303e-01,  1.64153636e-01,  7.95028955e-02, -2.70458043e-01,
                -2.11668864e-01, -5.72521053e-02,  6.35768250e-02, -2.35330611e-02,
                 2.30547283e-02, -1.61425527e-02,  2.34169751e-01,  1.05797522e-01,
                -2.37371206e-01,  9.70451385e-02, -1.58180431e-01,  1.58832557e-02,
                 4.48568054e-02, -2.69402843e-02, -4.42920113e-03,  3.33548822e-02,
                -3.01429704e-02, -6.90126121e-02,  1.74546055e-02, -2.33406663e-01,
                -6.54229820e-02, -9.37774926e-02, -8.94195065e-02, -4.40857373e-02,
                 1.27490565e-01,  7.58950040e-02, -3.76167297e-02, -1.25920862e-01,
                 2.96673980e-02,  2.55031615e-01, -6.86725453e-02, -1.04761690e-01,
                 5.73417321e-02,  1.97710954e-02,  8.85242298e-02,  5.26103526e-02,
                 1.27041927e-02,  1.69036433e-01, -9.81013551e-02,  5.11177517e-02,
                -1.42808780e-02, -8.33968744e-02, -1.08593017e-01, -6.68538138e-02,
                -3.38468850e-02, -2.38524713e-02, -1.27879947e-01,  5.82144707e-02,
                -3.81892063e-02, -1.24770008e-01, -4.16994765e-02,  8.30521062e-02,
                -1.45696864e-01,  5.02537703e-03, -1.85143307e-01,  1.28429858e-02,
                 1.54788718e-01,  9.48998928e-02, -9.91076604e-02,  3.07262074e-02,
                -7.46448264e-02, -1.45565912e-01,  8.09971318e-02, -1.02394082e-01,
                 1.80013806e-01, -3.50740948e-03, -9.78233516e-02, -2.67315023e-02,
                 1.14288693e-02,  1.30662575e-01, -3.07597145e-02,  1.00301551e-02,
                -1.85949933e-02,  3.02854925e-03, -2.39475772e-01, -1.03280842e-01,
                -4.14110757e-02, -1.07786208e-01,  1.72332853e-01,  1.48586228e-01,
                -5.59800677e-02, -1.01770714e-01, -1.40456497e-04,  7.08161592e-02,
                -6.00265600e-02,  5.70974834e-02, -1.39662713e-01, -1.36665432e-02,
                -3.70896095e-03, -2.66202539e-01, -1.12115331e-01,  1.67837013e-02,
                 1.08639998e-02, -6.53696507e-02,  1.10218013e-02, -2.82608211e-01,
                -1.30668402e-01,  1.59877643e-01, -1.70715392e-01, -1.26638962e-02,
                 1.00714201e-02, -8.85518715e-02,  3.89847420e-02, -1.03914486e-02,
                -2.73582879e-02, -5.84724322e-02, -1.52965458e-02, -7.86596630e-03,
                -1.04470089e-01,  6.59773275e-02, -1.79376543e-01, -1.75102025e-01,
                 3.86788025e-02,  1.10110536e-01, -1.26056150e-01,  2.20657960e-02,
                 1.19675398e-01,  1.27977245e-02,  1.01695769e-01, -1.13276228e-01,
                 1.39257144e-02,  3.22484449e-02, -1.09988369e-01, -9.57150087e-02,
                -1.00794202e-02, -3.03509999e-02,  2.03027800e-01, -1.83577985e-01,
                 9.22192559e-02,  8.35247710e-02,  1.49900258e-01,  1.01456545e-01,
                 5.66160074e-03,  7.51790479e-02,  1.04325771e-01,  1.00706615e-01,
                -1.80553749e-01, -2.13855994e-03,  1.19283445e-01, -8.37475583e-02,
                 1.03146628e-01,  8.73741731e-02, -6.43455833e-02, -2.44669896e-02,
                 1.35732234e-01, -1.22189552e-01, -4.22315784e-02, -3.45164584e-03,
                 6.11050986e-02,  1.02114992e-03, -9.21607912e-02,  2.17226171e-03,
                 1.49025127e-01, -1.69742107e-02,  1.18389361e-01, -1.67706655e-03,
                -1.28696844e-01, -5.02879210e-02, -4.16490622e-02, -1.44060329e-02,
                 1.48867831e-01,  6.42926842e-02,  8.50441903e-02, -6.81729913e-02,
                 7.16745034e-02, -6.78429902e-02, -6.06088005e-02,  6.50980175e-02,
                -2.89792158e-02, -1.78178381e-02, -2.56726481e-02,  1.78433374e-01,
                -1.70697849e-02, -1.83397811e-02, -2.82801073e-02, -6.66699335e-02,
                 4.96919341e-02, -1.04759008e-01,  7.98791796e-02, -2.75191575e-01,
                 1.02360494e-01,  1.18311912e-01,  2.76423007e-01,  1.01964548e-01,
                -2.44062804e-02,  5.96144944e-02,  2.92008847e-01,  1.91026106e-01,
                -2.12665573e-02,  4.73452024e-02,  1.39549840e-02, -6.54131696e-02,
                 1.45773709e-01, -9.51631814e-02, -5.66103719e-02,  5.24575077e-02,
                 1.81130752e-01, -2.22969595e-02, -4.99503920e-03, -1.29645959e-01,
                -3.72604057e-02, -4.91504855e-02,  4.41854820e-02,  1.35254441e-02,
                 7.28971213e-02,  9.40158218e-02,  5.72270108e-03, -1.92194462e-01,
                 1.41290769e-01, -8.33619535e-02,  7.24645928e-02,  2.21739739e-01,
                 1.12911843e-01, -1.41022161e-01,  1.53874010e-01,  2.37517003e-02,
                -2.76174366e-01,  5.48430942e-02,  6.53601587e-02, -1.80976354e-02,
                 8.39685574e-02,  8.15120190e-02, -5.21417661e-03,  1.44005135e-01,
                 1.56587139e-01, -3.86681445e-02, -8.48233104e-02, -4.52706292e-02,
                 9.76727828e-02, -1.14954695e-01,  1.76023915e-01,  2.25710589e-02,
                -7.28356466e-02, -1.16833881e-01,  9.67523605e-02,  1.45973824e-03,
                 7.70994946e-02, -5.22266328e-02, -1.57175615e-01,  7.05929622e-02,
                -3.47322747e-02,  3.31166647e-02, -9.16713476e-02, -1.42720699e-01,
                 1.11037269e-02,  7.85278529e-02, -2.98778743e-01,  2.35499758e-02,
                 4.47689518e-02, -1.09705515e-02,  9.48450491e-02, -1.22993216e-01,
                -1.68485880e-01,  4.15644310e-02,  3.53852250e-02,  1.23134248e-01,
                -2.65356302e-02,  4.03248370e-02, -3.98263708e-02,  4.89448160e-02,
                -8.10341761e-02, -1.09299943e-01,  2.44196765e-02, -5.86861596e-02,
                -4.73841280e-03, -1.72877614e-03, -4.57545929e-02, -2.54970372e-01,
```

Figure 14 OOV word embedding using Fasttext for word 'lndn'

```
In [21]: model.wv.similarity('London', test1)
Out[21]: 0.24964838920287968

In [23]: model.wv.most_similar(['lndn'])
Out[23]: [('ex-boss', 0.6343728303909302),
          ('boss,', 0.6315561532974243),
          ('boss!jen', 0.6305891275405884),
          ('grandnephew', 0.6271762251853943),
          ('telegraph', 0.6271493434906006),
          ('grandad', 0.6219912767410278),
          ('register,', 0.6207696795463562),
          ('assessor', 0.6191135048866272),
          ('boss', 0.6189303994178772),
          ('grandniece', 0.6179234981536865)]
```

Figure 15 Cosine Similarity of word 'lndn' using Fasttext model

From the above figure, we can see that the cosine similarity between the OOV word and its intended word '*London'* is quite low. We also see that the most similar words are also not similar in morphology. Thus, FastText performed poorly, due to no vowels being present in the word to make an accurate word embedding.

5.3.2.2 Test Case 2

In the second case we use the word 'fidditch' as used in sention 6.2 Test Case 2. Again this is an OOV word and a word embedding is produced. What we are interested in is the most similar words to the OOV word using FastText.

```
In [29]: model.wv.most_similar(test2)
Out[29]: [('ditch', 0.7729636430740356),
          ('van!mitch', 0.7682470679283142),
          ('fiddle', 0.7355583310127258),
          ('fiddle!dan', 0.7330726385116577),
          ('bassnectar', 0.7311386466026306),
          ('pitch', 0.7273550033569336),
          ('mitch', 0.7238729000091553),
          ('hitch', 0.7224185466766357),
          ('clutch', 0.7148953676223755),
          ('glitch', 0.7111034393310547)]
```

Figure 16 Most similar words to 'fidditch' using Fasttext model

Again, we see that since FastText does not rely on context, the most similar words are sorted based on morphology.

5.4 Results

Comparing both the OOV model and the FastText model, we see that these are two different approaches to produces word embeddings. Although FastText is faster than the OOV model in terms of training, there is no difference in speed observed when producing word embeddings. The benefit of FastText is that, since it relies on the morphology of the word itself and not the context, word embeddings can be produced without requiring any context. FastText does perform better for OOV words that are misspelt as the character n-grams closely resemble the original word. However, the OOV model designed in this project requires the context to determine the word embedding. The OOV model can produce more efficient embeddings if the use case is for analogy tasks. The OOV model's word embeddings are thus

better performers for entity recognition tasks, for which the final model will be used for in ChatBot Application.

6 Conclusion

Producing a word embedding for words that are not present in the training set is a complicated task, due to limitations in correctly representing the meaning of the unseen word as a vector. It also depends on the method used initially for producing the initial word vocabulary. In this report we focused on two types of word embedding algorithms; GloVe and FastText. The GloVe algorithm produces word embeddings based on a co-occurrence matrix of the word with respect to other words in context, while FastText does not consider the meaning of the word in its context, but the morphology of the word based on its character n-grams. In this report we focused heavily on the GloVe model as ChatBot Application is used for entity recognition tasks in its Natural Language Processing tasks, where the GloVe model is more suited for, due to its algorithm which produces word embeddings that have a good representation of the meaning.

The model to produce word embeddings for Out of Vocabulary(OOV) words in a sentence for a vocabulary using the GloVe word vectors is developed. The model uses a Bi-directional Recurrent Neural Network with Long Short-Term Memory cells to predict other words in place of the OOV word in a sentence. The prediction is based on the most probable replacement for the OOV word based on the context, which gives us a basic idea of what the word embedding could be, based on the context. This is similar to how a human would decode the meaning for an unknown word by using the context to infer its meaning. The OOV model is trained on a large text corpus that consists of common sentences to train it on regular natural language. This dataset is appropriate for ChatBot Application's use case of providing a Natural Language interface for its optimisation tools. The model is designed to be efficient and is saved for use in a production setting. In the analysis of the OOV model, we feed sentences as input into the model, from which the model automatically recognises for OOV words, and the produces embedding for the OOV word based on the context it is in. If not OOV word is present in the sentence, then all the words already exist in the vocabulary and embeddings are not produced. Thus, this model is effective.The OOV model is then compared to FastText, which produces word embeddings based on the character n-grams. The approach is very different to the GloVe algorithms, so the word embeddings produced cannot be compared. However, we use cosine similarity as a measure to analyse the word embeddings of the models. The OOV model's word embeddings produced have a more accurate representation of the meaning of the word, while FastText's OOV word embeddings only contain the character morphology information. Thus, the OOV model developed in this report is a better model for ChatBot Application's Natural Language Processing Engine.

6.1 Limitations and Scope for Improvement

Since the crux of the model is based on the function to predict replacement words in place of the OOV word, there is a lot of emphasis on the training data used to train the model. The training data plays a huge role and larger training corpus is needed for training the model for versatile use cases. If a sentence is input into the model where the sentence sequence is not recognised in training, the word embeddings produced could be of poor quality, where the representation of the word meaning might not be accurate. To avoid such cases a larger training text corpus can be used. This was not done in this project due to limitations of using a Laptop for building the model with low compute power.

Another limitation of the model is the sorting of the most probable replacement words predicted by the Recurrent Neural Network. In some cases, filler words and common conjunction words rank higher on the sorting for the words used for producing word embeddings. This is corrected using a list stop-words which are removed from the predictions. However, this cannot be fully avoided as other common words can also rank higher. The improvement for this can done by improving the weightage formula used in producing the OOV word's word embedding.

Another limitation of the model is that, only one sentence can be input at a time, because of the way the model was trained on single sentence sequences. This can be improved and a large document with multiple sentences can be input, by splitting up the document into sentences and then the sentences could be input as a stream to the model.

7 Reflective Chapter

This chapter discusses my personal reflections of the project including the problems encountered, how they were solved, and personal learning experiences. I consider the project in the context of my courses taken in my masters in Operations Research and Analytics.

7.1 Scope of the Project

Natural Language Processing(NLP) is an emerging field in the domain of artificial intelligence. Given lack of expertise or familiarity in this field, initially, it was difficult to narrow down the approach for this project. A lot of time was initially spent in understanding the problem definition and concept of vector representations of words. The terms of reference for the report was developed during this initial stage. After developing a basic understanding of natural language processing and word embeddings, the next steps involved understanding the software packages and libraries used by The client for their current NLP engine. Extensive review of the documentation of the NLP libraries specifically SpaCy and Gensim, allowed me to understand the software pipelines useful for the purpose of this project.

After understanding the context of the problem including implementations of NLP tools and the software pipelines used, the next step involved spending time researching the limitations of word embeddings in processing Out-of-Vocabulary(OOV) words and the methods implemented to tackle these limitations. Most of the research in addressing OOV words were domain specific and were not in line with requirements for the model defined by The client. A top down approach for domain research was undertaken which is reflected in the Literature Review Chapter (3). While researching different use cases of word embeddings, I came across statistical language modelling, which is mainly used in text generation techniques. This sparked the idea for a probabilistic model for word embedding prediction. After defining the focus, my research was focused on implementing a statistical model of text data, after which I was able to define and implement this model to be added to The client's existing NLP pipeline.

7.2 Discussion on Methodology

Implementing a statistical model for text data has multiple approaches. There was compelling research for using a Bayesian model to predict words based on word n-grams. However, the research on the accuracy and flexibility of using Recurrent Neural Networks was promising. The methodology of developing a probabilistic model using neural networks was decided based on their flexibility, promising research and my expertise with the topic through my MA429 course in Algorithmic Techniques for Data Mining with Professor Gregory Sorkin.

Once the approach of using a recurrent neural network to build a probabilistic model for word embedding prediction was confirmed, the code implementation of the model was studied. This implementation was based on The client current NLP model, and the same libraries and programming language was used. The initial vocabulary for the NLP engine was used accordingly, to which we update the predicted word embeddings based on the model developed.

7.2.1 Strengths of Approach

One of the major strengths of using a recurrent neural network based approach is that the model performs very well in understanding the relationship between words in a sentence. The context of an OOV word is understood based on the sequence and weights assigned based on this sequence through a neural network. The predictions for word embeddings are similar to how a regular person would predict the meaning of an unseen word based on its context that it is placed in. The model does a really good job of making use of the context of the OOV words.

7.2.2 Weaknesses of Approach

Since the crux of the model depends on the data used in the training of the recurrent neural network, this also becomes a major weakness for the model. Random text data cannot be provided as a training set for this model since the model is entirely dependent on the sequence of the words in a sentence in the training data. Another weakness of the implemented model in this report is that, the model is not trained on a large enough data set, due to limitations in compute power of my laptop. This can be easily corrected by using hardware with larger compute power. Also, the context is necessary for the word unlike the FastText model used for comparison, which does not require the context to produce embeddings for OOV words.

7.3 Interaction with The client

My supervisor at The client, Alex Lilburn, introduced me to The client's current tools used for their Natural Language Processing tools, which gave me a good head start to understand the problem's context. Weekly meetings were set up at The client's office initially in the first month, and bi-weekly meetings in the following months. The interaction with The client was mainly regarding the progress of my model and the requirements they were looking for in my model. Alex was helpful with providing references to software used and how to structure the code for the model. He was also my main source for understanding the organisational context of The client. The client was very flexible with the approach and data used for the project. The data used for training was found personally by researching text corpuses with meaningful, full length sentences. There was no deviation from the original terms of reference due to The client's experience in sponsoring projects and the periodic reviews on my progress.

7.4 Relevant MSc Courses

As mentioned before, my courses taken in LSE played a huge role in providing me with a familiarity into neural networks and machine learning. Specifically, ST447 Data Analysis and Statistical Methods, ST446

Distributed Computing for Big Data and MA429 Algorithmic Techniques for Data Mining allowed me to be familiar with different statistical learning methodologies and neural networks. ST447 Data Analysis and Statistical Methods course gave the required background for understanding the basics of probability distributions and their use cases, ST446 Distributed Computing for Big Data was helpful as it familiarised me with working with the Python programming language and efficient techniques in analysing large datasets. The most important course that helped me in finalising the methodology was MA429 Algorithmic Techniques for Data Mining.

7.5 Personal Learning

The complexity and the project and taking a deep dive into such a new field as Natural Language processing was quite challenging at first. This required me to amass a lot of knowledge in the field in order to implement a meaningful model that could be used by The client. Initially a lot of time was spent in literature review and understanding the field. This forced me to plan ahead to ensure that enough time was kept aside for model development, experimentation and tuning, and writing the report on these findings. I was able to successfully overcome these challenges thanks to Dr. László Vegh's input into time management and suggestions. Dr. David Newton's lectures and course material provided me clear guidelines on how to structure, manage my project and turn it into this valuable report. A lot of time spent on model development was discarded due to poor results of experimenting with different methodologies. These included experimenting with character based recurrent neural networks and Bayesian prediction models. This experience was still quite useful in learning the optics of NLP systems. Alex was very happy with the results and the model itself and will be implementing the model as an add-on to The client's NLP engine. The project was successful, and the results proved to be extremely promising for scaling the model. The success of this project has well-prepared me for future analytical engagements and I am more confident on tackling complex tasks such as this project. I was also able to learn a lot about The client's working culture, which helped in my understanding of client relationships and my overall personal growth.

8 References

Bojanowski∗, Piotr. 2016. “Enriching Word Vectors with Subword Information.” *Facebook AI Research.*

Has¸im Sak, Andrew Senior, Franc¸oise Beaufays. 2014. *Long Short-Term Memory Recurrent Neural Network Architectures.* Google, USA.

Jeff Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan,Kate Saenko, Trevor Darrell. 2014. “Long-Term Recurrent Convolutional Networks for Visual Recognition and Description.” *IEEE Transactions on Software Engineering* 99-108.

Jeffrey Pennington, Richard Socher, Christopher D. Manning. GloVe: Global Vectors for Word Representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* 1532–1543.

Kiser, Matt. 2016. *Introduction to Natural Language Processing (NLP).* 11 08. Accessed 08 2018.

Milad Mohammadi, Rohit Mundra, Richard Socher. 2013. *Deep Learning for NLP.* Notes, Stanford University.

Nasrin Mostafazadeh, Nathanael Chambers, Xiaodong He, Devi Parikh,Dhruv Batra,Lucy Vanderwende,Pushmeet Kohli,James Allen. 2016. “A Corpus and Cloze Evaluation for Deeper Understanding of Commonsense Stories.” *2016 North American Chapter of the ACL (NAACL HLT).* University of Rochester.

Samuel, Arthur L. 1988. “Some Studies in Machine Learning Using the Game of Checkers. I.” *Computer Games I* 335-365.

Selivanov, Dmitriy. 2017. *GloVe Word Embeddings.* 8 8.

Socher, Richard. n.d. “Natural Language Processing with Deep Learning.” *Stanford.edu.* Stanford University.

Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean. 2013. “Efficient Estimation of Word Representations in Vector Space.” 7 Sep.

Y. Bengio, R. Ducharme, P. Vincent. 2003. “A neural probabilistic language model.” *Journal of Machine Learning Research* 1137-1155.

Yann LeCun, Y. Bengio, Geoffrey Hinton. 2015. “Deep Learning.” *Nature* 463-444.

Zhou Yu, Vikram Ramanarayanan,David Suendermann-Oef,Xinhao Wang,Klaus Zechner,Lei Chen,Jidong Tao,Aliaksei Ivanou ,Yao Qian. 2015. “USING BIDIRECTIONAL LSTM RECURRENT NEURAL NETWORKS TO LEARN HIGH-LEVEL ABSTRACTIONS OF SEQUENTIAL FEATURES FOR AUTOMATED SCORING OF NON-NATIVE SPONTANEOUS SPEECH.” *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU).* Scottsdale, AZ, USA: IEEE.

9 Appendix

9.1 Project Terms of Reference

Sponsoring Organisation. The client. RocketSpace, 40 High St, Islington High St, London N1 8XB

Client Supervisor. Alex Lilburn

LSE Supervisor. Dr.Lazlo Vegh

Working project title. Language Modelling for Handling Out-of-Vocabulary Words in Natural Language Processing

Description of problem area. The client is developing a natural language interface (chatbot) that can interface with its location-based application to solve the travelling salesman problem. The client has decided to develop an inhouse Natural Language Processing (NLP) engine for the chatbot, which aims to extract the intent and entities from input sentences with the use of word embeddings, which is then used as input parameters for the location application. Word Embeddings encode the relationships between words (analogous to meaning) in the representations of the words themselves. A limitation of word embeddings is they are learned and therefore words must have been seen in the training data before, in order to have an embedding. There is a need to improve the NLP engine to account for words that are not in the training set as they either lack word embeddings or have some generic embedding.

Purpose of project. The goal of this project is to create a model which, given an unseen word and the context in which it appears, produces an embedding which would be representative of that word, had it been in the training data. This improves the overall accuracy of the NLP engine and the model will be incorporated into the NLP engine to enhance the chatbot. Different techniques and algorithms will be compared in order to select a well-performing model. This will involve understanding various of natural language processing tools and use of existing packages for model implementation.

Proposed method. The solution to the problem will be developed through the following steps:

- Undertaking literature search on word embeddings and word vectors
- Exploring different libraries and packages containing efficient implementations of vector space modelling and topic modelling
- Identifying the right algorithm and library for pre-trained word embedding data.
- Formulating the word embedding prediction model.
- Implementation and analysis of the model
- Results and conclusion

Data type and sources. Open-source data available online. Specifically, the data used for training the language model is the ROCStories Corpora available from the University of Rochester computer science department's website (http://cs.rochester.edu/nlp/rocstories/).

Hardware and software required and available.

- Personal computer
- Python 3 programming language
- Jupyter Notebook environment for python 3
- Open-source libraries for python 3
    - SpaCy
    - Keras
    - NumPy
    - Pandas
    - Pickle
    - Gensim

Deliverables.

- Paper/ electronic copy of LSE project report, excluding reflective chapter
- An efficient model along with the code implemented.

9.2 Glossary

CBOW Continuous-Bag-of-Words: The architecture for learning word embeddings using word2vec algorithm proposed by T. Mikolov et. al.

GloVe Global Vectors for Word Representation: The algorithm used for learning word embeddings for based on their occurrences with respect to other words in a text corpus.

LSTM Long Short-Term Memory: Units of a recurrent neural network, used for remembering values over arbitrary time intervals by regulating the flow of information into and out of unit cell.

NLP Natural Language Processing: A method of using artificial intelligence in computer science to study the interactions between computers and human languages, to process and analyse natural language.

NNLM Neural Net Language Model: a language model built using Neural Networks, exploiting their ability to learn distributed representations.

OOV Out of Vocabulary: Words that that are not present in the training data of a word vector model and are thus not in the vocabulary and do not have a word embedding assigned.

RNN Recurrent Neural Network: A class of neural networks where connection between separate nodes along a sequence are formed using directed graphs.

9.3 Implemented Code

The model is attached with this report as a Jupyter notebook so as to be able to be reproducible and and for the analysis. Below we show the implementation of the code

9.3.1 Data Pre-processing

```
import pandas as pd
df = pd.read_csv('ROCstories.csv', encoding='utf-8')
#remove unrequired columns
df=df.drop(df.columns[[0, 1]], axis=1)
df['Story'] = df.sum(axis=1)
#remove unrequired columns
leandata=df.drop(df.columns[[0,1,2,3,4]], axis=1)
cleandata=  cleandata.sum(axis=0)
#putput processed data as csv finle
cleandata.to_csv('cleandata.csv',index=False)
```

9.3.2 OOV Model using Language Modelling

9.3.2.1 Importing Libraries

```
#importing libraries
import spacy
from spacy.vocab import Vocab
import numpy
from numpy import array
from keras.preprocessing.text import Tokenizer
from keras.utils import to_categorical
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Bidirectional
from keras.layers import Embedding
from keras.models import load_model
import pickle
```

9.3.2.2 Reading Training Text Data

```
#reading processed data
data = open('cleandata.csv').read()
```

9.3.2.3 Text Data Sequencing Function

```
#function for preparing text data into sequences for training
def data_sequencing(data):
    # integer encode sequences of words
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts([data])
    with open('tokenizer.pkl', 'wb') as f: # Save the tokeniser by pickling it
        pickle.dump(tokenizer, f)

    encoded = tokenizer.texts_to_sequences([data])[0]
    # retrieve vocabulary size
```

```
    vocab_size = len(tokenizer.word_index) + 1
    print('Vocabulary Size: %d' % vocab_size)

    # create line-based sequences
    sequences = list()
    rev_sequences = list()
    for line in data.split('.'):
        encoded = tokenizer.texts_to_sequences([line])[0]
        rev_encoded = encoded[::-1]
        for i in range(1, len(encoded)):
            sequence = encoded[:i+1]
            rev_sequence = rev_encoded[:i+1]
            sequences.append(sequence)
            rev_sequences.append(rev_sequence)
    print('Total Sequences: %d' % len(sequences))


    #find max sequence length
    max_length = max([len(seq) for seq in sequences])
    with open('max_length.pkl', 'wb') as f: # Save max_length by pickling it
        pickle.dump(max_length, f)
    print('Max Sequence Length: %d' % max_length)

    # pad sequences and create the forward sequence
    sequences = pad_sequences(sequences, maxlen=max_length, padding='pre')
    # split into input and output elements
    sequences = array(sequences)
    X, y = sequences[:,:-1],sequences[:,-1]

    #pad sequences and create the reverse sequencing
    rev_sequences = pad_sequences(rev_sequences, maxlen=max_length, padding='pre')
    # split into input and output elements
    rev_sequences = array(rev_sequences)
    rev_X, rev_y = rev_sequences[:,:-1],rev_sequences[:,-1]

    return X,y,rev_X,rev_y,max_length,vocab_size
```

9.3.2.4 Returning values from text data sequencing function

```
#returning forward and reverse sequences along with max sequence
#length from the data

X,y,rev_X,rev_y,max_length,vocab_size = data_sequencing(data)
```

9.3.2.4.1 Output

```
Vocabulary Size: 3222
Total Sequences: 17310
Max Sequence Length: 35
```

9.3.2.5 Building Forward and Backward Sequence models

```
# define forward sequence model
model = Sequential()
model.add(Embedding(vocab_size,100, input_length=max_length-1))
#model.add(LSTM(100))
model.add(Bidirectional(LSTM(100)))
```

```
model.add(Dense(vocab_size, activation='softmax'))
print(model.summary())
```

9.3.2.5.1 Output

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 34, 100)           322200
_________________________________________________________________
bidirectional_1 (Bidirection (None, 200)               160800
_________________________________________________________________
dense_1 (Dense)              (None, 3222)              647622
=================================================================
Total params: 1,130,622
Trainable params: 1,130,622
Non-trainable params: 0
```

```
# define reverse model
rev_model = Sequential()
rev_model.add(Embedding(vocab_size, 100, input_length=max_length-1))
#rev_model.add(LSTM(100))
rev_model.add(Bidirectional(LSTM(100)))
rev_model.add(Dense(vocab_size, activation='softmax'))
print(rev_model.summary())
```

9.3.2.5.2 Output

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 34, 100)           322200
_________________________________________________________________
bidirectional_2 (Bidirection (None, 200)               160800
_________________________________________________________________
dense_2 (Dense)              (None, 3222)              647622
=================================================================
Total params: 1,130,622
Trainable params: 1,130,622
Non-trainable params: 0
```

9.3.2.6 Compiling forward and reverse sequencing models

```
# compile forward sequence network
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit network
model.fit(X, y,batch_size=100, epochs=200, verbose=2)
# save the model to file
```

```
model.save('model.h5')

# compile reverse sequence network
rev_model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit network
rev_model.fit(rev_X, rev_y,batch_size=100, epochs=200, verbose=2)
# save the model to file
rev_model.save('rev_model.h5')
```

9.3.2.6.1 Sample Output

```
Epoch 192/200
 - 38s - loss: 0.2260 - acc: 0.9184
Epoch 193/200
 - 38s - loss: 0.1871 - acc: 0.9276
Epoch 194/200
 - 38s - loss: 0.1745 - acc: 0.9281
Epoch 195/200
 - 39s - loss: 0.1690 - acc: 0.9285
Epoch 196/200
 - 38s - loss: 0.1683 - acc: 0.9302
Epoch 197/200
 - 38s - loss: 0.1656 - acc: 0.9293
Epoch 198/200
 - 42s - loss: 0.1681 - acc: 0.9284
Epoch 199/200
 - 42s - loss: 0.1692 - acc: 0.9300
Epoch 200/200
 - 40s - loss: 0.1664 - acc: 0.9287
```

9.3.2.7 Sequence Generating Function

```
# generate a sequence using a language model
def generate_seq(model, tokenizer, max_length, seed_text):
    if seed_text == "":
        return ""
    else:
        in_text = seed_text
        n_words = 1
        n_preds = 5 #number of words to predict for the seed text
        pred_words = ""
        # generate a fixed number of words
        for _ in range(n_words):
            # encode the text as integer
            encoded = tokenizer.texts_to_sequences([in_text])[0]
            # pre-pad sequences to a fixed length
            encoded = pad_sequences([encoded], maxlen=max_length, padding='pre')
            # predict probabilities for each word
            proba = model.predict(encoded, verbose=0).flatten()
            #take the n_preds highest probability classes
            yhat = numpy.argsort(-proba)[:n_preds]
            # map predicted words index to word
            out_word = ''

            for _ in range(n_preds):
                for word, index in tokenizer.word_index.items():
```

```
                    if index == yhat[_] and word not in stopwords:
                        out_word = word
                        pred_words += ' ' + out_word
                        #print(out_word)
                        break


        return pred_words
```

9.3.2.8 *Loading Models and required parameters*

```
# load the model
model = load_model('model.h5')
rev_model = load_model('rev_model.h5')

#load tokeniser and max_length
with open('tokenizer.pkl', 'rb') as f:
    tokenizer = pickle.load(f)

with open('max_length.pkl', 'rb') as f:
    max_length = pickle.load(f)

#loading stopwords to improve relevant word predictions
stopwords= open('stopwords').read().split()

#load spacy GloVe Model
nlp = spacy.load('en_core_web_md')
```

9.3.2.9 *Function for assigning word embeddings for OOV words*

```
#Find and set embeddings for OOV words
def set_embedding_for_oov(doc):
    #checking for oov words and adding embedding
    for token in doc:
        if token.is_oov == True:
            before_text = doc[:token.i].text
            after_text = str(array(doc)[:token.i:-1]).replace('[','').replace(']','')

            pred_before = generate_seq(model, tokenizer, max_length-
1, before_text).split()
            pred_after = generate_seq(rev_model, tokenizer, max_length-
1, after_text).split()

            embedding = numpy.zeros((300,))
            i=len(before_text)
            print('Words predicted from forward sequence model:')
            for word in pred_before:
                print(word)
                embedding += i*nlp.vocab.get_vector(word)
                i= i*.5
            i=len(after_text)
            print('Words predicted from reverse sequence model:')
            for word in pred_after:
                print(word)
                embedding += i*nlp.vocab.get_vector(word)
                i= i*.5
            nlp.vocab.set_vector(token.text, embedding)
            #print(token.text,nlp.vocab.get_vector(token.text))
```

9.3.3 FastText Model built for comparison

```
import gensim
from gensim.models import FastText
# import corpus and
from gensim.models.word2vec import LineSentence
data = LineSentence('Output.txt')
#model = ft.load_fasttext_format("crawl-300d-2M.vec")
model = FastText(data, size=300, window=3, min_count=1, iter=10)
from gensim.test.utils import get_tmpfile

fname = get_tmpfile("fasttext.model")
model.save(fname)
model = FastText.load(fname)  # loading model
```