Word embeddings: Helping computers understand language semantics

By Pushpam Punjabi

Humans have a very intuitive way to work with languages. Tasks such as understanding similar texts, translating a text, completing a text, and summarizing a text come very naturally to humans with the inherent understanding of the language semantics. But when it comes to computers, passing on this intuition is an uphill task! Sure, computers can assess how structurally similar two strings are. When you type “backstreat boys,” a computer might correct you to “backstreet boys,” but how do you make them understand the semantics of words?

· How do you make a computer infer that king and queen carry the same equivalence as man and woman?

· How do you make a computer infer that in a conversation about technology companies, the term Apple refers to the company and not the fruit?

· How do you make a computer infer that if someone is searching for football legends and has searched Ronaldo, then they might (should!) also be interested in Messi?

· How do you make a computer recommend “GoodFellas” or “The Irishman” when someone has browsed for “The Godfather”?

How do you accomplish this mammoth task of bridging the gap between humans and computers to infer the capacity of interpreting languages? The answer to these questions lies in the concept of “word embeddings”! Read on!

What are word embeddings?

Word embeddings are a way of representing words in a numerical format. In simple terms, it is a mathematical way of representing the meaning of words. Word embeddings represent words as vectors of real-valued numbers, where each dimension in the vector corresponds to a particular feature or aspect of the word's meaning. For example, one dimension might represent the word's gender, while another might represent its tense. The values in the vector indicate the strength of the association between the word and the feature. In a way, word embeddings are like a dictionary for a computer. Just like we use a dictionary to look up the meanings of words, a computer can use a word embedding to look up the numerical vector representation of a word. Let's look how these word embeddings are calculated!

Understanding word2vec

word2vec is an abbreviation for "word to vector" and is one of the widely used approaches to learn word embeddings. While there are many implementations of this approach, I'll present a simplified explanation of the core concept.

The intuition behind the approach

Let's think of how humans approach understanding a new word. Our natural approach is to make some sense of the word based on its context. E.g., say that you don't know the meaning of the word **mojo**. You don't know what it means or how to use it. You don't have access to any dictionary! But you see everybody around you using this word in various conversations such as:

· The team has lost its mojo!

· We need to get our mojos working again!

· Game of Thrones lost its mojo in the final season!

· It took me a long time to get my mojo back!

You also see other terms used in similar context, such as:

· The team has lost its power!

· We need to get our magic working again!

· Game of Thrones lost its charm in the final season!

· It took me a long time to get my energy back!

And then you connect the dots to make a mental map of mojo with charm or energy or magic!

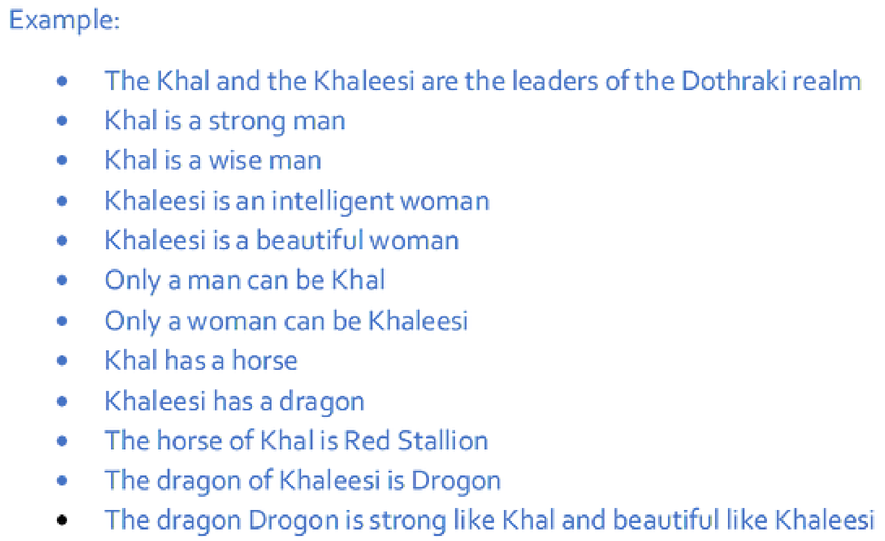
In the above example, we looked for words with similar meanings. You can imagine a similar exercise for understanding relationships (man-woman => king-queen), tenses (running => ran), and other aspects of a language.

The word2vec approach

word2vec works in a similar manner. It creates a mapping of the words and the context in which these words are used, and then uses a neural network to capture the word similarities in the form of a vector of numbers.

Let's understand word2vec with an example.

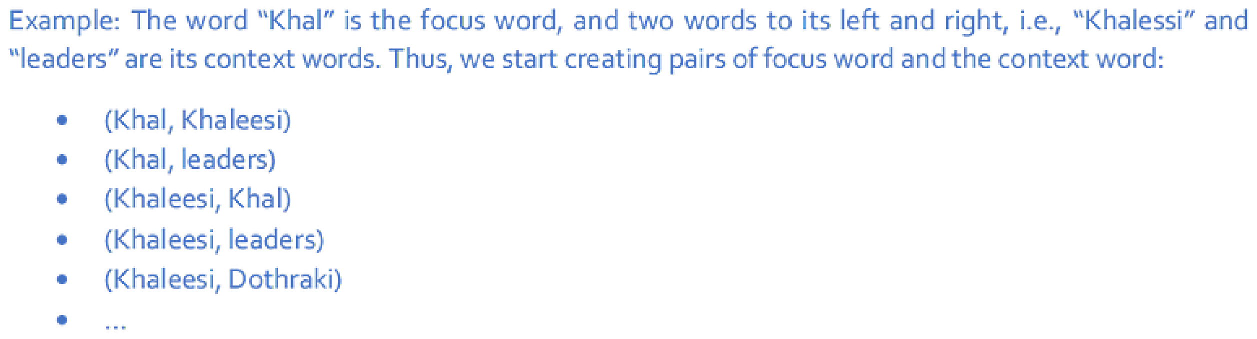
Step 1: The first thing we need is text. Consider the following sentences that state some facts of the fictional Dothraki royal family.



Example:

- The Khal and the Khaleesi are the leaders of the Dothraki realm
- Khal is a strong man
- Khal is a wise man
- Khaleesi is an intelligent woman
- Khaleesi is a beautiful woman
- Only a man can be Khal
- Only a woman can be Khaleesi
- Khal has a horse
- Khaleesi has a dragon
- The horse of Khal is Red Stallion
- The dragon of Khaleesi is Drogon
- The dragon Drogon is strong like Khal and beautiful like Khaleesi

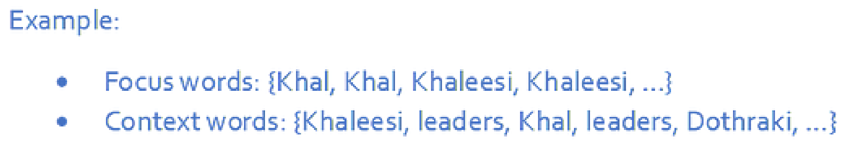
Step 2: For each word in these sentences, we identify some words before and after it and capture them as the context-words. The context window size is one of the hyperparameters that can be configured while creating the embeddings. For example, if we consider the context window as two words, then for each word, we look for two words before and after it to form its context words. At this step, some preprocessing is also done to exclude stop words such as: the, is, are, of, etc. Applying this to our example:



Example: The word "Khal" is the focus word, and two words to its left and right, i.e., "Khalessi" and "leaders" are its context words. Thus, we start creating pairs of focus word and the context word:

- (Khal, Khaleesi)
- (Khal, leaders)
- (Khaleesi, Khal)
- (Khaleesi, leaders)
- (Khaleesi, Dothraki)
- ...

Step 3: At this stage, we have a vector of focus words and another vector of context words. Consider that we have **n** focus words. Thus both focus and context words vectors are of length **n**.

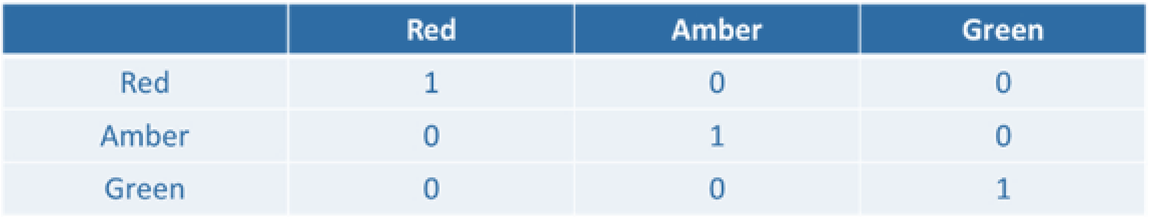


Example:

- Focus words: {Khal, Khal, Khaleesi, Khaleesi, ...}
- Context words: {Khaleesi, leaders, Khal, leaders, Dothraki, ...}

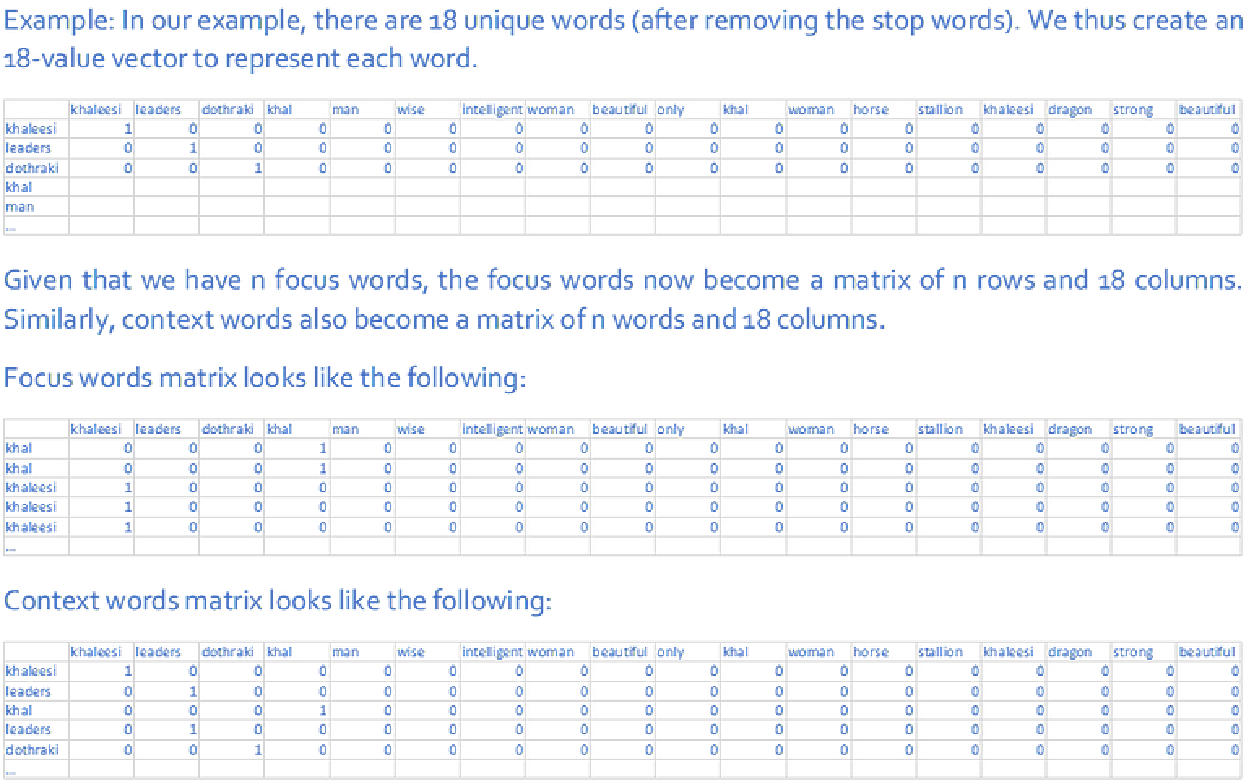
Step 4: Next, we want to use a neural network to map the focus words to the context words. But neural networks do not understand strings. Hence, we need to convert these strings into vectors of numbers. Let's take a short detour to understand a simple way to convert strings to numbers using **one hot encoding**.

One hot encodingconverts a word into a binary vector of zeros and one. Let's try this out with a simple example. Consider a language dictionary that has only three words: Red, Amber, and Green. Each word is represented as a vector of three numbers.



	Red	Amber	Green
Red	1	0	0
Amber	0	1	0
Green	0	0	1

The basic idea is to represent each word as a vector of zeros and ones, where the length of the vector is equal to the size of the vocabulary, i.e., the number of unique words in the text data.



Example: In our example, there are 18 unique words (after removing the stop words). We thus create an 18-value vector to represent each word.

	khaleesi	leaders	dothraki	khal	man	wise	intelligent	woman	beautiful	only	khal	woman	horse	stallion	khaleesi	dragon	strong	beautiful
khaleesi	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
leaders	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
dothraki	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
khal																		
man																		
...																		

Given that we have n focus words, the focus words now become a matrix of n rows and 18 columns. Similarly, context words also become a matrix of n words and 18 columns.

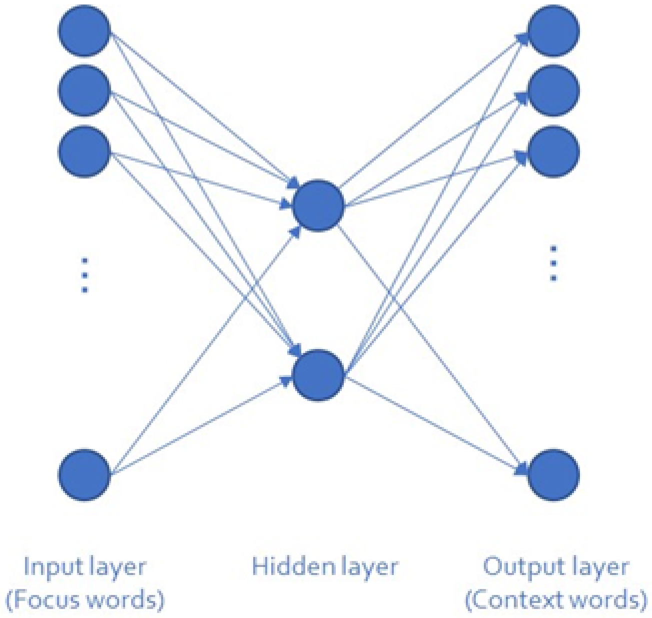
Focus words matrix looks like the following:

	khaleesi	leaders	dothraki	khal	man	wise	intelligent	woman	beautiful	only	khal	woman	horse	stallion	khaleesi	dragon	strong	beautiful
khal	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
khal	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
khaleesi	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
khaleesi	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
khaleesi	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
...																		

Context words matrix looks like the following:

	khaleesi	leaders	dothraki	khal	man	wise	intelligent	woman	beautiful	only	khal	woman	horse	stallion	khaleesi	dragon	strong	beautiful
khaleesi	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
leaders	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
khal	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
leaders	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
dothraki	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
...																		

Step 5: Now, we train a neural network. This neural network has one input layer, one output layer, and one hidden layer. The input layer consists of the focus words and the output layer consists of the context words. The intermediate hidden layer is the one responsible for creating embeddings! The number of nodes in the hidden layer is configurable. For ease of visualization, I set it to 2. If I have two nodes in the hidden layer, then each input node will have two weights connecting to the hidden layer. The weights on these edges are the word embeddings.

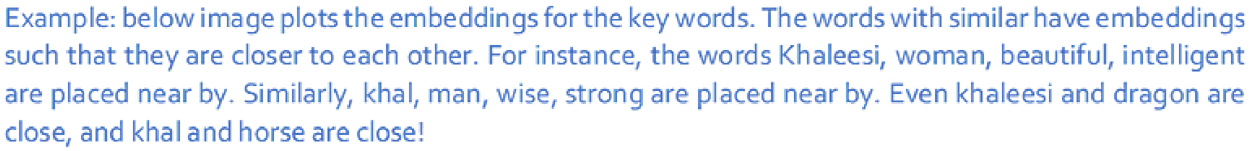


A simple model with one context word for one focus work and 2 nodes in the hidden layer

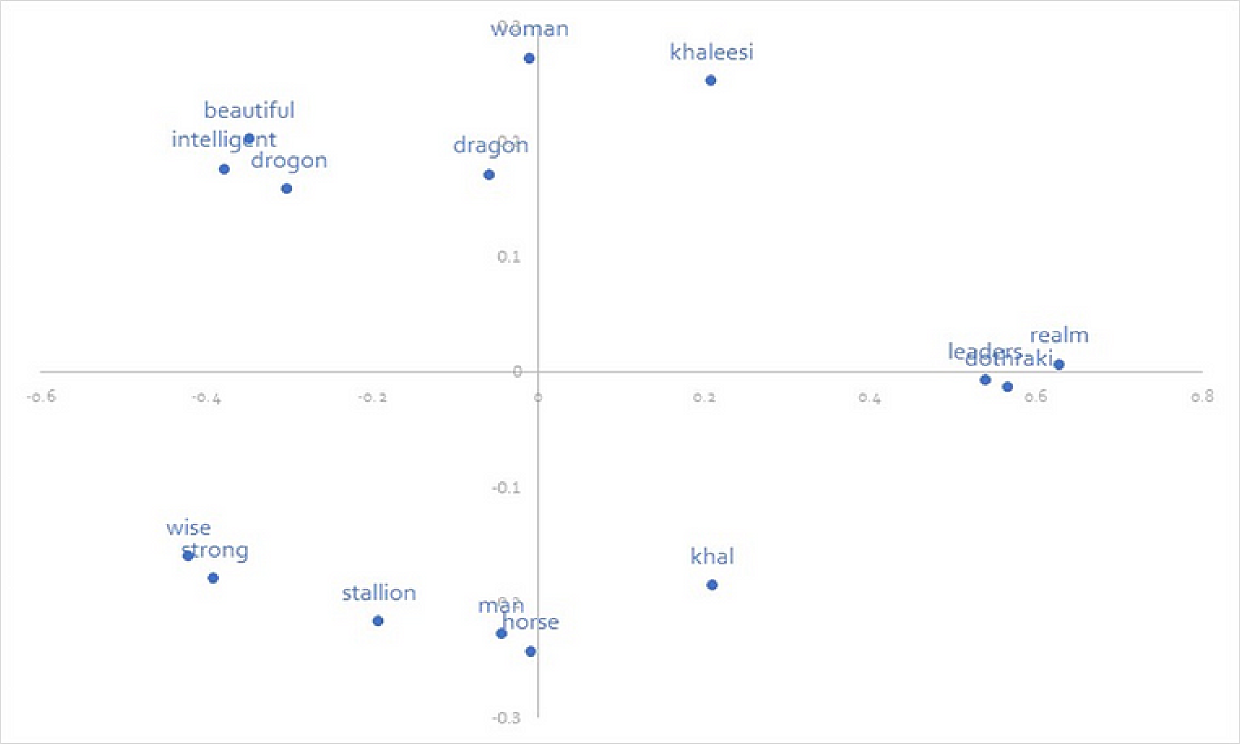
Without going into details about the neural network, consider that the task of the neural network is to take n 18-bit focus vectors as input, n 18-bit context vectors as output, and create an intermediate vector that best matches input to output! Since we have set the dimension size of the hidden layer as two, we create an 18*2 matrix in the hidden layer. In other words, for each word we form an embedding vector of size two.

The beauty of these embedding vectors is that they try to capture the context around the words such that words with similar context have smaller distance between them. This also reflects in a scatter plot where the words with similar context get placed closer to one another!

Step 6: We extract the 18*2 matrix that represents the embeddings for the 18 words. We plot this matrix to understand how the words affinity is captured by the neural network.



Example: below image plots the embeddings for the key words. The words with similar have embeddings such that they are closer to each other. For instance, the words Khaleesi, woman, beautiful, intelligent are placed near by. Similarly, khal, man, wise, strong are placed near by. Even khaleesi and dragon are close, and khal and horse are close!



To summarize, the approach includes the following steps:

· Read the text.

· Pre-process the text.

· Create a mapping of focus and context vectors.

· Create their one hot encodings.

· Train the neural network.

· Extract the weight from the hidden layer and use these weights as the word embedding vectors.

I have presented a simplified approach to explain word embeddings. The word2vec algorithms has some more nuances. There are two popular techniques — CBOW (Continuous Bag of Words) and Skip-gram. CBOW predicts the target word from the context words. Skip-gram takes an exact opposite approach and predicts the context words given the target word.

Harnessing the power of word embeddings

Word embeddings provide the power of understanding contextual similarities in words. This can be used in a variety of ways. Below are some examples:

· **Search Engines:** Word embeddings can improve the matches in a search engine. E.g., if you search for “soccer,” the search engine also gives you results for “football” as they’re two different names for the same game.

· **Language Translation:** Word embeddings are crucial for language translation. Two or more words with the same meaning words in two different languages would have similar vectors, which would make it easier for a computer to translate from one language to another. E.g., “engineer” in English is translated to “ingeniero” or “ingeniera” in Spanish. Word embeddings for all the three words would be similar, and hence, the machine would be able to translate text with better accuracy.

· **Chatbots:** We’ve seen increasing use of chatbots across different applications for different purposes. The users of a chatbot can write a query in any form and can use different words to convey the same thing. E.g., for a taxi booking application’s chatbot, a user can either say “book me a cab” or “please reserve a taxi for me.” Both sentences convey the same thing. Using word embeddings, a chatbot can understand that they’re the same and act on it accordingly.

ignio’s take

ignio leverages word embeddings for analyzing different types of data sources to create the context of an enterprise. Example use cases include: analysis of trouble tickets data to group tickets referring to similar incidents, mapping trouble tickets to ignio’s automation catalog to identify tickets that can be auto-resolved by ignio, or extracting entities of interest from tickets data such as fault type, entity name, entity type, etc.

Conclusion

This blog discussed the concept of word embeddings, which helps machine to understand semantics of texts. Word embeddings are a great analogy to how humans understand language — humans first understand the meaning of the words in any text, and then try to understand what the text entails. Similarly, a computer understands the meaning of words by using word embeddings. This blog also explored how to generate word embeddings using one of the most popular techniques — word2vec. Word embeddings are the steppingstone for the current advancements in Natural Language Processing (NLP) like BERT and ChatGPT.