Istanbul Technical University - Science and Letters Faculty

Mathematics Engineering Program

Graduation Project

Student Name: Arif Çakır

Student Number: 090190355

Course: MAT4091E

Advisor: Prof. Dr. Atabey Kaygun

Submission Date: June 8, 2023

Contents

1 Introduction

Language is a fundamental aspect of human communication and one of the most common ways to pass on information and culture throughout history. Due to their nature of encapsulating information, different languages are studied by linguists throughout the years. Thanks to this scientific foundation, machine learning, and computer science are also advanced upon this topic. Natural Language Processing (NLP) is the subfield of machine learning that studies the processing of human language by computers. One of the most popular NLP techniques in recent years is Word Embedding, which represents words as vectors in semantic space that allows applying mathematical operations on them to analyse. Word Embedding can be used on various subjects such as text classification, translation, and sentiment analysis with promising results. It is aimed in this paper to study several Word Embedding techniques and their applications. The aim is to create a model by applying various Word Embedding models and with the help of this model, describing the relationships and similarities of languages.

The rest of the paper is organised as follows: In Section 2, autoencoders are discussed to understand the idea behind word embedding. After that, section 3 contains information about Word Embedding in general and several Word Embedding techniques are provided. Word Embedding techniques touched on in the paper are Term Frequency (TF), Inverse Document Frequency (IDF), Word2Vec, Global Vectors for Word Representation (GloVe), Bidirectional Encoder Representations from Transformers (BERT). This is followed by Section 4 which introduces the dataset The Divine Comedy by Dante Alighieri. After that, several Word Embedding models are applied to the dataset and the outputs of those models are analysed. Finally paper ends with some discussion and a conclusion in Section 6.

2 Word Embedding Techniques

Word Embedding is one of the most popular natural language processing techniques due to its vector representation of the words. As Agarwal stated, capturing semantic meaning of the words in a vector of text is the ambition of the word embedding techniques (Agarwal,

2022). With respect to that, some of the most popular word embedding techniques will be studied in this section. Those techniques are TF, which counts rarity of words; IDF, which counts rarity of words; Skip-Gram method, which is used by Word2Vec; Continuous Bag of Words method, that also used by Word2Vec; GloVe, which captures co-occurrence of words; and BERT, family of masked-language models introduced by Google.

2.1 Term Frequency - Inverse Document Frequency

Term Frequency (TF) is a word embedding technique which counts the occurrence of the words in a document. TF can be shown as

$$TF(i) = \frac{\log(Frequency(i,j))}{\log(TotalNumber(j))}. \tag{1}$$

where $Frequency(i,j)$ is the frequency of a word that occurred in a j word document and $TotalNumber(j)$ is the total number of the words in the document.

On the other hand, Inverse Document Frequency (IDF) is practically the opposite of the TF method. In this method, algorithm relies on the information that gained from the words which are rarely used. IDF can be written as

$$IDF(i) = \log(\frac{TotalNumber(j)}{Frequency(j,i)}). \tag{2}$$

where $Frequency(i,j)$ is the frequency of a word that occurred in a j word document and $TotalNumber(j)$ is the total number of the words in the document.

TF-IDF mainly shows the degree of relevancy of word i in the document j, while the main disadvantage of TF-IDF is it does not grasp contextual relationship between the words. As Kınık and Güran stated, TF-IDF does not capture semantic relationship between words, and accepts them as independent values (Kınık and Güran, 2021). Due to TF-IDF's lack of capturing semantic relationship of the words, TF-IDF mainly used to detect stop words.

2.2 Skip-Gram Algorithm

Skip-Gram model is used by word embedding tasks of Word2Vec model, which will be discussed in following chapters. Dive Into Deep Learning (n.d) states that a word can be used for generating its surrounding words in skip-gram model. For example, if the sentence "the man loves his son" is taken and "loves" is chosen as the center word, then skip-gram model considers the conditional probability for generating the words. This architecture can be seen in **Figure 1**. Due to this approach, each word has a two dimensional vector representations in skip-gram model.

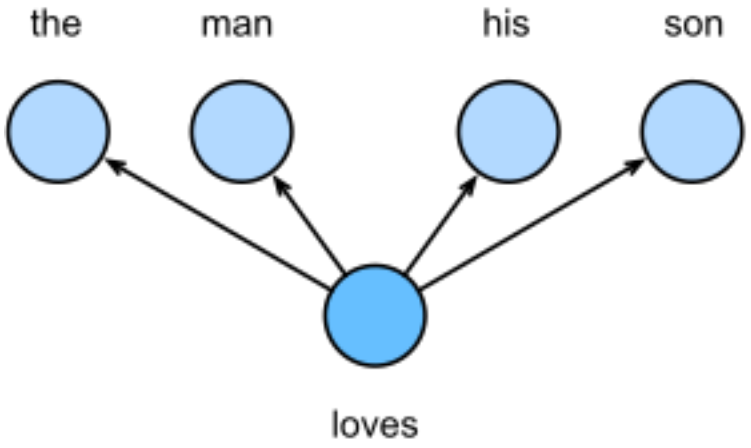


Figure 1: Skip-Gram model architecture, from Dive Into Deep Learning

According to Dive Into Deep Learning, for any word with index i in the directory, $\mathbf{v}_i \in \mathbb{R}^d$ and $\mathbf{u}_i \in \mathbb{R}^d$ are its two vector representations, where $\mathbf{v}_i$ is when the word is used as the "center word" and $\mathbf{u}_i$ is when the word is used as the "context word". If w_o is any context word and w_c is given center word, then conditional probability of generating w_o can be modelled as

$$P(w_o|w_c) = \frac{exp({\mathbf{u}_o}^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} exp({\mathbf{u}_i}^\top \mathbf{v}_c)}. \tag{3}$$

where $\mathcal{V}$ is the vocabulary index set. If a text sequence of length T is given and word at

time step t is denoted as w^t, then likelihood function of skip-gram model for context widow size m is as following:

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m, j \neq 0} P(w^{t+j}|w^t). \tag{4}$$

The skip-gram model parameters are center word and context word vector for each word in the corpus. In order to train skip-gram model, given loss function must be minimized:

$$-\sum_{t=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} logP(w^{(t+j)}|w^{(t)}).$$

Moreover, while using (stochastic) gradient descent to minimize the loss function, gradients of log conditional probability must be obtained. For center word w_c and context word w_o, log conditional probability is

$$logP(w_o|w_c) = \mathbf{u}_o^\top \mathbf{v}_c - log(\sum_{i \in \mathcal{V}} exp(\mathbf{u}_i^\top \mathbf{v}_c)). \tag{5}$$

And the gradient of center word $\mathbf{v}_c$ can be obtained as

$$\frac{\partial P(w_o|w_c)}{\partial \mathbf{v}_c} = \mathbf{u}_o - \sum_{j \in \mathcal{V}} P(w_o|w_c)\mathbf{u}_j. \tag{6}$$

After this calculations, Dive Into Deep Learning states that we obtain center word vector $\mathbf{v}_i$ and context word vector $\mathbf{u}_i$ for index i in the dictionary.

2.3 Continuous Bag of Words

The other method for word embedding in Word2Vec is continuous bag of words (CBOW). The main difference between skip-gram and CBOW is instead of generating surrounding words respect to center word, CBOW generates center word with the help of surrounding words. If the same example "the man loves his son" is taken for CBOW model, instead of generating surrounding words based on center word "loves", the model generates center word "loves" from its surroundings. This architecture can be seen in **Figure 2**.

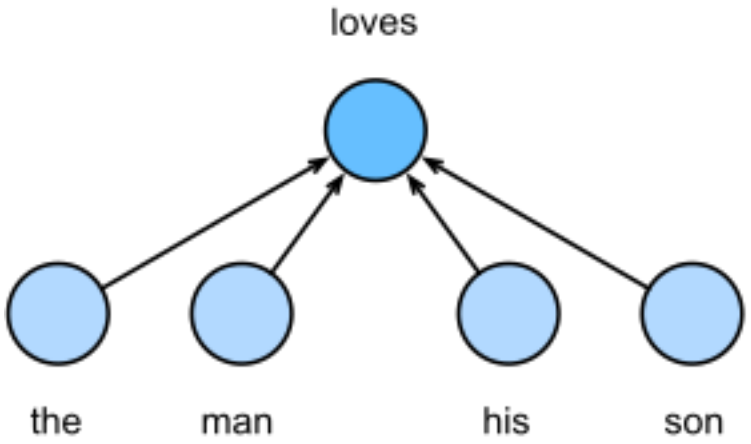


Figure 2: CBOW model architecture, from Dive Into Deep Learning

According to Dive Into Deep Learning (n.d.), in order to calculate conditional probability, context word vectors are averaged because there are multiple words. For any word with index i in the dictionary, $\mathbf{v}_i \in \mathbb{R}^d$ is the context word while $\mathbf{u}_i \in \mathbb{R}^d$ is the center word. It can be seen that meanings are switched compared to skip-gram model. While $w_o, ..., w_{o_{2m}}$ the conditional probability of generating center word w_c can be modelled as following:

$$P(w_c|w_o, ..., w_{o_{2m}}) = \frac{exp(\frac{1}{2m}u_c^\top(\mathbf{v}_{o_1} + ... + \mathbf{v}_{o_{2m}}))}{\sum_{i\in\mathcal{V}} exp(\frac{1}{2m}u_c^\top(\mathbf{v}_{o_1} + ... + \mathbf{v}_{o_{2m}}))}. \tag{7}$$

For simplicity, $\mathcal{W}_o = \{w_o, ..., w_{o_{2m}}\}$ and $\bar{\mathbf{v}}_o = (\mathbf{v}_{o_1} + ... + \mathbf{v}_{o_{2m}})/(2m)$. Then equation given above is simplified as following:

$$P(w_c|\mathcal{W}_o) = \frac{exp(u_c^\top \bar{\mathbf{v}}_o)}{\sum_{i\in\mathcal{V}} exp(u_i^\top \bar{\mathbf{v}}_o)}. \tag{8}$$

Furthermore, Dive Into Deep Learning (n.d.) states that if the length of a text sequence is T and the word at time t is $w^{(t)}$, then for context window of size m the likelihood function of CBOW model is

$$\prod_{t=1}^{T} P(w^{(t)}|w^{(t-m)}, ..., w^{(t-1)}, w^{(t+1)}, ..., w^{(t+m)}). \tag{9}$$

Due to CBOW and skip-gram models being similar, training them also almost same. According to Dive Into Deep Learning, to train CBOW model, maximum likelihood estimation of CBOW model is equal to minimization of following loss function

$$-\sum_{t=1}^{T} logP(w^{(t)}|w^{t-m)}, ..., w^{(t-1)}, w^{(t+1)}, ..., w^{(t+m)}). \tag{10}$$

where

$$logP(w_c|\mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - log(\sum_{i \in \mathcal{V}} exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)). \tag{11}$$

Through differentiation, gradient with respect to any cotext word vector $\mathbf{v}_{o_i}$ can be obtained as

$$\frac{\partial logP(w_c|\mathcal{W}_o)}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m}(\mathbf{u}_c - \sum_{j \in \mathcal{V}} P(w_j|\mathcal{W}_o \mathbf{u}_j)). \tag{12}$$

2.4 Global Vectors for Word Representation

Global Vectors for Word Representation (GloVe) is an unsupervised learning algorithm that developed by scientist led by Jeffery Pennington in Stanford University. Unlike Word2Vec, GloVe captures global contextual information of words by calculating a global word-word co-occurrence matrix. For example, in a large corpus, the word "liquid" more likely to co-occur with "water" than "ice", but word "solid" more likely to co-occur with "ice" than "steam". According to Agarwal, only local context of words is captured by Word2Vec (Agarwal, 2022). On the other hand, entire corpus is considered by GloVe and a large matrix that can capture co-occurrence of words within the corpus is created.

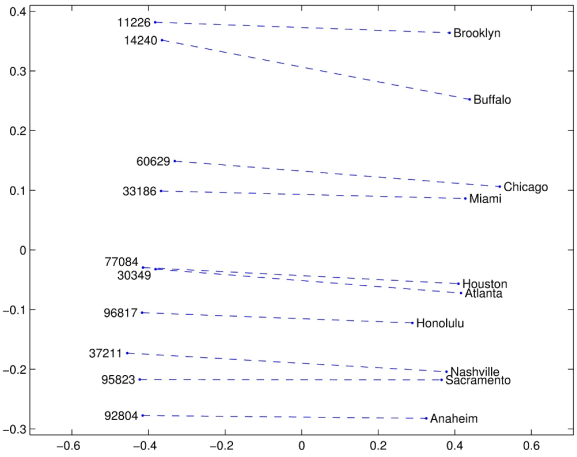


Figure 3: GloVe vectors capturing relation between city and zip code, from Stanford University

Agarwal continues by GloVe has the combination of the advantages of two-word vector learning methods: matrix factorization like latent semantic analysis (LSA) and local context window method (like Skip-Gram or CBOW). LSA is the technique that analyses the relationship between a set of documents and the terms they contain by using singular value decomposition.

Furthermore, the GloVe method's computational time is reduced by a rather simpler least square error function. As it is stated in Dive Into Deep Learning, Glove makes three changes to skip-gram model square loss (n.d.). where vectors $\mathbf{v}_i \in \mathbb{R}^d$ and $\mathbf{u}_i \in \mathbb{R}^d$ keep same representations as skip-gram model, those there changes are as following:

1. Using variables $p'_{ij} = x_{ij}$ and $q'_{ij} = exp(\mathbf{u}_j^\top \mathbf{v}_i)$ that are not probabilistic distributions. After taking their logarithms, the squared loss term becomes
$(log(p'_{ij}) - log(q'_{ij}))^2 = (\mathbf{u}_j^\top \mathbf{v}_i - log(x_{ij}))^2$.

2. For each word w_i, adding two scalar model parameters: the center word bias b_i and context word bias c_i.

3. Replacing the weight of each loss term $h(x_{ij}$ where $h(x)$ is increasing in the interval of $[0, 1]$.

Therefore, loss function of GloVe is:

$$\sum_{i\in\mathcal{V}}\sum_{j\in\mathcal{V}} h(x_{ij})(u_j^\top \mathbf{v}_i + b_i + c_j - log(x_{ij}))^2 \tag{13}$$

Where suggested choice for $h(x)$ is if $x < c$, then $h(x) = (x/c)^\alpha$, else $h(x) = 1$.

Finally, when compared to Word2Vec GloVe handles out of vocabulary words better. Due to that, GloVe performs better in word analogy and named entity recognition tasks.

2.5 Bidirectional Encoder Representations from Transformers

Created by researchers at Google in 2018, Bidirectional Encoder Representations from Transformers (BERT) is a family of masked-language models. Before discussing BERT; terms context-independent, context-sensitive, task-specific and task-agnostic must be discussed.

A context-independent function $f(x)$ only takes token x as its input. Due to the complex semantics and synonyms in natural languages, context-independent representations miss the meaning of the words in some cases. For example, the word "bank" can be used in both the sentence "I sat by the bank and enjoyed the view of the river." and the sentence I went to the bank to deposit some money". A context-independent algorithm might miss the difference between this cases.

On the other hand, context-sensitive function $f(x, c(x))$ is dependent to both token x and context $c(x)$. According to Dive Into Deep Learning, some of the most popular context sensitive representations are language-model-augmented sequence tagger (TagLM), Context Vectors (CoVe), and ELMo (Embeddings from Language Models).

When it comes to task-specific and task-agnostic representation, task-specific representation is when a model is optimized to a specific task, while a task-agnostic representation is when model is independent from a task based architecture. For instance, ELMo is an task-specific solution while it is not necessary to create specific architecture for each NLP task. The Generative Pre-Training (GPT) model is a context-sensitive, task-agnostic representation. However, according to Dive Into Deep Learning, GPT only

look left to right because of autoregressive nature of natural languages. For example, the sentences "A crane is flying." and "A crane is crashed." is taken, because of word "crane" being sensitive to context in its left, GPT will return the same representation of "crane". Both ELMo and GPT fails in some cases. According to Dive Into Deep Learning (n.d.), combination of both representations BERT, encodes context bidirectionally and requires minimal architecture changes for a wide range of natural language processing tasks. Differences between ELMo, GPT and BERT can be seen in **Figure 4**.

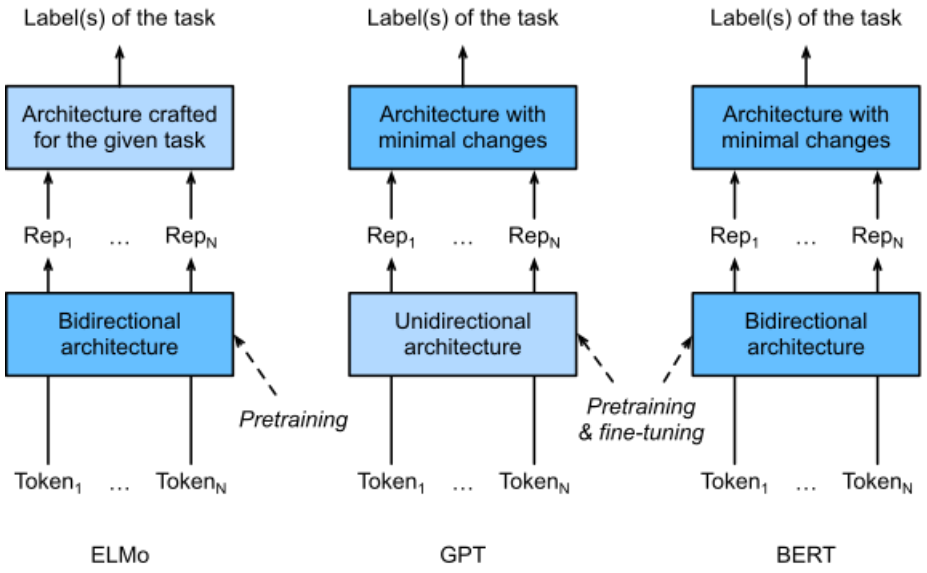


Figure 4: Difference between ELMo, GPT, and BERT, from Dive Into Deep Learning

Furthermore, Dive Into Deep Learning continues by using pretrained transformer encoder, any token based on its bidirectional context can be represented by BERT. Furthermore, BERT is similar to GPT in two aspect during supervised learning of downstream tasks.

1. BERT representations will be fed into an added output layer with minimal changes to the model architecture.

2. while the additional output layer will be trained from scratch, all the parameters of the

pretrained transformer encoder are fine-tuned.

Finally, Agarwal states that there are two variants of BERT: BERT-Base and BERT-Large. While BERT-Base has 110 million parameters, BERT-Large has 340 million parameters.

3 Word2Vec

Word2Vec is a word embedding model that was published by researchers led by Tomáš Mikolov at Google in 2013. As stated in Gensim documentations (n.d.), words are embeded in lower-dimensional vector space by Work2Vec. The method used to compress data into lower dimension is called autoencoding. After that, in this vector space, vectors which have similarity of context between them are close to each other while words that have different meanings are distant to each other. Word2Vec Model calculates the cosine similarity of those words in order to find relations between them. Due to that, the term autoencoders are discussed and it is followed by discussion of Word2Vec model in this section.

3.1 Autoencoders

In order to understand word embedding, the term autoencoders must be discussed. Autoencoders are unsupervised learning techniques that compress the input into lower dimension and then reconstruct output from this. Autoencoders are mainly used in tasks like anomaly detection, feature detection, facial recognition, and word embedding. Typical structure of an autoencoder can be seen in **Figure 5**.

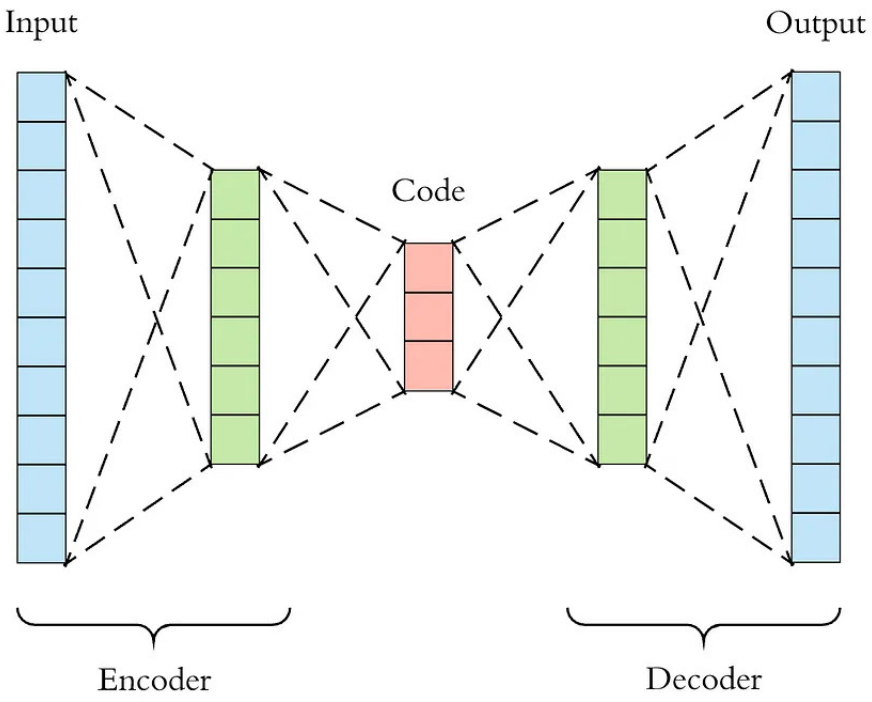


Figure 5: Typical autoencoding structure, from Towards Data Science

As it can be seen in Figure 1, an autoencoder reduces input into lower dimensional data by using neural networks, basically compressing them. After that, this reduced form code is used in mathematical operations and data model. Finally, autoencoder decodes the code to get an output with the same size as the input. There are several modifications of autoencoding like denoising autoencoders which applies random noise, sequence-to-sequence autoencoders which produces sequences of fixed sized vectors, or variational autoencoders.

3.2 Word2Vec

After obtaining vectors, Word2Vec uses cosine similarity metric to measure similarity of the words. Due to that, this architecture is similar to autoencoders, where there is encoder and decoder layers. If cosine value of two words is 0, then words do not hold similarity. If cosine value of two words is 1, then the words are overlapping. Due to that, Word2Vec model is mostly used in semantic analysis. On the other hand, the main disadvantage of

Word2Vec model is that it can not handle out of vocabulary words well. The words not presenting in training data is called out of vocabulary words. As Chandran stated (2020), a random vector representation is assigned for out of vocabulary words by Word2Vec, and they can be not optimal. Word2Vec constructs vectors with two methods: Skip-Gram and CBOW methods. Those two methods are discussed in previous chapters. Briefly, Skip-Gram generates surrounding words by center word, while CBOW does the opposite: generating the center word by surrounding words. In following application section, both Skip-Gram and CBOW models are used.

4 Application

In this section, Skip-Gram and CBOW are applied on The Divine Comedy by Dante Alighieri. Before applying models on The Divine Comedy, brief information about dataset is given and then dataset is prepared for the models.

4.1 Information About the Dataset

The Divine Comedy is an epic poem written by Italian poet Dante Alighieri in the 14th century. While it is considered one of the greatest works of literature, the poem is divided into three parts. Those three parts represent different realms of afterlife and consists of Inferno (Hell), Purgatorio (Purgatory), and Paradiso (Paradise).

The Divine Comedy is chosen for this paper due to several reasons. Firstly and most significantly, The Divine Comedy is a voluminous work that consists of wide range of characters, concepts, names. Due to its rich vocabulary, it provides great spectrum of words to use for model. Secondly, The Divine Comedy has a great cultural significance for both European and world literature. Due to this importance, it has countless adaptations in various languages and finding this adaptations is easier compared to other works of literature. Finally, The Divine Comedy explores fundamental aspects of human nature like sin, love, redemption. The concepts that explored in The Divine Comedy are mostly universal and due to that appear in different languages. Thanks to that, working on The Divine Comedy can show how a language reflects universal concepts.

For this paper, versions of The Divine Comedy in different languages are used and the data is taken from Project Gutenberg. Six different versions of The Divine Comedy in different languages are used and those languages are Dutch, English, Finnish, German, and Italian.

4.2 Application

Firstly, required libraries are imported. pandas library is used for data manipulation and analysis, while functions from nltk library are used for text processing: word_tokenize is used for tokenization, stopwords is used for removing stopwords, and WordNetLemmatizer is used for lemmatization. In following, gensim is used for Word2Vec model. matplotlib.pyplot is used for plotting. Also, cosine_similarity is used for calculating cosine similarity between two vectors while dendogram and linkage are used for hierarchical clustering.

In [ ]:
```
import pandas as pd
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import gensim
import gensim.downloader as api
from gensim.models import word2vec
from simalign import SentenceAligner
import matplotlib.pyplot as plt
from sklearn.metrics.pairwise import cosine_similarity
from scipy.cluster.hierarchy import dendrogram, linkage
```

After importing required libraries, data is imported. Data is acquired by Gutenberg Project as discussed in previous chapters and saved as txt files.

In [ ]:
```
Dutch = open('Dutch.txt').read()
English = open('English.txt').read()
Finnish = open('Finnish.txt').read()
German = open('German.txt').read()
Italian = open('Italian.txt').read()
languages = [Dutch, English, Finnish, German, Italian]
names = ['Dutch', 'English', 'Finnish', 'German', 'Italian']
```

This is followed by lemmatization and tokenization processes. Lemmatization is grouping together the inflected forms of words to analyze them as a single item, while tokenization is the process of breaking a stream of text up into words, phrases, symbols, or other meaningful elements called tokens. Tokenization is followed by removal of stopwords from the dataset. This processed datasets are saved as tokenized_"language_name" variables.

In [ ]:
```
def data_tokenizer(language, language_name, encoding = 'utf-8'):
    lemmatized = WordNetLemmatizer().lemmatize(language)
    tokenized = nltk.word_tokenize(lemmatized)
    f=[word.lower() for word in tokenized if word.isalpha()]
    stop_words = set(nltk.corpus.stopwords.words(language_name))
    [stopped] = [[i for i in j if i not in stop_words] for j in [f]]
```

```
    return stopped

tokenized_Dutch = data_tokenizer(Dutch,'Dutch')
tokenized_English = data_tokenizer(English,'English')
tokenized_Finnish = data_tokenizer(Finnish,'Finnish')
tokenized_German = data_tokenizer(German,'German')
tokenized_Italian = data_tokenizer(Italian,'Italian')
tokenized_names = 'tokenized_'+pd.Series(names)
tokenized_languages = [tokenized_Dutch, tokenized_English, tokenized_Finnish, tokenized
```

For more consistent models, only most common 50 words are selected and used for further analysis. This is done by using most_common_words function. This function takes a list and transforms it to a dataset, then counts the number of words and sorts them in descending order. Finally, it returns the most common 50 words.

In [ ]:
```
def most_common_words(lang):
    df = pd.DataFrame(lang, columns = ['Language'])
    df_sorted = df.groupby(['Language'])['Language'].count().reset_index(
                        name='Count').sort_values(['Count'], ascending=False)
    return df_sorted.Language[:50].reset_index(drop=True)

Dutch_most_common = most_common_words(tokenized_Dutch)
English_most_common = most_common_words(tokenized_English)
Finnish_most_common = most_common_words(tokenized_Finnish)
German_most_common = most_common_words(tokenized_German)
Italian_most_common = most_common_words(tokenized_Italian)
most_common_names = 'most_common_'+pd.Series(names)
most_common_languages = [Dutch_most_common, English_most_common, Finnish_most_common, G
most_common_words_ = pd.DataFrame(most_common_languages).T
most_common_words_.columns = most_common_names
```

Most common words can are as following:

In [ ]:
```
most_common_words_.head(10)
```

Out[ ]:

	most_common_Dutch	most_common_English	most_common_Finnish	most_common_German	most_co
0	den	thou	ma	sprach	
1	gij	one	mi	sah	
2	zoo	thee	mut	drum	
3	zóó	unto	näin	schon	
4	wanneer	upon	sa	mehr	
5	zeide	said	mun	wohl	
6	wij	thy	jo	ward	
7	waar	us	mulle	licht	
8	mijne	made	min	gleich	
9	oogen	eyes	kaikki	wer	

After that, most common 50 words are aligned between each other to be used in word2vec model. This is done by alingment function. This function takes a language and alings it with English. It returns aligned_"language_name" variables. While aligning, mwmf key is used because it has the best results.

In [ ]:
```
aligner =  SentenceAligner(model="bert", token_type="bpe", matching_methods="mai")
def alingment(language):
    aligned = aligner.get_word_aligns(English_most_common.to_list(), language.to_list()
    mwmf = pd.DataFrame(aligned['mwmf'])
    return language.reindex(mwmf[0]).reset_index(drop=True)
```

```
Some weights of the model checkpoint at bert-base-multilingual-cased were not used whe
n initializing BertModel: ['cls.predictions.decoder.weight', 'cls.predictions.transfor
m.dense.weight', 'cls.predictions.transform.LayerNorm.weight', 'cls.predictions.bias',
'cls.predictions.transform.LayerNorm.bias', 'cls.seq_relationship.bias', 'cls.seq_rela
tionship.weight', 'cls.predictions.transform.dense.bias']
- This IS expected if you are initializing BertModel from the checkpoint of a model tr
ained on another task or with another architecture (e.g. initializing a BertForSequenc
eClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertModel from the checkpoint of a mode
l that you expect to be exactly identical (initializing a BertForSequenceClassificatio
n model from a BertForSequenceClassification model).
2023-06-08 17:10:41,285 - simalign.simalign - INFO - Initialized the EmbeddingLoader w
ith model: bert-base-multilingual-cased
```

In [ ]:
```
Dutch_aligned = alingment(Dutch_most_common)
English_aligned = alingment(English_most_common)
Finnish_aligned = alingment(Finnish_most_common)
German_aligned = alingment(German_most_common)
Italian_aligned = alingment(Italian_most_common)
aligned_names = 'aligned_'+pd.Series(names)
aligned_languages = [Dutch_aligned, English_aligned, Finnish_aligned, German_aligned, I
Aligned_DataFrame = pd.DataFrame(aligned_languages).T
Aligned_DataFrame.columns = names
```

After aligning, most common 10 aligned words are as following:

In [ ]:
```
Aligned_DataFrame.head(10)
```

Out[ ]:

	Dutch	English	Finnish	German	Italian
0	den	thou	ma	sprach	ch
1	den	one	ma	sprach	ch
2	gij	thee	mi	sah	sì
3	zoo	unto	mut	drum	de
4	zóó	upon	näin	drum	d
5	wanneer	said	sa	schon	d
6	zeide	thy	mun	schon	s
7	wij	us	jo	mehr	quel
8	wij	made	jo	wohl	me
9	waar	eyes	mulle	ward	poi

This aligned words are going to be used in Word2Vec model. There will be 2 Word2Vec models for each language. One will be trained with Skip-Gram, while other will be trained with CBOW.

In [ ]:
```
def skipgram(language):
    return gensim.models.Word2Vec(language, vector_size = 50, sg = 1).wv
def cbow(language):
    return gensim.models.Word2Vec(language, vector_size = 50,sg = 0).wv

skipgram_Dutch = skipgram(Dutch_aligned)
skipgram_English = skipgram(English_aligned)
skipgram_Finnish = skipgram(Finnish_aligned)
skipgram_German = skipgram(German_aligned)
skipgram_Italian = skipgram(Italian_aligned)

cbow_Dutch = cbow(Dutch_aligned)
cbow_English = cbow(English_aligned)
cbow_Finnish = cbow(Finnish_aligned)
cbow_German = cbow(German_aligned)
cbow_Italian = cbow(Italian_aligned)
```

For using Word2Vec model in clustering, each word must be represented by a vector instead of a matrix. Due to that, following **flat()** function is for flattening the language matrices. This function takes the language matrix, and transforms it into a list. After that, it flattens the list and returns it.

In [ ]:
```
def flat(model):
    vocab = list(model.index_to_key)
    vectors = model[vocab]
    vectors_flatten = vectors.flatten()
    return vectors_flatten
```

In this part, Skip-Gram model will be used for clustering. Firstly, each language is flattened and saved as an array. After that, those vectors are combined as a dataframe named skipgram. Names of the languages is index of this dataset and and columns are corresponding vectors. NaN values are droppen to being able to use the dataframe in clustering.

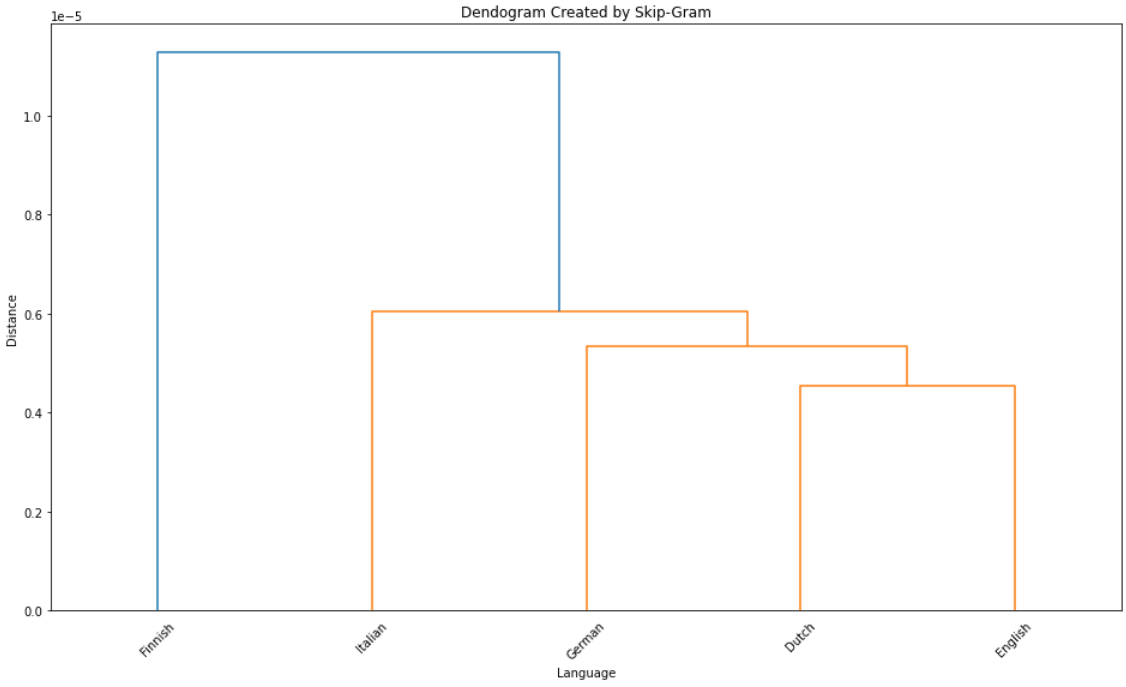
In [ ]:
```
flat_skipgram_Dutch = flat(skipgram_Dutch)
flat_skipgram_English = flat(skipgram_English)
flat_skipgram_Finnish = flat(skipgram_Finnish)
flat_skipgram_German = flat(skipgram_German)
flat_skipgram_Italian = flat(skipgram_Italian)
skipgram = pd.DataFrame([flat_skipgram_Dutch, flat_skipgram_English, flat_skipgram_Finr
```

After creating skipgram dataframe, cosine similarity is calculated for the dataset. This metric returns cosine value of angle between two vectors. If this cosine value is 1, it means that two vectors are identical. If it is 0, it means that two vectors are orthogonal. If it is -1, it means that two vectors

are opposite of each other. After calculating cosine similarity, linkage is used for hierarchical clustering. This linkage function takes cosine similarity as input and returns a linkage matrix. Linkage matrix is a matrix that contains information about hierarchical clustering. This is followed by plotting dendogram. Dendogram is a tree diagram that shows the arrangement of the clusters produced by hierarchical clustering. X label of the dendogram is the languages, while Y label is the distance between clusters. The dendogram can be seen below.

In [ ]:
```
skipgram_similarity = cosine_similarity(skipgram)
Z = linkage(skipgram_similarity, 'ward')
plt.figure(figsize=(16, 9))
dendrogram(Z, leaf_rotation=90, leaf_font_size=7., labels = skipgram.index)
plt.title('Dendogram Created by Skip-Gram')
plt.ylabel('Distance')
plt.xlabel('Language')
plt.xticks(rotation = 45, fontsize = 10)

plt.show()
```



As it can be seen in the dendogram, Finnish is clustered different from other 4 languages. This is caused by the fact that Finnish is not an Indo-European language. Finnish is an Uralic language, which is a language family that contains languages such as Hungarian and Estonian. Furthermore, Italian is also clustered different from other 3 languages. This is caused by the fact that Italian is a Romance language, which is a language family that contains languages such as Spanish and French.

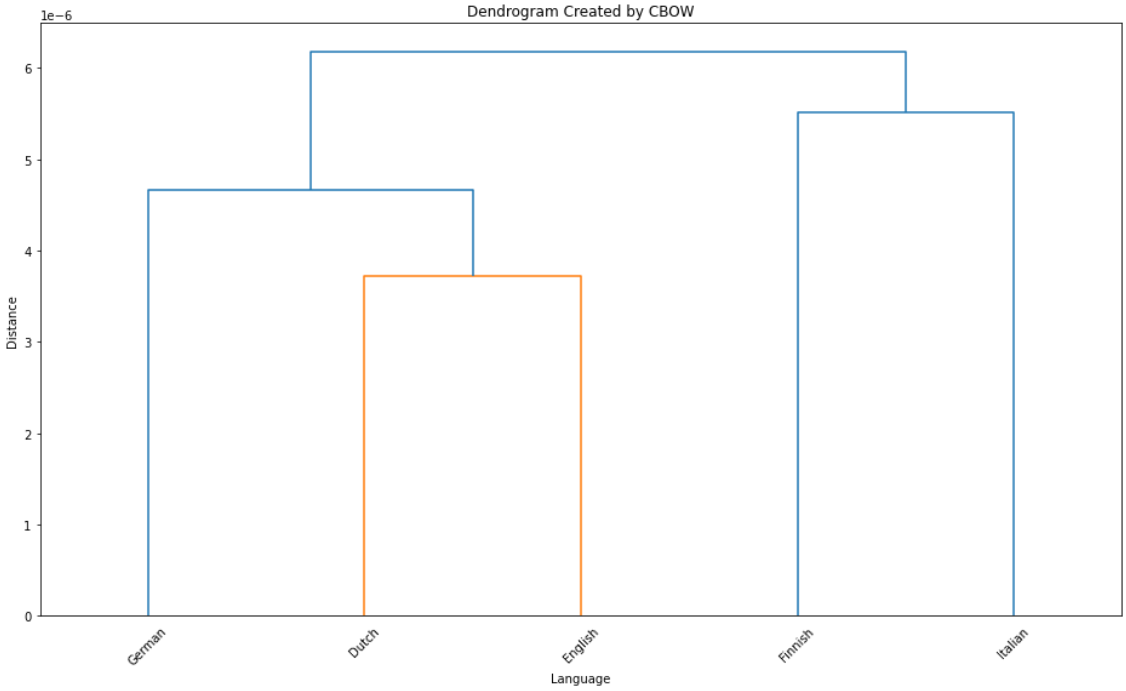
Finally, Dutch and English is clustered together instead of German. This is caused by the fact that Dutch and English are West Germanic languages, while German is a High Germanic language. Thanks to these results, it can be said that clustering by Skip-Gram algorithm is successful.

After Skip-Gram, CBOW model will be used for clustering. Firstly, each language is flattened and saved as an array. After that, those vectors are combined as a dataset named cbow. Names of the languages is index of this dataset and and columns are corresponding vectors. To use in clustering, NaN values are dropped.

In [ ]:
```
flat_cbow_Dutch = flat(cbow_Dutch)
flat_cbow_English = flat(cbow_English)
flat_cbow_Finnish = flat(cbow_Finnish)
flat_cbow_German = flat(cbow_German)
flat_cbow_Italian = flat(cbow_Italian)
cbow = pd.DataFrame([flat_cbow_Dutch, flat_cbow_English, flat_cbow_Finnish, flat_cbow_G
```

cbow dataframe is used to calculate cosine similarity with the function cosine_similarity. While this section of the code is same with the previous one, it is repeated to be able to compare results. After calculating cosine similarity, linkage is used for hierarchical clustering. This linkage function takes cosine similarity as input and returns a linkage matrix. This is followed by plotting dendogram. Dendogram is a tree diagram that shows the arrangement of the clusters produced by hierarchical clustering. Labels are same as Skip-Gram dendogram: X label of the dendogram is the languages, while Y label is the distance between clusters. The dendogram can be seen below.

In [ ]:
```
cbow_similarity = cosine_similarity(cbow)
Z_cbow = linkage(cbow_similarity, 'ward')
plt.figure(figsize=(16, 9))
dendrogram(Z_cbow, leaf_rotation=90, leaf_font_size=7., labels = cbow.index)
plt.title('Dendrogram Created by CBOW')
plt.ylabel('Distance')
plt.xlabel('Language')
plt.xticks(rotation = 45, fontsize = 10)
plt.show()
```



In this dendogram, it can be seen that Dutch and English are clustered close to each other and German is the closest language to them. This relationship caused by the same reason as Skip-Gram dendogram: While Dutch and English are West Germanic languages, German is a High Germanic language. Furthermore, instead of being clustered with other Indo-European languages, Italian is clustered with Uralic language Finnish in this dendogram. This mistake might be caused by morphological similarity between Italian and Finnish. Instead of generating surrounding words from center word, CBOW model generates center word from surrounding words, and that might cause to morphological similarity of those two languages to effect clustering. Finally, it can be seen that except Italian, the clustering is same as Skip-Gram dendogram. Thanks to these results, it can be said that clustering by CBOW algorithm partially successful.

5 Discussion

References

[1] Agarwal, N. (2022). *The Ultimate Guide To Different Word Embedding Techniques In NLP*. KDnuggets. https://www.kdnuggets.com/2021/11/guide-word-embedding-techniques-nlp.html

[2] Chandran, S. (2020). *Introduction to Text Representations for Language Processing — Part 2*. Towards Data Science. https://towardsdatascience.com/introduction-to-text-representations-for-language-processing-part-2-54fe6907868

[3] Dive Into Deep Learning. (n.d.). *The Skip Gram Model*. https://d2l.ai/chapter_natural-language-processing-pretraining/word2vec.html#the-skip-gram-model

[4] Dive Into Deep Learning. (n.d.). *The Continuous Bag of Words (CBOW) Model*. https://d2l.ai/chapter_natural-language-processing-pretraining/word2vec.html#the-continuous-bag-of-words-cbow-model

[5] Dive Into Deep Learning. (n.d.). *Bidirectional Encoder Representations from Transformers (BERT)*. https://d2l.ai/chapter_natural-language-processing-pretraining/bert.html#bidirectional-encoder-representations-from-transformers-bert

[6] Gensim. (n.d.). *Introducing: the Word2Vec Model*. https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html#sphx-glr-auto-examples-tutorials-run-word2vec-py

[7] Pennington, J. Socher R., Manning, C. D. (n.d.). *GloVe: Global Vectors for Word Representation*. https://nlp.stanford.edu/projects/glove/

[8] Kınık, D., Güran, A. (2021). TF-IDF ve Doc2Vec Tabanlı Türkçe Metin Sınıflandırma Sisteminin Başarım Değerinin Ardışık Kelime Grubu Tespiti ile Arttırılması, *Avrupa Bilim ve Teknoloji Dergisi*, 21, 323-332.