Data Processing for AI – Unstructured Data

Converting text to features

In this section, we are going to cover basic to advanced feature engineering methods. We are going to discuss the following recipes

Tutorial & Concept1. One hot encoding

Tutorial & Concept2. Count vectorizer

Tutorial & Concept3. N-grams

Tutorial & Concept4. Term Frequency-Inverse Document Frequency (TF-IDF)

Tutorial & Concept5. Word Embedding

Now that all the text pre-processing steps are discussed, let's explore feature engineering, the foundation for Natural Language Processing. As we already know, machines or algorithms cannot understand the characters/words or sentences, they can only take numbers as input which also include binary. But the inherent nature of text data is unstructured and noisy which makes impossible to interact with machines.

The procedure of converting raw text data into machine understandable format(numbers)is called feature engineering of text data. machine learning and deep algorithms performance and accuracy is fundamentally dependent on the type of feature engineering technique used.

In this chapter, we will discuss different type of feature engineering methods along with some state of art techniques, their functionality, advantages, disadvantages and examples for each and of all make you realize the importance of feature engineering.

Tutorial & Concept 3-1.

The traditional method used for feature engineering is One hot encoding. If anyone knows basics of machine learning, one hot encoding is something they should have come across for sure at some point of time or may be most of the time. It's a process of converting categorical variables into feature or columns and coding one or zero for the presence of that particular category. We are going to use the same logic here and number of features going to be the number of total tokens present in the whole corpus.

Problem

How to convert text to feature using one hot encoding?

Data Processing for AI – Unstructured Data

Solution

One hot encoding will basically convert characters or words into binary numbers as shown below.

	I	love	NLP	is	future
I love NLP	1	1	1	0	0
NLP is future	0	0	1	1	1

How It Works

There are so many functions to generate One hot encoding. We will take one function and discuss in depth.

Step 1-1 Store the text to a variable

```
Text = "I am learning NLP"
```

Step 1-2 Execute below function on the text data

Below is the function from pandas library to convert text to feature.

```
# Importing the library
import pandas as pd

# Generating the features
pd.get_dummies(Text)
```

Result :

```
   I  NLP  am  learning
0  1    0   0         0
1  0    0   1         0
2  0    0   0         1
3  0    1   0         0
```

Output has 4 feature since the distinct words present in the input was 4.

Tutorial & Concept 3-2.

The above approach has a disadvantage. It doesn't take frequency of the word occurring into consideration. If a particular word is appearing multiple time, there is chance of missing the information if its not included in analysis. Count vectorizer will solve that problem.

In this recipe, we will see the other method of converting text to feature which is count vectorizer.

Program: M.Tech in Artificial Intelligence
Batch: 01
Module: TRIM I Module IV
Credits: 04 (4 Saturdays)
Dates: 23rd, 30th Jan, 6th, 13th Feb 2021

Data Processing for AI – Unstructured Data

Problem

How to convert text to feature using count vectorizer?

Solution

Count vectorizer is almost similar to one hot encoding. The only difference is instead of checking whether the particular word is present or not, it will count the words which are present in the document.

Observe the below example. The word "I" and "NLP" occurring twice in first document.

	I	love	NLP	is	future	will	learn	in	2month
I love NLP and I will learn NLP in 2month	2	1	2	0	0	1	1	1	1
NLP is future	0	0	1	1	1	0	0	0	0

How It Works

Sklearn has a feature extraction function extract features out of text. Let's discuss how to execute the same. Import the "CountVectorizer" function from Sklearn as explained below.

```
#importing the function
from sklearn.feature_extraction.text import CountVectorizer

# Text
text = ["I love NLP and I will learn NLP in 2month "]

# create the transform
vectorizer = CountVectorizer()

# tokenizing
vectorizer.fit(text)


# encode document
vector = vectorizer.transform(text)

# summarize & generating output
print(vectorizer.vocabulary_)
print(vector.toarray())
```

Result:

```
{'love': 4, 'nlp': 5, 'and': 1, 'will': 6, 'learn': 3, 'in': 2, '2month': 0}
[[1 1 1 1 1 2 1]]
```

The fifth token "nlp" has appeared twice in the document.

Tutorial & Concept 3-3.

If you observe above methods, each word is considered as a feature. There is a drawback in this method.

It doesn't consider the previous and the next word, to see if that would give a proper and complete meaning to the words.

For example: consider the word "not bad". If this is split into individual words then it will lose out on conveying "good" what this word actually means.

As we saw, we might lose potential information or insight. Because lot of words makes sense once they are put together. This problem can be solved by N grams.

N grams are the fusion of multiple letters or multiple words. They are formed in such way that even the previous and next words are captured.

1. Uni gram is the unique words present the sentence.
2. Bi gram is the combination of 2 words.
3. Trigram is 3 words and so on

For example,

"I am learning NLP"

Unigrams: "I", "am", " learning", "NLP"

Bi grams: "I am", "am learning", "learning NLP"

Tri grams: "I am learning", "am learning NLP"

Problem

Generate the N grams for the given sentence.

Solution

There are lot of packages which will generate the N grams. The one which is mostly used is textblob.

How It Works

Following are the steps.

Data Processing for AI – Unstructured Data

Step 3-1 Generating N grams using Textblob

Lets just see how to generate N grams using Textblob and nltk packages.

Text = "I am learning NLP"

Use the below Textblob function to create N grams. Use the text which is defines above and mention the the "n" based in the requirement.

```
#For unigram :
TextBlob(Text).ngrams(1)
```

Output :

```
[WordList(['I']), WordList(['am']), WordList(['learning']), WordList(['NLP'])]
```

```
#For Bigram :
TextBlob(Text).ngrams(2)
```

OR

```
#Even nltk library can be used.
from nltk import ngrams

n = 4
fourgrams = ngrams(Text.split(), n)
```

Step 3-2 Bi-gram based features for a document

Just like in last recipe, we use count vectorizer to generate features. Using the same function, lest generate Bi gram features and see how the output looks like.

```
#importing the function
from sklearn.feature_extraction.text import CountVectorizer

# Text
text = ["I love NLP and I will learn NLP in 2month "]

# create the transform
vectorizer = CountVectorizer(ngram_range=(2,2))

# tokenizing
vectorizer.fit(text)

# encode document
vector = vectorizer.transform(text)

# summarize & generating output
```

```
print(vectorizer.vocabulary_)
print(vector.toarray())
```

Result:

```
{'love nlp': 3, 'nlp and': 4, 'and will': 0, 'will learn': 6, 'learn nlp': 2, 'nlp in': 5, 'in 2mo
nth': 1}
[[1 1 1 1 1 1 1]]
```

The ouptut has features with Bi grams and for our example, count is one for all the tokens.

Tutorial & Concept 3-4.

Again, in above mentioned text to feature methods, there are few draw backs hence the introduction of TF-IDF. Below are the disadvantages of above methods.

The whole idea of having TF-IDF is to reflect on how important a word is to a document in a collection or corpus and hence ignoring or normalizing words appeared frequently in all the documents

Let's say a particular word is appearing in all the documents of the corpus, it will get more importance in out previous methods. But TF-IDF again tries to normalize that as well.

Problem

Text to feature using TF-IDF.

Solution

Term frequency (TF): Term frequency is simply the ratio of the count of a word present in a sentence, to the length of the sentence.

TF is basically capturing the importance of the word irrespective of the length of the document. For example, a word with frequency of 3 with length of sentence being 10 is not same as when the word length of sentence is 100 words. It should get more importance in first scenario that is what TF does.

Inverse Document Frequency: IDF of each word is the log of the ratio of the total number of rows to the number of rows in a particular document in which that word is present.

IDF = log(N/n), where, N is the total number of rows and n is the number of rows in which the word was present.

IDF will measure the rareness of a term. Words like "a", "the" shows up in all the documents of the corpus, but rare words won't be there in all the documents. So, if a word is appearing in almost all documents, then that word is of no use to us since its not helping to classify or information retrieval. IDF will nullify this problem.

TF-IDF is the simple product of TF and IDF so that both the drawbacks are addressed which makes predictions and information retrieval relevant.

How It Works

Step 4-1 Read the text data

```
Text = ["The quick brown fox jumped over the lazy dog.",
"The dog.",
"The fox"]
```

Step 4-2 Execute below code on the text data

```
#Import TfidfVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer

#Create the transform
vectorizer = TfidfVectorizer()

#Tokenize and build vocab
vectorizer.fit(text)

#Summarize
print(vectorizer.vocabulary_)
print(vectorizer.idf_)
```

Result:

```
Text = ["The quick brown fox jumped over the lazy dog.",
"The dog.",
"The fox"]

{'the': 7, 'quick': 6, 'brown': 0, 'fox': 2, 'jumped': 3, 'over': 5, 'lazy': 4, 'dog': 1}

[ 1.69314718   1.28768207   1.28768207   1.69314718   1.69314718   1.69314718
  1.69314718   1.        ]
```

If you observe, “the” is appearing in all the 3 documents and it doesn’t add much value and hence the vector value is 1 which less than all the other vector representations of the tokens.

All these methods or techniques we have looked into so far are based on frequency and hence called as frequency based embedding’s or features. And in the next Tutorial & Conceptlet us look at prediction based embedding’s, typically called as word embedding’s.

Data Processing for AI – Unstructured Data

Tutorial & Concept 3-5.

Pre-requisite: This section assumes that you have a working knowledge of how a neural network works and the mechanisms by which weights in the neural network are updated. If new to Neural Network, would suggest you to go through chapter 1 to gain a basic understanding of how NN works.

Even though all previous methods solve the most of the problems, once we get into more complicated problems where we want to capture the semantic relation between the words or on huge data these methods fail to perform.

Below are the challenges:

1. All these techniques also fail to capture the context and meaning of the words. All the methods discussed so far, basically depends on the appearance or frequency of the words. But now let us look at how to capture the context or semantic relations. That is, how frequently the words are appearing close by.
    a. I am eating *apple*.
    b. I am using *apple*.

    If you observe the above example, apple gives different meaning when its used with different close by words, eating and using.

2. For a problem like document classification (book classification in library), a document is really huge and there are humungous number of tokens generated. In these scenarios, your number of features go out of control where in hampering the accuracy and performance

A machine/algorithm can match two documents/texts and say whether they are same or not. But how do we make machines tell you about cricket or Virat Kholi when you search for M S Dhoni? How do you make a machine understand that "Apple" in "Apple is a tasty fruit" is a fruit that can be eaten and not a company?

The answer to the above questions lie in creating a representation for words that capture their meanings, semantic relationships and the different types of contexts they are used in.

The above challenges are addressed by **Word Embeddings**.

Word embedding is the feature learning techniques where words from the vocabulary are mapped to vectors of real numbers capturing the contextual hierarchy.

Program: M.Tech in Artificial Intelligence
Batch: 01
Module: TRIM I Module IV
Credits: 04 (4 Saturdays)
Dates: 23rd, 30th Jan, 6th, 13th Feb 2021

REVA Academy for Corporate Excellence (RACE)

Data Processing for AI – Unstructured Data

Problem

You want to implement word embedding's.

Solution

Word embeddings are prediction based and they use shallow neural networks to train the model which will lead to learn the weight and use them as vector representation.

word2vec: word2vec is the deep leaning google framework to train word embeddings. It will use all the words of whole corpus and predicts the nearby words. It will create a vector for all the words present in the corpus in a way so that context is captures. It also outperforms any other methodologies in the space of word similarity and word analogies.

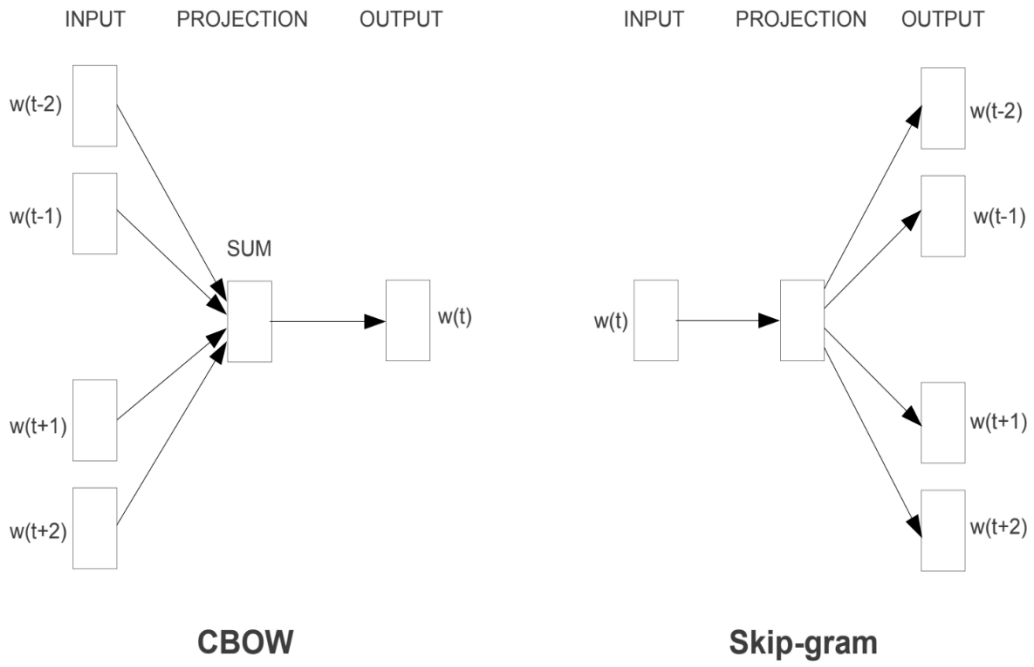
There are mainly 2 types in word2vec.

a. Skip gram

b. Continuous Bag of Words (CBOW)

Data Processing for AI – Unstructured Data

How It Works



Reference from "Efficient Estimation of Word Representations in Vector Space", 2013.

The image shows the architecture of the CBOW and skip gram algorithms used to build word embeddings. Let's see how these models work in detail.

a) Skip gram:

The skip-gram model (Mikolov et al., 2013) is used to predict the probabilities of a word given the context word or words.

Let's take a small sentence and understand how it actually works. Each sentence will generate target word and context which is the words nearby. The number of words to be considered around target variable is called window size. The below shows all the possible target and context variable for window size 2. Window size needs to be selected based on data and the resources at your disposal. More the window size, higher the computing power.

Text = "I love NLP and I will learn NLP in 2 months"

	Target word	Context
I love NLP	I	love, NLP

Data Processing for AI – Unstructured Data

I love NLP and	love	love, NLP, and
I love NLP and I will learn	NLP	I, love, and, I
…	…	…
in 2 months	month	in,2

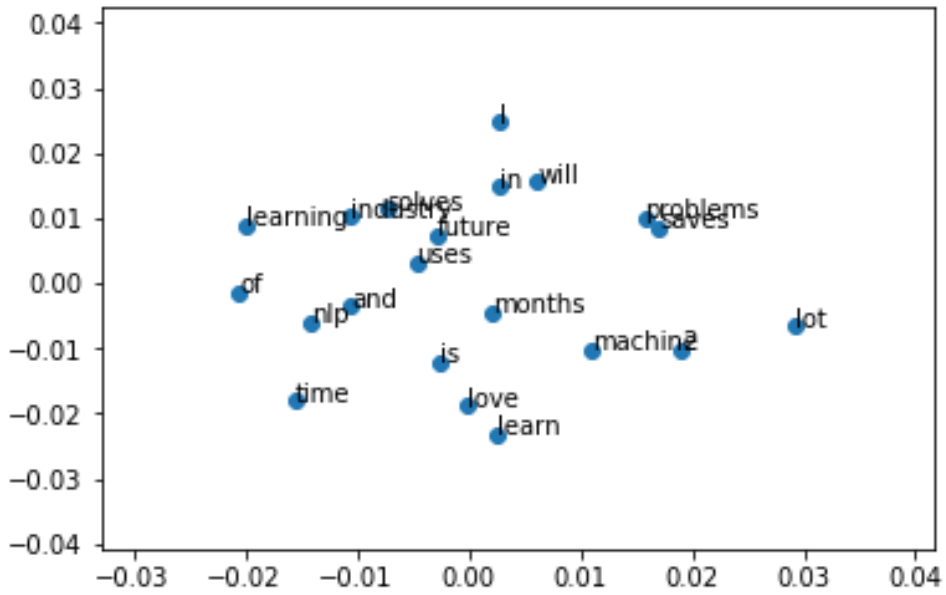
Since it takes lot of text and computing power lets go ahead and take sample data and build skip gram model.

As mentioned in the 3rd chapter, import the text corpus and break them into sentences. Perform some cleaning and preprocessing like removal of punctuation, digits and split the sentences into words or tokens etc.

```
#Example sentences


sentences = [['I', 'love', 'nlp'],
                    ['I', 'will', 'learn', 'nlp', 'in', '2','months'],
                    ['nlp', 'is', 'future'],
                    ['nlp', 'saves', 'time', 'and', 'solves', 'lot', 'of', 'industry',
'problems'],
                    ['nlp', 'uses', 'machine', 'learning']]


# training the model
skipgram = Word2Vec(sentences, size =50, window = 3, min_count=1,sg = 1)
print(skipgram)

# access vector for one word
print(skipgram['nlp'])

# save model
skipgram.save('skipgram.bin')

# load model
skipgram = Word2Vec.load('skipgram.bin')

# vizualize
X = skipgram[skipgram.wv.vocab]
pca = PCA(n_components=2)
result = pca.fit_transform(X)

# create a scatter plot of the projection
pyplot.scatter(result[:, 0], result[:, 1])
words = list(skipgram.wv.vocab)
for i, word in enumerate(words):
       pyplot.annotate(word, xy=(result[i, 0], result[i, 1]))
pyplot.show()
```

Result :

Data Processing for AI – Unstructured Data



But to train these models, it requires huge amount of computing power. So, let's go ahead and use Google's pretrained model which was trained over 100 billion words.

Download the model from below path and keep it in your local.

https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit

Import the genism package and follow the steps to understand Google's word2vec.

```
# import genism package
import  genism


# load the saved model

model = gensim.models.Word2Vec.load_word2vec_format('C:\\Users\\GoogleNews-vectors-
negative300.bin', binary=True)


#Checking how similarity works.

print (model.similarity('this', 'is'))

Output:
0.407970363878
```

Data Processing for AI – Unstructured Data

```
#Lets check one more.
print (model.similarity('post', 'book'))

Output:
0.0572043891977
```

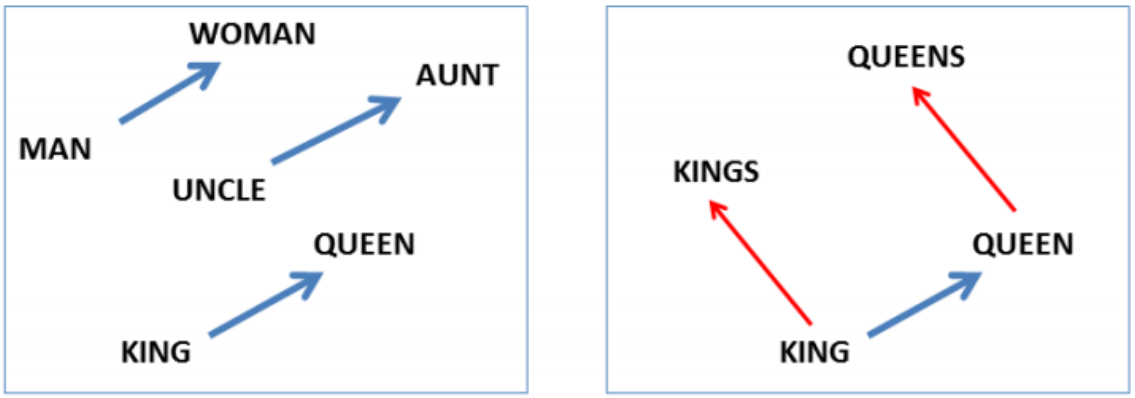
"This" and "is" has good amount of similarity but similarity between words "post" and "book" are poor. For any given set of words, it uses the vectors of both the words and calculates similarity between the words.

```
# Finding odd one out.

model.doesnt_match('breakfast cereal dinner lunch';.split())

Output:
'cereal'
```

Of 'breakfast', 'cereal', 'dinner' and 'lunch', only cereal is the word which is not anywhere related to remaining 3 words.

```
# Its also finding the relations between words.

word_vectors.most_similar(positive=['woman', 'king'], negative=['man'])
Output:

queen: 0.7699
```

If you add 'woman' and 'king' and minus man, its predicting queen as output with 77% confidence. Isn't it amazing?



(Mikolov et al., NAACL HLT, 2013)

Data Processing for AI – Unstructured Data

T - SNE plots:

T - SNE plot is one of the ways to evaluate word embeddings. If we see the below plot, this is representation of car manufactures.

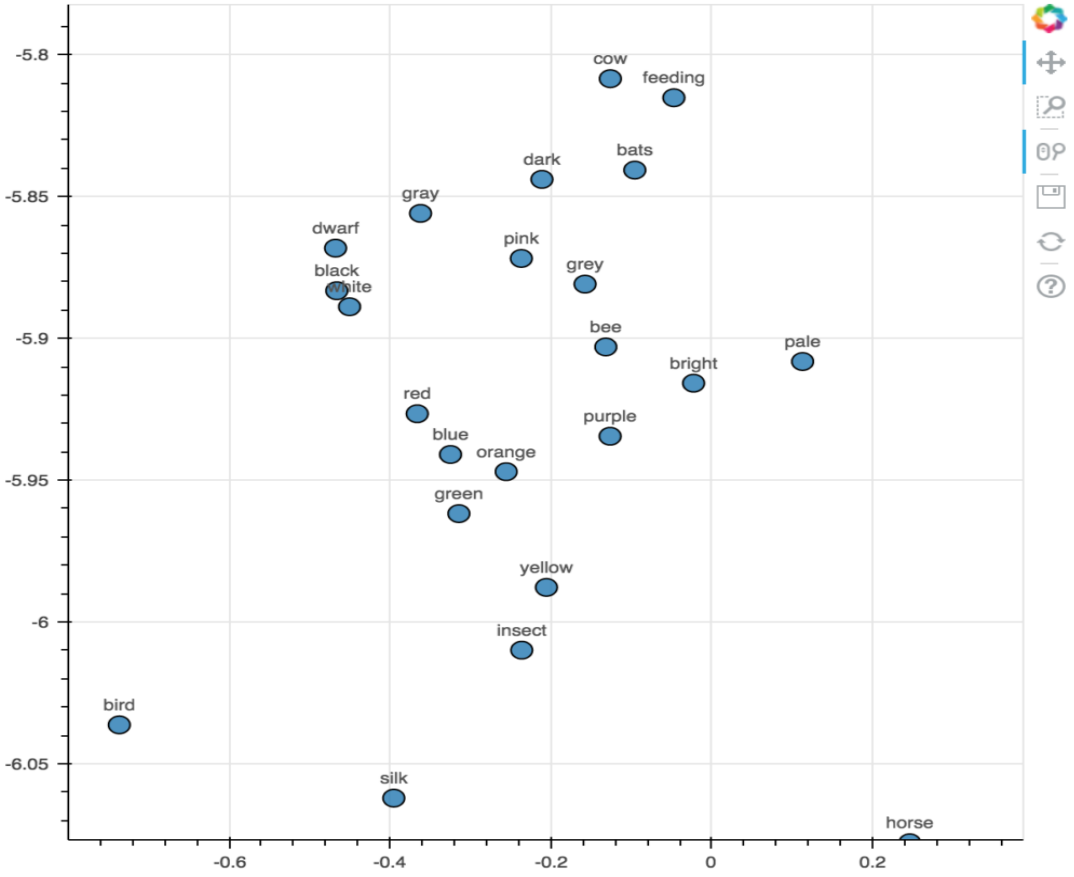
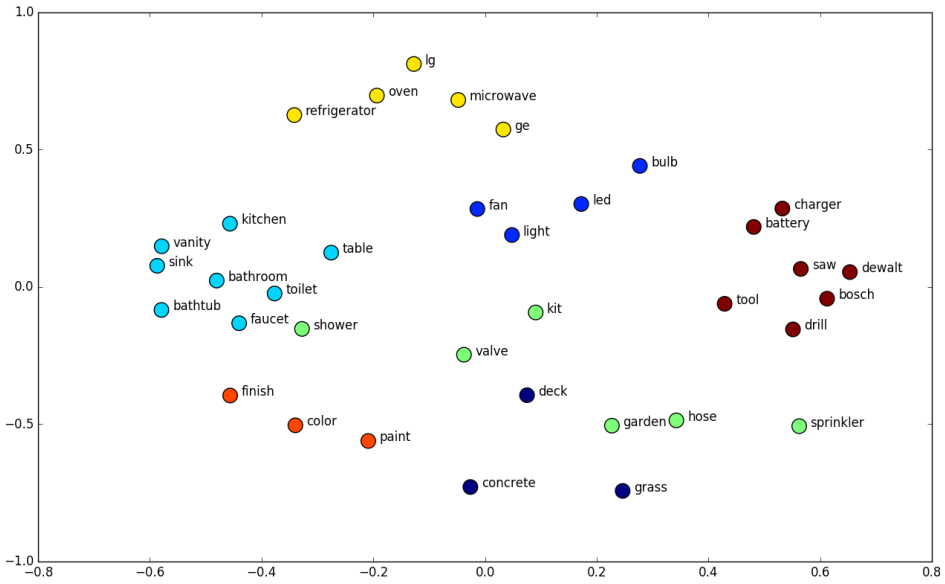


Image: https://nlpforhackers.io/word-embeddings/

This is the output from a word embeddings. If you closely observe all the colors are located to close to each other, for example "orange", "red"," blue" are near to each other. In the same way, "cow", "bats" are closer.

Data Processing for AI – Unstructured Data



Above is the word embedding's output representation of home interiors and exterior.

If you clearly observe all the words related to electric fittings are near to each other, similarly words related to bathroom fittings and so on. This is beauty of word embedding's.

FastText

FastText is another deep learning framework developed by Facebook.

It's the improvised version of word2vec. Word2vec basically considers words to build the representation. But FastText takes each character while computing the representation of the word.

Let's see how to build a FastText word embeddings.

```
# Import FastText

from gensim.models import FastText


# Training the model
```

```
model = FastText(Text,size=, window=, min_count=, workers=, min_n=, max_n=)
```

You can use the above syntax to train the model. Again, since it takes lot of time to train the model, we are not doing it here.

One advantage of using FastText is, let's say a particular word is not present in word2vec, you won't get output for that word. But since FastText is build on character level, even for the word which was not there in training it will provide results. FastText will take longer time to train the model but gives good result.

Introduction

Before we start, have a look at the below examples.

1. You open Google and search for a news article on the ongoing Champions trophy and get hundreds of search results in return about it.
2. Nate silver analysed millions of tweets and correctly predicted the results of 49 out of 50 states in 2008 U.S Presidential Elections.
3. You type a sentence in google translate in English and get an Equivalent Chinese conversion.

So what do the above examples have in common?

You possible guessed it right – **TEXT processing**. All the above three scenarios deal with humongous amount of text to perform different range of tasks like clustering in the google search example, classification in the second and Machine Translation in the third.

Humans can deal with text format quite intuitively but provided we have millions of documents being generated in a single day, we cannot have humans performing the above the three tasks. It is neither scalable nor effective.

So, how do we make computers of today perform clustering, classification etc on a text data since we know that they are generally inefficient at handling and processing strings or texts for any fruitful outputs?

Data Processing for AI – Unstructured Data

Sure, a computer can match two strings and tell you whether they are same or not. But how do we make computers tell you about football or Ronaldo when you search for Messi? How do you make a computer understand that “Apple” in “Apple is a tasty fruit” is a fruit that can be eaten and not a company?

The answer to the above questions lie in creating a representation for words that capture their *meanings*, *semantic relationships* and the different types of contexts they are used in.

And all of these are implemented by using Word Embeddings or numerical representations of texts so that computers may handle them.

Below, we will see formally what are Word Embeddings and their different types and how we can actually implement them to perform the tasks like returning efficient Google search results.

Table of Contents

1. What are Word Embeddings?

In very simplistic terms, Word Embeddings are the texts converted into numbers and there may be different numerical representations of the same text. But before we dive into the details of Word Embeddings, the following question should be asked – Why do we need Word Embeddings?

As it turns out, many Machine Learning algorithms and almost all Deep Learning Architectures are incapable of processing *strings* or *plain text* in their raw form. They require numbers as inputs to perform

any sort of job, be it classification, regression etc. in broad terms. And with the huge amount of data that is present in the text format, it is imperative to extract knowledge out of it and build applications. Some real world applications of text applications are – sentiment analysis of reviews by Amazon etc., document or news classification or clustering by Google etc.

Let us now define Word Embeddings formally. A Word Embedding format generally tries to map a word using a dictionary to a vector. Let us break this sentence down into finer details to have a clear view.

Take a look at this example – **sentence**=" Word Embeddings are Word converted into numbers "

A *word* in this **sentence** may be "Embeddings" or "numbers " etc.

A *dictionary* may be the list of all unique words in the **sentence.** So, a dictionary may look like – ['Word','Embeddings','are','Converted','into','numbers']

A ve*ctor* representation of a word may be a one-hot encoded vector where 1 stands for the position where the word exists and 0 everywhere else. The vector representation of "numbers" in this format according to the above dictionary is [0,0,0,0,0,1] and of converted is[0,0,0,1,0,0].

This is just a very simple method to represent a word in the vector form. Let us look at different types of Word Embeddings or Word Vectors and their advantages and disadvantages over the rest.

2. Different types of Word Embeddings

The different types of word embeddings can be broadly classified into two categories-

1. Frequency based Embedding
2. Prediction based Embedding

Let us try to understand each of these methods in detail.

2.1 Frequency based Embedding

There are generally three types of vectors that we encounter under this category.

1. Count Vector
2. TF-IDF Vector

Let us look into each of these vectorization methods in detail.

2.1.1 Count Vector

Consider a Corpus C of D documents {d1,d2…..dD} and N unique tokens extracted out of the corpus C. The N tokens will form our dictionary and the size of the Count Vector matrix M will be given by D X N. Each row in the matrix M contains the frequency of tokens in document D(i).

Let us understand this using a simple example.

D1: He is a lazy boy. She is also lazy.

D2: Neeraj is a lazy person.

The dictionary created may be a list of unique tokens(words) in the corpus =['He','She','lazy','boy','Neeraj','person']

Here, D=2, N=6

The count matrix M of size 2 X 6 will be represented as –

	He	She	lazy	boy	Neeraj	person
D1	1	1	2	1	0	0
D2	0	0	1	0	1	1

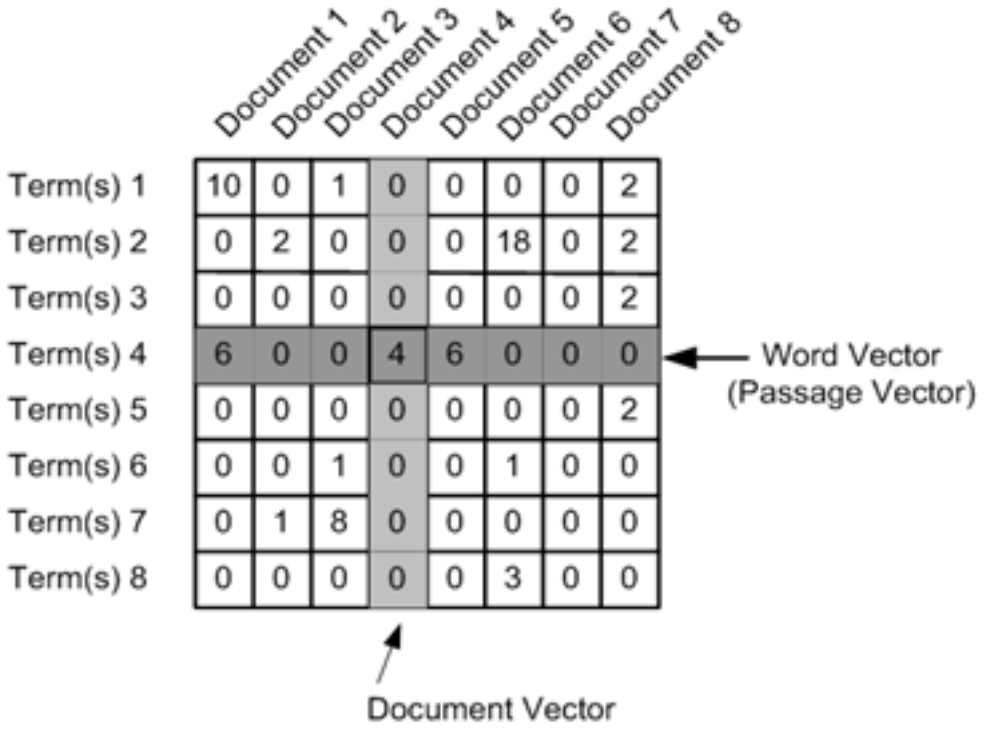
Now, a column can also be understood as word vector for the corresponding word in the matrix M. For example, the word vector for 'lazy' in the above matrix is [2,1] and so on.Here, the *rows* correspond to the *documents* in the corpus and the *columns* correspond to the *tokens* in the dictionary. The second row in the above matrix may be read as – D2 contains 'lazy': once, 'Neeraj': once and 'person' once.

Now there may be quite a few variations while preparing the above matrix M. The variations will be generally in-

Data Processing for AI – Unstructured Data

1. The way dictionary is prepared.
   Why? Because in real world applications we might have a corpus which contains millions of documents. And with millions of document, we can extract hundreds of millions of unique words. So basically, the matrix that will be prepared like above will be a very sparse one and inefficient for any computation. So an alternative to using every unique word as a dictionary element would be to pick say top 10,000 words based on frequency and then prepare a dictionary.
2. The way count is taken for each word.
   We may either take the frequency (number of times a word has appeared in the document) or the presence(has the word appeared in the document?) to be the entry in the count matrix M. But generally, frequency method is preferred over the latter.

Below is a representational image of the matrix M for easy understanding.



Data Processing for AI – Unstructured Data

2.1.2 TF-IDF vectorization

This is another method which is based on the frequency method but it is different to the count vectorization in the sense that it takes into account not just the occurrence of a word in a single document but in the entire corpus. So, what is the rationale behind this? Let us try to understand.

Common words like ‘is’, ‘the’, ‘a’ etc. tend to appear quite frequently in comparison to the words which are important to a document. For example, a document **A** on Lionel Messi is going to contain more occurences of the word “Messi” in comparison to other documents. But common words like “the” etc. are also going to be present in higher frequency in almost every document.

Ideally, what we would want is to down weight the common words occurring in almost all documents and give more importance to words that appear in a subset of documents.

TF-IDF works by penalising these common words by assigning them lower weights while giving importance to words like Messi in a particular document.

So, how exactly does TF-IDF work?

Consider the below sample table which gives the count of terms(tokens/words) in two documents.

Document 1

Term	Count
This	1
is	1
about	2
Messi	4

Document 2

Term	Count
This	1
is	2
about	1
Tf-idf	1

Now, let us define a few terms related to TF-IDF.

Data Processing for AI – Unstructured Data

TF = (Number of times term t appears in a document)/(Number of terms in the document)

So, TF(This,Document1) = 1/8

TF(This, Document2)=1/5

It denotes the contribution of the word to the document i.e words relevant to the document should be frequent. eg: A document about Messi should contain the word 'Messi' in large number.

IDF = log(N/n), where, N is the number of documents and n is the number of documents a term t has appeared in.

where N is the number of documents and n is the number of documents a term t has appeared in.

So, IDF(This) = log(2/2) = 0.

So, how do we explain the reasoning behind IDF? Ideally, if a word has appeared in all the document, then probably that word is not relevant to a particular document. But if it has appeared in a subset of documents then probably the word is of some relevance to the documents it is present in.

Let us compute IDF for the word 'Messi'.

IDF(Messi) = log(2/1) = 0.301.

Now, let us compare the TF-IDF for a common word 'This' and a word 'Messi' which seems to be of relevance to Document 1.

TF-IDF(This,Document1) = (1/8) * (0) = 0

TF-IDF(This, Document2) = (1/5) * (0) = 0

TF-IDF(Messi, Document1) = (4/8)*0.301 = 0.15

As, you can see for Document1 , TF-IDF method heavily penalises the word 'This' but assigns greater weight to 'Messi'. So, this may be understood as 'Messi' is an important word for Document1 from the context of the entire corpus.

2.2 Prediction based Vector

Pre-requisite: This section assumes that you have a working knowledge of how a neural network works and the mechanisms by which weights in an NN are updated. If you are new to Neural Network, I would suggest you go through this awesome article by Sunil to gain a very good understanding of how NN works.

So far, we have seen deterministic methods to determine word vectors. But these methods proved to be limited in their word representations until Mitolov etc. el introduced word2vec to the NLP community. These methods were prediction based in the sense that they provided probabilities to the words and proved to be state of the art for tasks like word analogies and word similarities. They were also able to achieve tasks like King -man +woman = Queen, which was considered a result almost magical. So let us look at the word2vec model used as of today to generate word vectors.

Word2vec is not a single algorithm but a combination of two techniques – CBOW(Continuous bag of words) and Skip-gram model. Both of these are shallow neural networks which map word(s) to the target variable which is also a word(s). Both of these techniques learn weights which act as word vector representations. Let us discuss both these methods separately and gain intuition into their working.

2.2.1 CBOW (Continuous Bag of words)

The way CBOW work is that it tends to predict the probability of a word given a context. A context may be a single word or a group of words. But for simplicity, I will take a single context word and try to predict a single target word.

Suppose, we have a corpus C = “Hey, this is sample corpus using only one context word.” and we have defined a context window of 1. This corpus may be converted into a training set for a CBOW model as follow. The input is shown below. The matrix on the right in the below image contains the one-hot encoded from of the input on the left.

Data Processing for AI – Unstructured Data

Input	Output			Hey	This	is	sample	corpus	using	only	one	context	word
Hey	this		Datapoint 1	1	0	0	0	0	0	0	0	0	0
this	hey		Datapoint 2	0	1	0	0	0	0	0	0	0	0
is	this		Datapoint 3	0	0	1	0	0	0	0	0	0	0
is	sample		Datapoint 4	0	0	1	0	0	0	0	0	0	0
sample	is		Datapoint 5	0	0	0	1	0	0	0	0	0	0
sample	corpus		Datapoint 6	0	0	0	1	0	0	0	0	0	0
corpus	sample		Datapoint 7	0	0	0	0	1	0	0	0	0	0
corpus	using		Datapoint 8	0	0	0	0	1	0	0	0	0	0
using	corpus		Datapoint 9	0	0	0	0	0	1	0	0	0	0
using	only		Datapoint 10	0	0	0	0	0	1	0	0	0	0
only	using		Datapoint 11	0	0	0	0	0	0	1	0	0	0
only	one		Datapoint 12	0	0	0	0	0	0	1	0	0	0
one	only		Datapoint 13	0	0	0	0	0	0	0	1	0	0
one	context		Datapoint 14	0	0	0	0	0	0	0	1	0	0
context	one		Datapoint 15	0	0	0	0	0	0	0	0	1	0
context	word		Datapoint 16	0	0	0	0	0	0	0	0	1	0
word	context		Datapoint 17	0	0	0	0	0	0	0	0	0	1

The target for a single datapoint say Datapoint 4 is shown as below

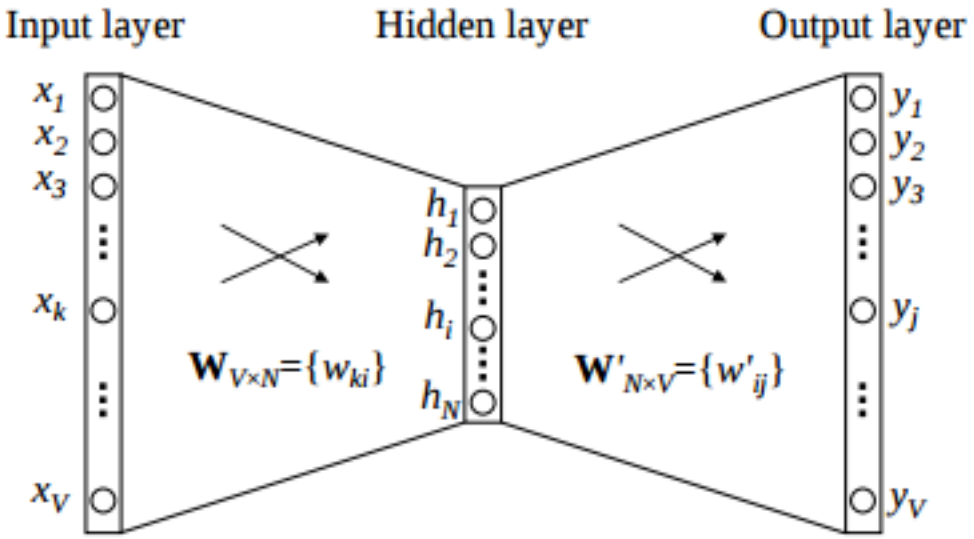
Hey	this	is	sample	corpus	using	only	one	context	word
0	0	0	1	0	0	0	0	0	0

This matrix shown in the above image is sent into a shallow neural network with three layers: an input layer, a hidden layer and an output layer. The output layer is a softmax layer which is used to sum the probabilities obtained in the output layer to 1. Now let us see how the forward propagation will work to calculate the hidden layer activation.

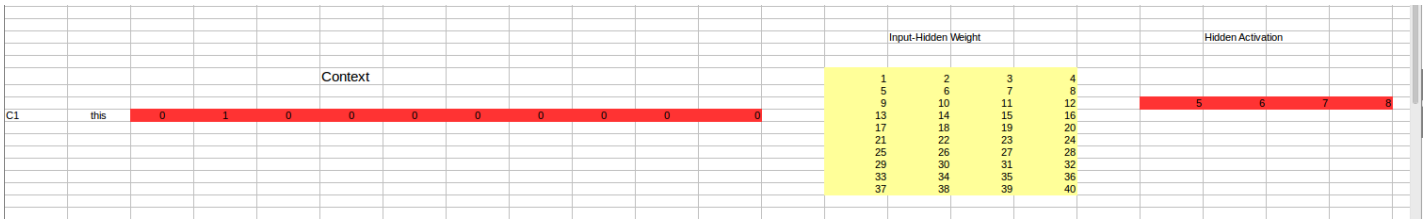
Let us first see a diagrammatic representation of the CBOW model.

Program: M.Tech in Artificial Intelligence
Batch: 01
Module: TRIM I Module IV
Credits: 04 (4 Saturdays)
Dates: 23rd, 30th Jan, 6th, 13th Feb 2021

Data Processing for AI – Unstructured Data



The matrix representation of the above image for a single data point is below.

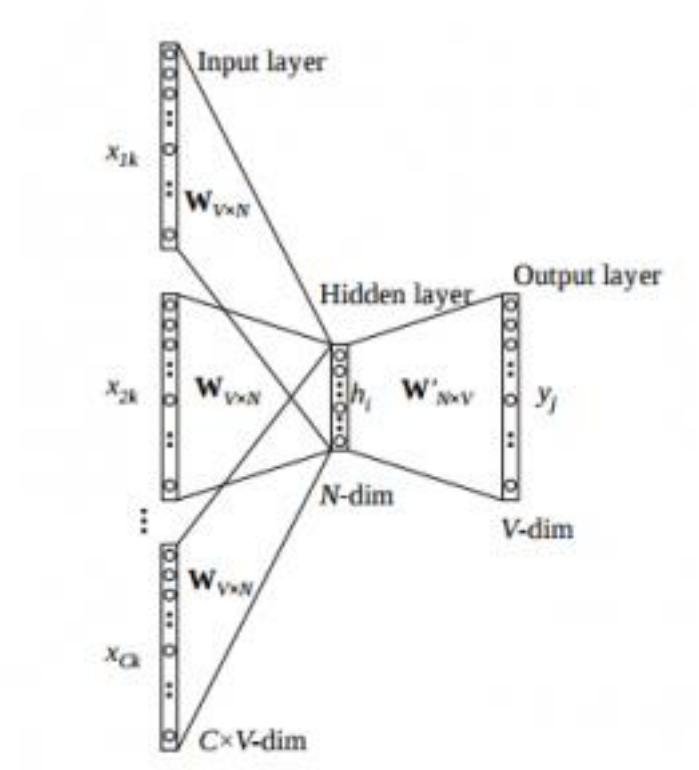


The flow is as follows:

1. The input layer and the target, both are one- hot encoded of size [1 X V]. Here V=10 in the above example.
2. There are two sets of weights. one is between the input and the hidden layer and second between hidden and output layer.
   Input-Hidden layer matrix size =[V X N] , hidden-Output layer matrix size =[N X V] : Where N is the number of dimensions we choose to represent our word in. It is arbitary and a hyper-parameter for a Neural Network. Also, N is the number of neurons in the hidden layer. Here, N=4.
3. There is a no activation function between any layers.(More specifically, I am referring to linear activation)
4. The input is multiplied by the input-hidden weights and called hidden activation. It is simply the corresponding row in the input-hidden matrix copied.

5. The hidden input gets multiplied by hidden- output weights and output is calculated.
6. Error between output and target is calculated and propagated back to re-adjust the weights.
7. The weight between the hidden layer and the output layer is taken as the word vector representation of the word.

We saw the above steps for a single context word. Now, what about if we have multiple context words? The image below describes the architecture for multiple context words.



Below is a matrix representation of the above architecture for an easy understanding.

Context

C1	this	0	1	0	0	0	0	0	0	0	0
C2	corpus	0	0	0	0	1	0	0	0	0	0
C3	context	0	0	0	0	0	0	0	0	1	0

Input-Hidden Weight

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32
33	34	35	36
37	38	39	40

Hidden Activation

5	6	7	8
17	18	19	20
33	34	35	36

Average hidden Activation

18.33333333	19.33333333	20.33333333	21.33333333

Data Processing for AI – Unstructured Data

The image above takes 3 context words and predicts the probability of a target word. The input can be assumed as taking three one-hot encoded vectors in the input layer as shown above in red, blue and green.

So, the input layer will have 3 [1 X V] Vectors in the input as shown above and 1 [1 X V] in the output layer. Rest of the architecture is same as for a 1-context CBOW.

The steps remain the same, only the calculation of hidden activation changes. Instead of just copying the corresponding rows of the input-hidden weight matrix to the hidden layer, an average is taken over all the corresponding rows of the matrix. We can understand this with the above figure. The average vector calculated becomes the hidden activation. So, if we have three context words for a single target word, we will have three initial hidden activations which are then averaged element-wise to obtain the final activation.

In both a single context word and multiple context word, I have shown the images till the calculation of the hidden activations since this is the part where CBOW differs from a simple MLP network. The steps after the calculation of hidden layer are same as that of the MLP as mentioned in this article – Understanding and Coding Neural Networks from scratch.

The differences between MLP and CBOW are mentioned below for clarification:

1. The objective function in MLP is a MSE(mean square error) whereas in CBOW it is negative log likelihood of a word given a set of context i.e -log(p(wo/wi)), where p(wo/wi) is given as

$$p(w_O|w_I) = \frac{\exp\left({v'_{w_O}}^{\top} v_{w_I}\right)}{\sum_{w=1}^{W} \exp\left({v'_w}^{\top} v_{w_I}\right)}$$

wo : output word
wi: context words

2. The gradient of error with respect to hidden-output weights and input-hidden weights are different since MLP has sigmoid activations(generally) but CBOW has linear activations. The method however to calculate the gradient is same as an MLP.

Advantages of CBOW:

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

Disadvantages of CBOW:

1. CBOW takes the average of the context of a word (as seen above in calculation of hidden activation). For example, Apple can be both a fruit and a company but CBOW takes an average of both the contexts and places it in between a cluster for fruits and companies.
2. Training a CBOW from scratch can take forever if not properly optimized.

2.2.2 Skip – Gram model

Skip – gram follows the same topology as of CBOW. It just flips CBOW's architecture on its head. The aim of skip-gram is to predict the context given a word. Let us take the same corpus that we built our CBOW model on. C="Hey, this is sample corpus using only one context word." Let us construct the training data.

Data Processing for AI – Unstructured Data

Input	Output(Context1)	Output(Context2)
Hey	this	<padding>
this	Hey	is
is	this	sample
sample	is	corpus
corpus	sample	corpus
using	corpus	only
only	using	one
one	only	context
context	one	word
word	context	<padding>

The input vector for skip-gram is going to be similar to a 1-context CBOW model. Also, the calculations up to hidden layer activations are going to be the same. The difference will be in the target variable. Since we have defined a context window of 1 on both the sides, there will be "**two" one hot encoded target variables** and "**two" corresponding outputs** as can be seen by the blue section in the image.

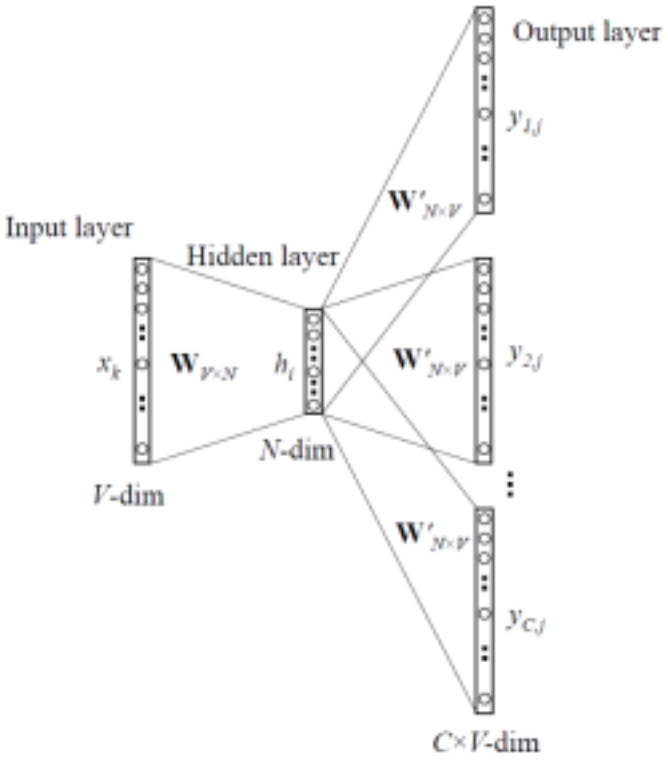
Two separate errors are calculated with respect to the two target variables and the two error vectors obtained are added element-wise to obtain a final error vector which is propagated back to update the weights.

The weights between the input and the hidden layer are taken as the word vector representation after training. The loss function or the objective is of the same type as of the CBOW model.

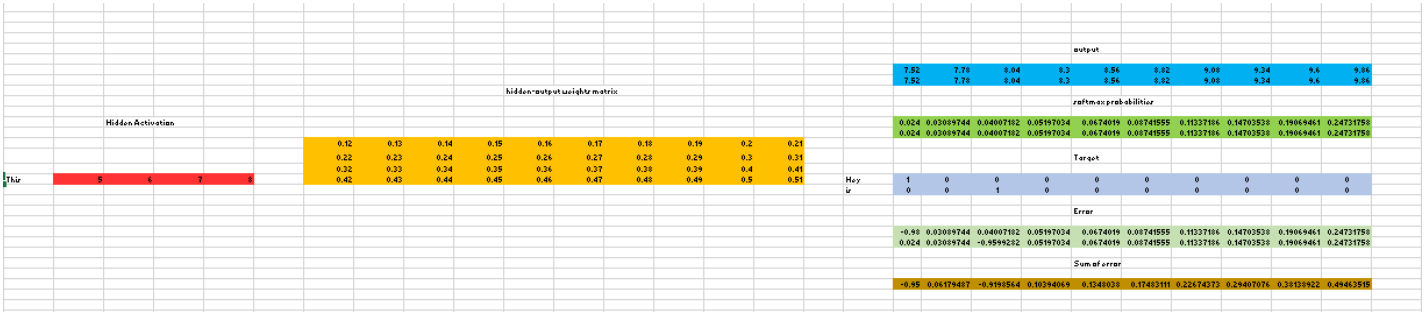
The skip-gram architecture is shown below.

Program: M.Tech in Artificial Intelligence
Batch: 01
Module: TRIM I Module IV
Credits: 04 (4 Saturdays)
Dates: 23rd, 30th Jan, 6th, 13th Feb 2021

Data Processing for AI – Unstructured Data



For a better understanding, matrix style structure with calculation has been shown below.



Let us break down the above image.

Input layer size – [1 X V], Input hidden weight matrix size – [V X N], Number of neurons in hidden layer – N, Hidden-Output weight matrix size – [N X V], Output layer size – C [1 X V]

In the above example, C is the number of context words=2, V= 10, N=4

1. The row in red is the hidden activation corresponding to the input one-hot encoded vector. It is basically the corresponding row of input-hidden matrix copied.
2. The yellow matrix is the weight between the hidden layer and the output layer.
3. The blue matrix is obtained by the matrix multiplication of hidden activation and the hidden output weights. There will be two rows calculated for two target(context) words.
4. Each row of the blue matrix is converted into its softmax probabilities individually as shown in the green box.
5. The grey matrix contains the one hot encoded vectors of the two context words(target).
6. Error is calculated by substracting the first row of the grey matrix(target) from the first row of the green matrix(output) element-wise. This is repeated for the next row. Therefore, for **n** target context words, we will have **n** error vectors.
7. Element-wise sum is taken over all the error vectors to obtain a final error vector.
8. This error vector is propagated back to update the weights.

Advantages of Skip-Gram Model

1. Skip-gram model can capture two semantics for a single word. i.e it will have two vector representations of Apple. One for the company and other for the fruit.
2. Skip-gram with negative sub-sampling outperforms every other method generally.

This is an excellent interactive tool to visualise CBOW and skip gram in action. I would suggest you to really go through this link for a better understanding.

3. Word Embeddings use case scenarios

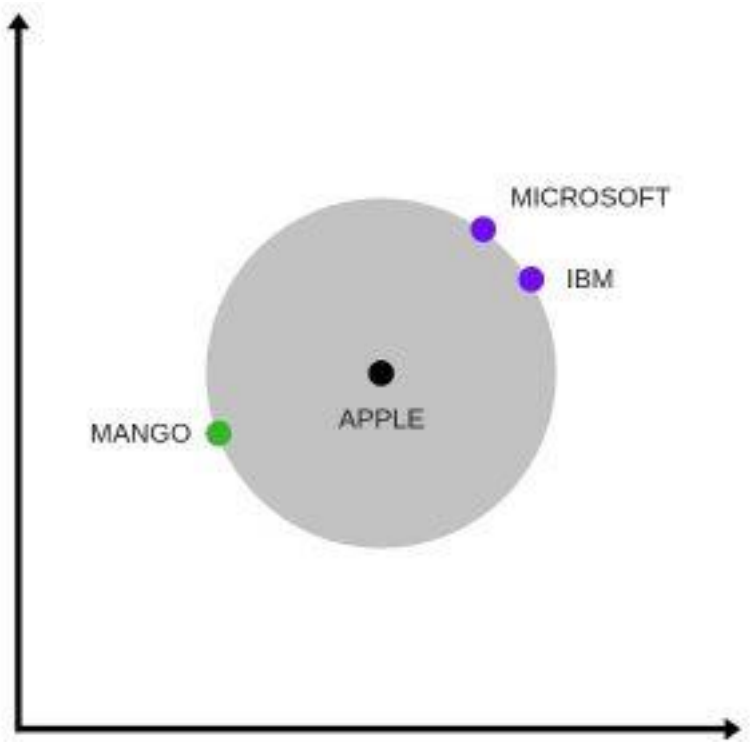
Since word embeddings or word Vectors are numerical representations of contextual similarities between words, they can be manipulated and made to perform amazing tasks like-

1. Finding the degree of similarity between two words.
```
model.similarity('woman','man')
0.73723527
```
2. Finding odd one out.
```
model.doesnt_match('breakfast cereal dinner lunch';.split())
'cereal'
```

Data Processing for AI – Unstructured Data

3. Amazing things like woman+king-man =queen
```
model.most_similar(positive=['woman','king'],negative=['man'],topn=1)
queen: 0.508
```
4. Probability of a text under the model
```
model.score(['The fox jumped over the lazy dog'.split()])
0.21
```

Below is one interesting visualisation of word2vec.



The above image is a t-SNE representation of word vectors in 2 dimension and you can see that two contexts of apple have been captured. One is a fruit and the other company.

5. It can be used to perform Machine Translation.

The above graph is a bilingual embedding with chinese in green and english in yellow. If we know the words having similar meanings in chinese and english, the above bilingual embedding can be used to translate one language into the other.

4. Using pre-trained word vectors

We are going to use google's pre-trained model. It contains word vectors for a vocabulary of 3 million words trained on around 100 billion words from the google news dataset. The downlaod link for the model is this. Beware it is a 1.5 GB download.

```
from gensim.models import Word2Vec

#loading the downloaded model
model = Word2Vec.load_word2vec_format('GoogleNews-vectors-
negative300.bin', binary=True, norm_only=True)

#the model is loaded. It can be used to perform all of the tasks mentioned above.

# getting word vectors of a word
dog = model['dog']
```

```
#performing king queen magic
print(model.most_similar(positive=['woman', 'king'], negative=['man']))

#picking odd one out
print(model.doesnt_match("breakfast cereal dinner lunch".split()))

#printing similarity index
print(model.similarity('woman', 'man'))
```

5. Training your own word vectors

We will be training our own word2vec on a custom corpus. For training the model we will be using gensim and the steps are illustrated as below.

word2Vec requires that a format of list of list for training where every document is contained in a list and every list contains list of tokens of that documents. I won't be covering the pre-preprocessing part here. So let's take an example list of list to train our word2vec model.

sentence=[['Neeraj','Boy'],['Sarwan','is'],['good','boy']]

```
#training word2vec on 3 sentences
model = gensim.models.Word2Vec(sentence, min_count=1,size=300,workers=4)
```

Let us try to understand the parameters of this model.

sentence – list of list of our corpus
min_count=1 -the threshold value for the words. Word with frequency greater than this only are going to be included into the model.
size=300 – the number of dimensions in which we wish to represent our word. This is the size of the word vector.
workers=4 – used for parallelization

```
#using the model
#The new trained model can be used similar to the pre-trained ones.

#printing similarity index
print(model.similarity('woman', 'man'))
```