NLP

Data Cleaning

The raw text data comes directly after the various sources are not cleaned. We apply multiple steps to make data clean.

Tokenizing

Lower Casing

Punctuation Removal

Stop Words Removal

Spelling Correction

Stemming

We use Stemming to normalize words. In English and many other languages, a single word can take multiple forms depending upon context used. For instance, the verb “study” can take many forms like “studies,” “studying,” “studied,” and others, depending on its context. When we tokenize words, an interpreter considers these input words as different words even though their underlying meaning is the same. Moreover, as we know that NLP is about analyzing the meaning of content, to resolve this problem, we use stemming.

Stemming normalizes the word by truncating the suffix from the word and returning the stem of the word. For example, the words “studies,” “studied,” “studying” will be reduced to “studi,” making all these word forms to refer to only one token. Notice that stemming may not give us a dictionary, grammatical word for a particular set of words.

Lemmatization

Lemmatization tries to achieve a similar base “stem” for a word. However, what makes it different is that it finds the dictionary word instead of truncating the original word. Lemmatization makes use of the vocabulary to get the root form of the word. Stemming does not consider the context of the word. That is why it generates results faster, but it is

less accurate than lemmatization. For exam, during lemmatization, the word “studies” displays its dictionary word “study.”

If accuracy is not the project’s final goal, then stemming is an appropriate approach. If higher accuracy is crucial and the project is not on a tight deadline, then the best option is Lemmatization. Lemmatization takes into account Part Of Speech (POS) values. Also, lemmatization may generate different outputs for different values of POS.

Part of Speech Tagging

Every word in a sentence is also associated with a part of speech (pos) tag (nouns, verbs, adjectives, adverbs etc). The pos tags defines the usage and function of a word in the sentence

Chunking

Chunking means to extract meaningful phrases from unstructured text. By tokenizing a book into words, it’s sometimes hard to infer meaningful information. It works on top of Part of Speech(PoS) tagging. Chunking takes PoS tags as input and provides chunks as output. Chunking literally means a group of words, which breaks simple text into phrases that are more meaningful than individual words.

Vectorization

The process of converting text into vector is called vectorization.

Count Vector

Count Vector Vectorization produces bag of word model. Why it is called bag of words because any order of the words in the document only tells us weather word is present in the document or not.

Let's say we have 2 documents as below:

"Dog hates a cat. It loves to go out and play."

"Cat loves to play with a ball."

We can build a corpus from above 2 documents just by combining it.

Corpus = "Dog hates a cat. It loves to go out and play. Cat loves to play with a ball."

And features will be all unique words = ['and', 'ball', 'cat', 'dog', 'go', 'hates', 'it', 'loves', 'out', 'play', 'to', 'with']

Bag of Words takes a document from a corpus and converts it into a numeric vector by mapping each document word to a feature vector.

Features / Words	and	ball	Cat	Dog	go	hates	it	loves	out	play	to	with
Cat	0	0	**1**	0	0	0	0	0	0	0	0	0
Loves	0	0	0	0	0	0	0	**1**	0	0	0	0
To	0	0	0	0	0	0	0	0	0	0	**1**	0
Play	0	0	0	0	0	0	0	0	0	**1**	0	0
With	0	0	0	0	0	0	0	0	0	0	0	**1**
A	0	0	0	0	0	0	0	0	0	0	0	0
Ball	0	**1**	0	0	0	0	0	0	0	0	0	0
BoW for Document 2	0	1	1	0	0	0	0	1	0	1	1	1

Here, each word can be represented as an array of the length of total numbers of features (words). All the values of this array will be zero apart from one position. That position representing words address inside the feature vector. The final BoW representation is the sum of words feature vector.

By using the CountVectorizer function we can convert a text document to matrix (sparse matrix) of word count. Countvectorizer produces sparse matrix which sometime not suited for some machine learning model hence first convert this sparse matrix to dense matrix then apply machine learning model

TF IDF

The BoW method is simple and works well but it treats all words equally and cannot distinguish very common words or rare words. Tf-idf solves this problem of BoW Vectorization.

tf-idf(t, d) = tf(t, d) * log(N/(df + 1))

The TF-IDF scheme is a type of bag words approach where instead of adding zeros and ones in the embedding vector, you add floating numbers that contain more useful information compared to zeros and ones.

Term frequency-inverse document frequency (tf-idf) gives a measure to know the importance of a word in consideration of how frequently it occurs in a document and a corpus. The idea behind TF-IDF scheme is the fact that words having a high frequency of occurrence in one document and less frequency of occurrence in all the other documents are more crucial for classification.

Term frequency gives a measure of the frequency of a word in a document. Term frequency for a word is a ratio of no. of times a word appears in a document to total no. of words in the document.

tf ('word') = No. times of 'word' appears in document / total number of words in a document

Inverse document frequency is a measure of the importance of the word. It measures how common a particular word is across all the documents in the corpus. It is the logarithmic ratio of no. of total documents to no. of a document with a particular word.

idf(t) = No. of total documents / No. of a document with 'word' in it

Now there are few other problems with the IDF , in case of a large corpus,say 100,000,000 , the IDF value explodes , to avoid the effect we take the log of idf .

During the query time, when a word which is not in vocab occurs, the df will be 0. As we cannot divide by 0, we smoothen the value by adding 1 to the denominator and numerator.

that's the final formula:

idf('word') = log (No. of total documents +1 / (No. of a document with 'word' in it)+1)

Ex: "Cat loves to play with the ball"

"Cat is an animal and cat is dangerous"

Calculating tf idf for Cat for Document 1.

tf('Cat') = 1 / 6

idf('Cat') = log (2/2+1) =

tfidf= ⅙ * log(⅔) = ⅙ * 0.1 = 0.016

A combination of N words together are called N-Grams. N grams (N > 1) are generally more informative as compared to words (Unigrams) as features.

Unigrams are the single unique words in a sentence

Bigrams are the combination of 2 words ie “not bad”,” turn off”.

Trigrams are the combination of 3 words.

ngram_range =(1, 1) means only unigrams,

ngram_range = (1, 2) means unigrams with bigrams

ngram_range=(2, 2) means only bigrams

While implementing the BOW and TF-IDF models we can include n-grams in vocabulary

Word Embedding

Word embedding is a feature learning technique where words are mapped to vectors using their contextual hierarchy

The potential drawback with one-hot encoded feature vector approaches such as bag of words and TF-IDF approaches are

- the feature vector for each document can be huge. For instance, if you have a half million unique words in your corpus and you want to represent a sentence that contains 10 words, your feature vector will be a half million dimensional one-hot encoded vector where only 10 indexes will have 1. This is a wastage of space and increases algorithm complexity exponentially resulting in the curse of dimensionality.

In Word embedding approach, the size of the embedding vector is very small. Each dimension in the embedding vector contains information about one aspect of the word. We do not need huge sparse vectors unlike the bag of words and TF-IDF approaches.

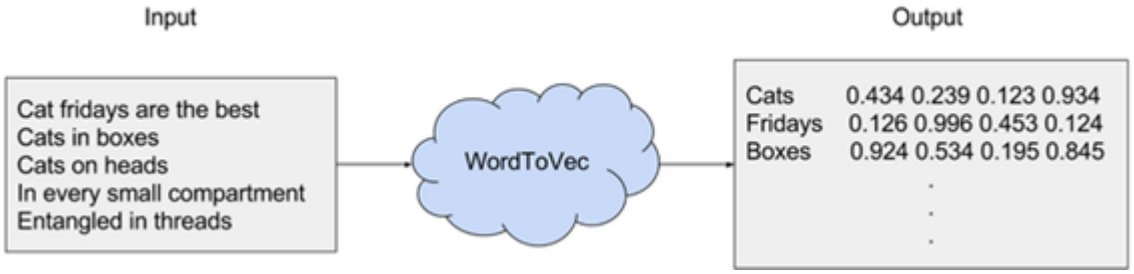
- they ignore the context of words. “I like you” and “I love you” will have completely different feature vectors according to TF-IDF and BOW model, but that’s not correct.

Word2Vec and GloVe are the two popular models to create word embedding of a text. These models takes a text corpus as input and produces the word vectors as output

Word2Vec

Word2Vec is a technique to produce word embedding for better word representation.

Word2Vec is an algorithm that accepts text corpus as an input and outputs a vector representation for each word.



The purpose and usefulness of Word2vec is to group the vectors of similar words together in vector space or we can say a well-trained set of word vectors will place similar words close to each other in n-dimensional space where n refers to the dimensions of the vector. The placement is done in such a way that similar meaning words appear together and dissimilar words are located far away. This is also termed as a semantic relationship.

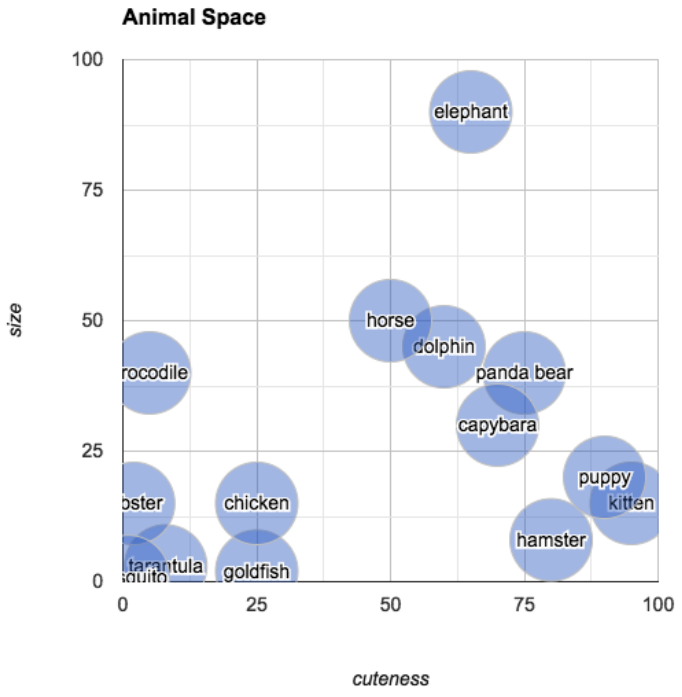
Word2vec is one of the most popular technique to learn word embeddings using a two-layer neural network. Word2vec is a two-layer neural net where there is input one hidden layer and output that processes text by vectorizing words. Its input is a text corpus and its output is feature vectors that represent words in that corpus. The vectors we use to represent words are called neural word embeddings.

While Word2vec is not a deep neural network, it turns text into a numerical form that deep neural networks can understand. The output of the Word2vec neural net is a vocabulary in which each item has a vector attached to it which can be fed into a deep-learning net or simply queried to detect relationships between words.

The idea behind Word2Vec is pretty simple. We're making an assumption that the meaning of a word can be inferred by the company it keeps. This is analogous to the saying, "show me your friends, and I'll tell who you are." The essential thought of word embedding is that words that occur in a comparative context will be nearer to one another in the vector space.

We'll begin by considering a small subset of english words for animals. Our task is to be able to write computer programs to find similarities among these words and the creatures they designate.

	cuteness (0–100)	size (0–100)
kitten	95	15
hamster	80	8
tarantula	8	3
puppy	90	20
crocodile	5	40
dolphin	60	45
panda bear	75	40
lobster	2	15
capybara	70	30
elephant	65	90
mosquito	1	1
goldfish	25	2
horse	50	50
chicken	25	15



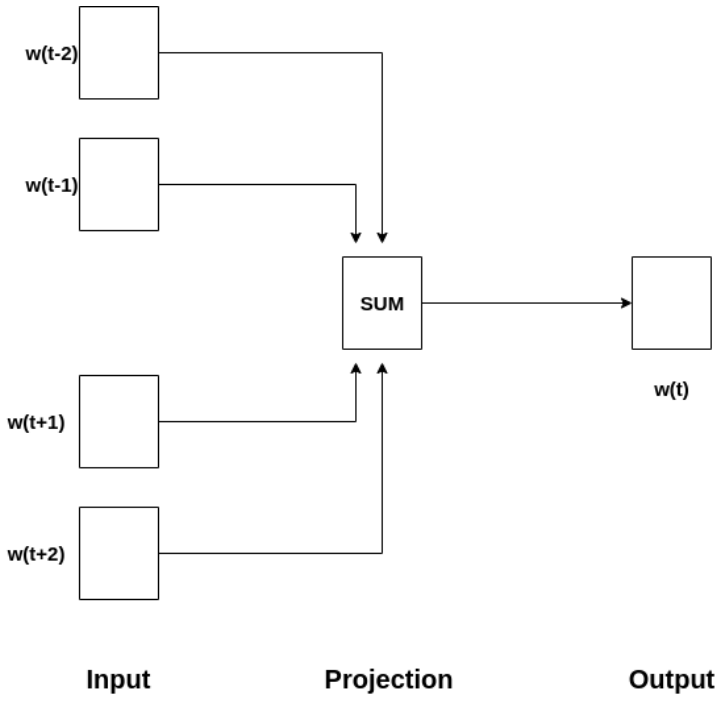
Word2vec is similar to an autoencoder, encoding each word in a vector but rather than training against the input words through reconstruction or we can say word2vec trains words against other words that neighbour them in the input corpus. It does so in one of two ways, either using context to predict a target word (CBOW) or using a word to predict a target context (Skip-gram). When the feature vector assigned to a word cannot be used to accurately predict that word's context, the components of the vector are adjusted. Each

word's context in the corpus is the teacher sending error signals back to adjust the feature vector. The vectors of words judged similar by their context are nudged closer together by adjusting the numbers in the vector.

It works on the idea of Distributional semantics which means that we can understand the meaning of a word by understanding the company (context) that a word keeps. So, in order to learn the representation of a word, we will try to learn the context in which a particular word appears and will hope that this word's vector will be similar to the vector of the context words.

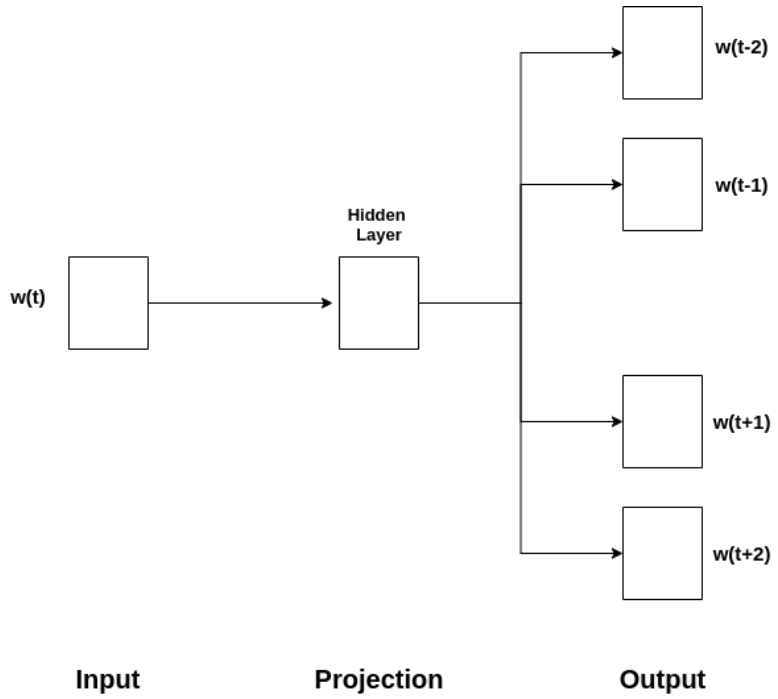
Learning word representation is essentially unsupervised but labels are needed to train the model. There are two architectures of Word2vec - Continuous bag of words (CBOW) model and Skip gram model which convert unsupervised representation to supervised form for model training. Given a corpus (set of sentences), the model loops on the words of each sentence and either try to use the current word w in order to predict its neighbors/ context or it employs each of these contexts to predict the current word w.

In CBOW, the model uses the context to make a prediction about the current word.



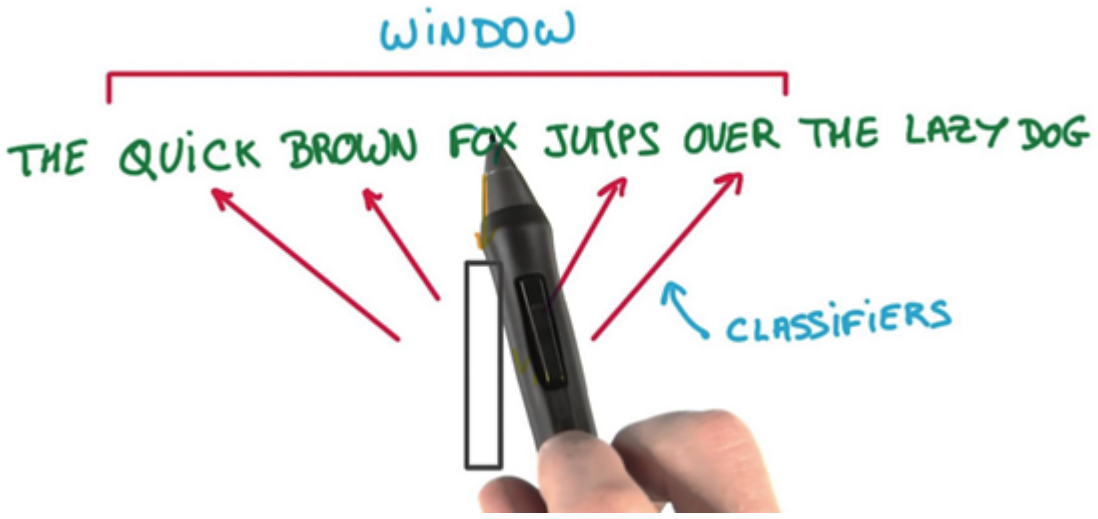
In CBOW, the current word is predicted using the window of surrounding context windows. For example, if wi-1, wi-2, wi+1, wi+2 are given words or context, this model will provide wi.

In Skip-Gram, the model iterates over the words in the corpus and predicts the neighbors/ the context



Skip-Gram performs opposite of CBOW which implies that it predicts the given sequence or context from the word. You can reverse the example to understand it. If wi is given, this will predict the context or wi-1, wi-2, wi+1, wi+2. One can treat it as the reverse of the CBOW model where the input is the word and model provides the context or the sequence. We can also conclude that the target is fed to the input and output layer is replicated multiple times to accommodate the chosen number of context words.

Word2vec provides an option to choose between CBOW (continuous Bag of words) and skim-gram. To limit the number of words in each context and tune the performance of the model, a parameter called window size is used. The recommended value is 10 for skip-gram and 5 for CBOW.



Layers in Word2Vec Neural Network

- Input layer

First of all, we cannot feed a word as string into a neural network. Instead, we feed words as one-hot vectors, which is basically a vector of the same length as the vocabulary, filled with zeros except at the index that represents the word we want to represent which is assigned "1".

- Hidden layer

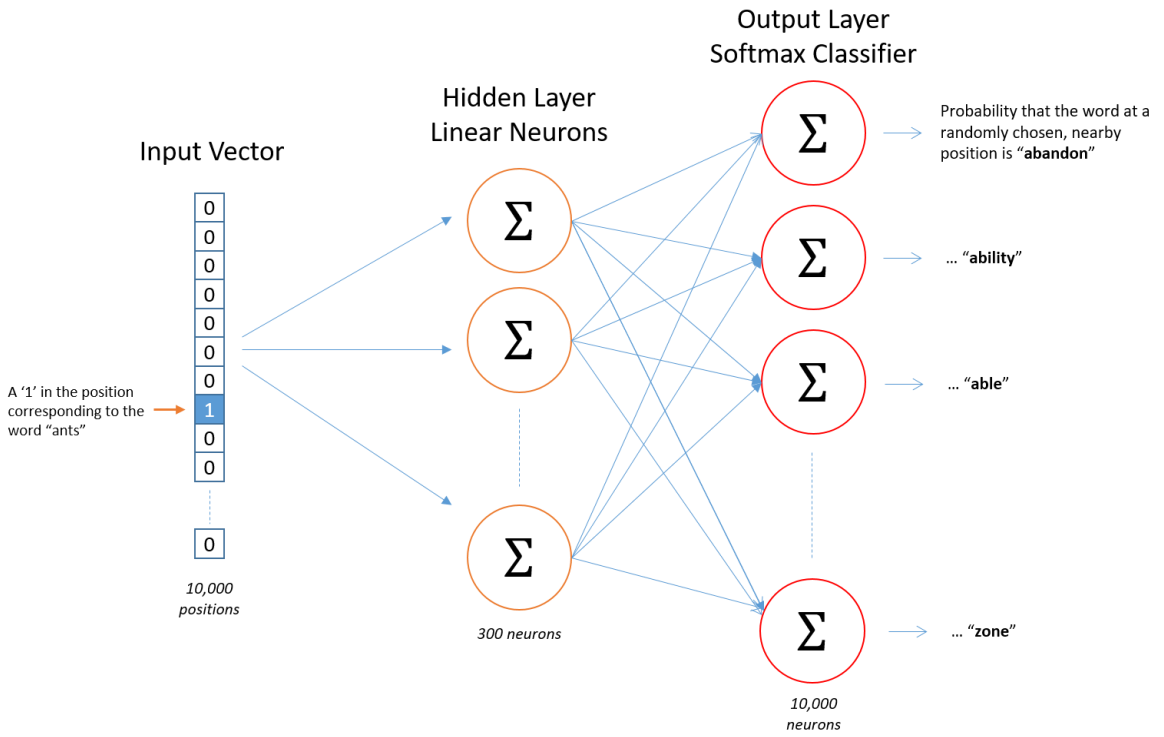
The hidden layer is a standard fully-connected (Dense) layer which takes one hot vector as input and gives word vector as output. Number of neurons used in hidden layer for training is a hyper parameter and is defined by the user. There is no activation function on the hidden layer neurons.

Let's understand it through an example.

We're going to represent an input word like "ants" as a one-hot vector. This vector will have 10,000 components (one for every word in our vocabulary) and we'll place a "1" in the position corresponding to the word "ants" and 0s in all of the other positions.

The output of the network is a single vector (also with 10,000 components) containing for every word in our vocabulary, the probability that a randomly selected nearby word is that vocabulary word

And we're going to say that we're learning word vectors with 300 features. So the hidden layer is going to be represented by a weight matrix with 10,000 rows (one for every word in our vocabulary) and 300 columns (one for every hidden neuron). 300 features is what Google used in their published model trained on the Google news dataset.



$$[0 \quad 0 \quad 0 \quad 1 \quad 0] \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = [10 \quad 12 \quad 19]$$

In the following figure, first matrix is a one-hot encoding vector for a given word, second matrix is feature matrix, also called weight matrix in which i-th line is the vector representation of the i-th word and the last matrix is the word vector for the input word.

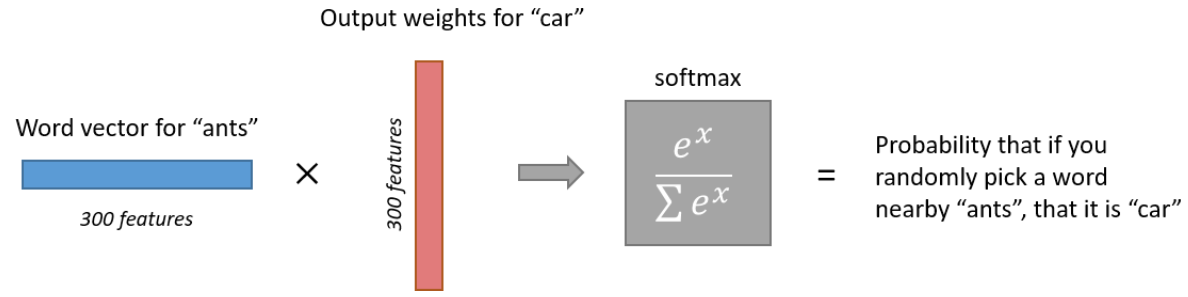
Now, if you multiply a 1 x 10,000 one-hot vector by a 10,000 x 300 matrix, it will effectively give 1*300 matrix. The output of the hidden layer (1 x 300 vector) is just the word vector for the input word. The end goal of hidden layer is to learn this hidden layer weight matrix.

- Output layer

The output layer outputs probabilities for the target words from the vocabulary through softmax function. The target words here would be all the words except the input word.

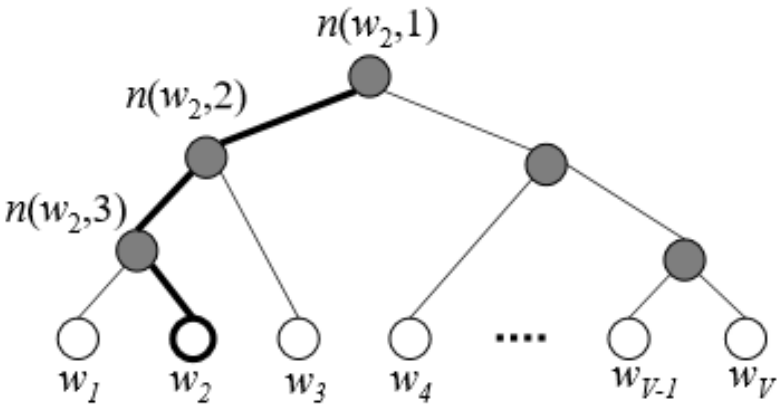
The output layer has predefined number of output neurons with each having a weight vector which it multiplies against the word vector resulted from the hidden layer and then it

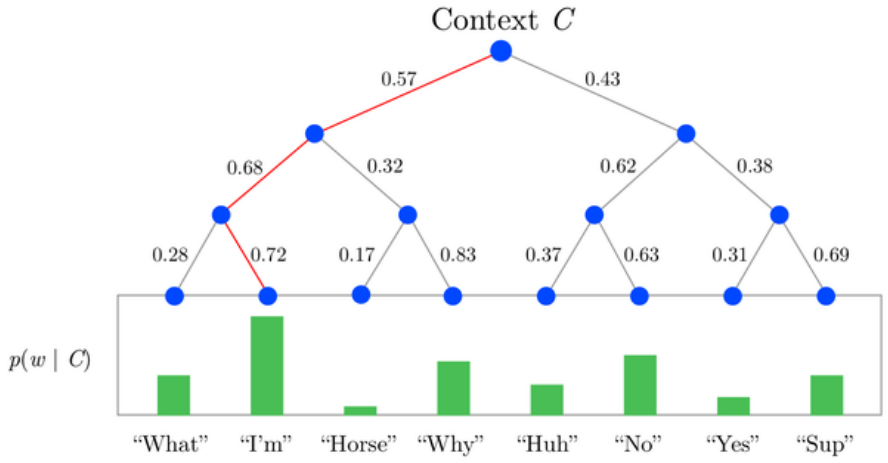
applies the function exp(x) to the result producing an output between 0 and 1. The sum of output values from all the neurons will add up to 1. Finally, in order to get the outputs to sum up to 1, we divide this result by the sum of the results from all output neurons.



Regular Softmax function converts normalized embedding to probabilities. The training speed for models with softmax output layers quickly decreases as the vocabulary size grows or we can say when the number of outputs at the output layer increases, computing the softmax becomes very expensive. There are different strategies that can be used to make the computation more efficient or to approximate the softmax. These approaches can be grouped into softmax-based and sampling-based approaches. Softmax-based approaches are methods that keep the softmax layer intact, but modify its architecture to improve its efficiency. Sampling-based approaches on the other hand completely do away with the softmax layer and instead optimise some other loss function that approximates the softmax.

Hierarchical Softmax is a is an alternative to softmax that is faster to evaluate. it is O(logn) time to evaluate compared to O(n) for softmax. Hierarchical Softmax strategy has been inspired by multi-layer binary trees. Under this strategy, a hierarchical tree is constructed to index all the words in vocabulary and the probability of a word is calculated through the product of probabilities on each edge on the path to that node i.e. output layer is represented as a binary tree in hierarchical softmax with the V words in vocabulary as its leaf nodes and the intermediate nodes as internal parameters. Each intermediate node explicitly represents the relative probabilities of its child nodes. The idea behind decomposing the output layer to a binary tree was to reduce the complexity to obtain probability distribution from O(V) to O(log (V)). Since it is a binary tree it is obvious that there are V-1 intermediate nodes. For each leaf node there exists a distinct path from root node and this path is used to estimate the probability of the word represented by the leaf node. Replacing a softmax layer with H-Softmax can yield speedups for word prediction tasks of at least 50 times.





Negative sampling is another strategy to approximate the softmax. In Word2Vec model training, assuming stochastic gradient descent, weight matrices in the neural network are updated for each training sample to correctly predict output. Let's assume that the training corpus has 10,000 unique vocabs and the hidden layer is 300-dimensional (the dimension of the hidden layer h is also called the size of a word vector). This means that there are 3,000,000 neurons in the output weight matrix that need to be updated for each training sample. Since the size of the training corpus is very large, updating 3M neurons for each training sample is unrealistic in terms of computational efficiency. Negative sampling addresses this issue by updating only a small fraction of the output weight neurons for each training sample. In negative sampling, K negative samples are randomly drawn from a noise distribution(K is a hyper-parameter that can be empirically tuned with a typical range of [5, 20]. K = 2 - 5 works well for large data sets and K= 5-20 works well for small data sets.). For each training sample, you randomly draw K number of negative samples from a noise distribution and the model will update K+1 times N neurons in the output weight matrix. For example, if you set K=9, the model will update 9+1 times 300 = 3000 neurons, which is only 0.1% of the 3M neurons in weight matrix. This is computationally much cheaper than the

original computations and yet maintains a good quality of word vectors. This helps to speed up the prediction time of the model.

When training the Word2Vec network on word pairs, the input is a one-hot vector representing the input word and the training output is also a one-hot vector representing the output word. But when you evaluate the trained network on an input word, the output vector will actually be a probability distribution (a bunch of floating point values, not a one-hot vector).

Word2Vec is trained on more than 6 billion words using shallow neural networks. A pre-trained word vector is a text file containing billions of words with their vectors. we only need to map words from our data with the words in the word vector in order to get the vectors. You can download the word vector file. Pre-trained word vector file come in (50,100,200,300) dimension. Here, dimension is the length of the vector of each word in vector space. More dimension means more information about that word but bigger dimensions takes longer time for model training.