

Universidad de Alcalá

Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Artificial Intelligence based Model for Ancient Texts Analysis

Author: Sergio Hernández González

Advisor: Juan José Cuadrado Gallego

2024

UNIVERSIDAD DE ALCALÁ
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

Trabajo Fin de Grado

Artificial Intelligence based Model for Ancient Texts Analysis

Author: Sergio Hernández González

Advisor: Juan José Cuadrado Gallego

Tribunal:

President: _____

1st Vocal: _____

2nd Vocal: _____

Deposit date: June 9th, 2024

A mi familia y amigos. . .

“Yeah Science!”

Jesse Pinkman

Resumen

Este proyecto tiene como objetivo el estudio y aplicación de técnicas de procesamiento del lenguaje natural (NLP) para identificar segmentos de texto o frases relacionados en distintos lenguajes. El objetivo principal que se pretende cumplir con el desarrollo del proyecto es poder dar un primer paso hacia una herramienta que facilite el trabajo de investigación en este campo, permitiendo relacionar textos, o fragmentos de los mismos, escritos en distintos lenguajes de forma rápida.

Palabras clave: Aprendizaje automático, NLP, Lenguas antiguas.

Abstract

This project aims to study and apply state-of-the-art techniques and tools from the field of natural language processing (NLP) to the identification of related text segments or sentences written in different languages. The main goal to be achieved is taking a first step towards a tool that facilitates the research work in this field, allowing to relate texts, or fragments of them, written in different languages in a fast way.

Keywords: Machine learning, NLP, Ancient languages.

Contents

Resumen	VII
Abstract	IX
Contents	XI
List of Figures	XV
List of Tables	XVII
List of Acronyms	XIX
List of Symbols	XIX
1 Introduction	1
1.1 Studied techniques	2
1.2 Document structure	3
2 Theoretical Study	5
2.1 Introduction	5
2.2 Natural Language Processing	5
2.2.1 How do computers understand human language?	6
2.2.1.1 Character encoding	6
2.2.1.2 Label encoding	7
2.2.1.3 One-hot encoding	8
2.2.1.4 Word embeddings	10
2.2.1.5 Sentence embeddings	15
2.2.1.6 Conclusion	17
2.2.2 Semantic similarity	17
2.2.3 Text processing	19
2.2.3.1 Tokenization	19
2.2.3.2 Stopword Removal	20
2.2.3.3 Stemming and Lemmatization	20
2.2.3.3.1 Stemming	20
2.2.3.3.2 Lemmatization	21
2.2.3.4 Text Cleaning and Normalization	22
2.2.3.5 Regular expressions	22
2.3 Machine Learning	23
2.3.1 Different approaches	23
2.3.1.1 Supervised Learning	23

2.3.1.2	Unsupervised Learning	23
2.3.1.3	Semi-Supervised Learning	24
2.3.1.4	Reinforcement Learning	24
2.4	Word2Vec	24
2.4.1	Continuous Bag-of-Words Model	26
2.4.2	Continuous Skip-gram Model	27
2.4.3	Creation process	27
2.4.3.1	The fake task	27
2.4.3.2	Word2Vec Neural Network architecture	29
2.4.3.3	Training process	31
2.4.3.3.1	Training algorithm	31
2.4.3.3.2	Loss function	32
2.4.3.3.3	Activation function - softmax	33
2.4.3.3.4	Negative sampling	34
2.4.3.3.5	Subsampling of frequent words	35
2.4.4	Differences with GloVe	36
2.4.5	Differences with FastText	37
3	Experimentation	39
3.1	Tools used	39
3.1.1	Programming language	39
3.1.2	Package manager	40
3.1.3	Environment	41
3.1.4	Libraries	41
3.1.4.1	Numpy	41
3.1.4.2	NLTK	42
3.1.4.3	Gensim	42
3.2	Using pre-trained models	42
3.3	Building an English model	50
3.3.1	Obtaining data	51
3.3.2	Exploring data	51
3.3.3	Processing data	51
3.3.4	Training the model	54
3.4	Building a Middle High German model	56
3.4.1	Obtaining data	56
3.4.2	Exploring data	56
3.4.3	Processing data	57
3.4.4	Training the model	60
4	Results	63
5	Conclusion and future directions	65
5.1	Conclusions	66
5.1.1	Feasibility and Challenges	66
5.1.1.1	Feasibility Study	66
5.1.1.1.1	Training a Word Embedding Model on Middle High German Cor- pora	66
5.1.1.1.2	Collaboration with Professor Thomas Leek	67

5.1.1.2	Challenges	67
5.1.1.3	Conclusion	68
5.1.2	Technology and Methods	68
5.1.3	Impact and Applications	68
5.2	Future directions	68
5.2.1	Improvements	69
5.2.1.1	Addressing data scarcity	69
5.2.1.2	Using a different approach	70
5.2.2	New methods and technologies	71
5.2.2.1	Contextual embeddings	71
5.2.2.2	Vectorial spaces alignment	72
5.2.2.3	Sentence embeddings	72
5.2.3	Tools	73
5.2.3.1	CLTK	73
5.2.3.2	CADE	73
5.3	Final conclusion	73
	Bibliography	75
	Appendix A Tools and resources	79

List of Figures

2.1	ASCII table	6
2.2	One hot encoded vectors in a 3D space	9
2.3	2D analogy to embeddings	11
2.4	3D analogy to embeddings	12
2.5	Semantic relationships in 3D space (image from [4])	13
2.6	Similarity metrics for embeddings	18
2.7	Stemming vs lemmatization by Niraj Bhoi[11]	21
2.8	CBOW and Skip-gram architecture by Mikolov et al.[2]	26
2.9	Skip-gram neural network architecture in detail	29
2.10	Dense or fully connected layer diagram with 3 neurons in both dense and previous layers .	30
2.11	Cross entropy loss versus prediction accuracy (generated with matplotlib for python) . . .	33
2.12	Softmax example in and out	34
2.13	GloVe matrix factorization process (inspired by image at [23])	37
3.1	Import Gensim API	43
3.2	Download models from Gensim API	43
3.3	Vector for 'hello'	44
3.4	Vector type	45
3.5	Vector shape	45
3.6	Cosine similarity between 'dog' and 'cat'	46
3.7	Cosine similarity between 'dog' and 'code'	46
3.8	Cosine similarity between 'dog' and 'god'	46
3.9	Top 10 most similar words to 'computer'	47
3.10	2D embedding algebra	48
3.11	$V(\text{king}) - V(\text{man}) + V(\text{woman}) = V(\text{queen})$ (within Word2Vec based model's space) . . .	49
3.12	$V(\text{king}) - V(\text{man}) + V(\text{woman}) = V(\text{queen})$ (within GloVe based model's space)	49
3.13	$V(\text{king}) - V(\text{man}) + V(\text{woman}) = V(\text{queen})$ (within fastText based model's space)	50
3.14	Sample sentences from English dataset	51
3.15	Import NLTK and RE	52
3.16	Load stopwords	52
3.17	Load and test lemmatizer	52
3.18	Function to remove anything that is not a-z characters	52
3.19	Function to clean data	53
3.20	Extraction of sentences from text file	53
3.21	Sentence number 216.757 after reading from text file	53
3.22	Sentence number 216.757 after processing it	54
3.23	Sentence number 4.853.720 after processing all the data	54
3.24	Sentence number 4.853.720 in Wikipedia	54

3.25	Training function from Gensim library	54
3.26	Save the model to disk	55
3.27	Load a model from disk	55
3.28	Shape of the embedding, it is 200 as specified	55
3.29	Top 10 most similar words to 'efficiency'	55
3.30	Cosine similarity between 'computer' and 'programming' embeddings	56
3.31	Word that doesn't match from 'dog', 'cat', 'monkey' and 'car'	56
3.32	Diplomatic format fragment of corpora	56
3.33	Modern format fragment of corpora	57
3.34	Normalised fragment of corpora	57
3.35	Libraries used to extract and process Middle High German (MHG) text	58
3.36	Function that writes any number of sentences to a plain text file	58
3.37	Function that, given the contents in a page, removes first and last lines (metadata)	58
3.38	Function that processes each sentence removing special characters and extra spaces	58
3.39	Main function that uses all of the previous to extract, process, and save the data	59
3.40	Fragment of the text file containing all the processed sentences	60
3.41	Number of sentences used to train the MHG model	60
3.42	Cosine similarity between 'Christ' and 'got' within MHG custom model	60
4.1	Cosine similarity between 'hêrre' and 'got' within MHG custom model	64
4.2	Ten most similar words to 'keiser' within MHG custom model	64

List of Tables

2.1	Example of label encoded words in alphabetical order	7
2.2	Label encoding	8
2.3	One-hot encoding	9
2.4	GloVe co-occurrence matrix	36

List of Acronyms

AI	Artificial Intelligence.
ANN	Artificial Neural Network.
API	Application Programming Interface.
ASCII	American Standard Code for Information Interchange.
BERT	Bidirectional Encoder Representations from Transformers.
BPE	Byte-Pair Encoding.
CBOW	Continuous Bag-of-Words.
CNN	Convolutional Neural Network.
GPT	Generative Pre-trained Transformer.
LLM	Large Language Model.
MHG	Middle High German.
ML	Machine Learning.
NLP	Natural Language Processing.
NN	Neural Network.
OOV	Out-of-vocabulary.
POS	Part of speech.
RE	Regular Expression.
SGD	Stochastic gradient descent.

Chapter 1

Introduction

This document aims to take a first step towards a tool that facilitates the identification of related text fragments from texts written in different ancient languages.

This project arises as a collaboration with the University of Wisconsin-Stevens Point^[1], specifically with Professor and researcher Thomas Leek. The idea that gave rise to this project was Thomas Leek's proposal to develop an application that would facilitate the task of relating fragments of texts written in ancient languages to researchers like himself. Such an application would allow researchers and linguists to find relationships between text fragments written in different ancient languages in a fast and easy way. These relationships could result in ancient translations or adaptations being found.

With the idea of working in this direction, this project arises. This project aims to take a first step towards the creation of the described tool. This first step will include the preliminary study and the creation of a plan for the continuation of the work.

As mentioned above the objective is to lay a solid foundation on which to continue to work. Therefore this project aims to study the feasibility of developing the described application, the knowledge, materials or tools needed to carry it out, and finally the impact it could have if it was built.

The core functionality of said tool is the capability of evaluating the relationship between two given fragments of text written in two different ancient languages. When we talk about evaluating a relationship between two pieces of text, we just mean evaluating if those pieces of text are related in any way, making it possible for one to be a translation or adaptation of the other. This could be summarized as evaluating semantic similarity between two pieces of text.

Evaluation of semantic similarity is a task that has a great variety of applications such as sentiment analysis or plagiarism detection. Since this is such a useful task there is a lot of related work, but a vast majority of it was done using English data, and most of the remaining used other modern languages such as Spanish, German, Chinese, etc. However there are few people working on applying these state-of-the-art technologies to ancient languages and this can be due to mainly two circumstances:

The first one is ancient languages being very low resource languages, this is because there are no more people generating data everyday, as there are for modern languages (social media, books, online posts, ...), this makes applying state-of-the-art Machine Learning (ML) techniques considerably less interesting, since they work better with more data.

The second one is that modern languages have much clearer commercial applications, this makes ancient languages less interesting for companies and developers around the world.

As a consequence of the above this project aims to study how these technologies are applied to modern languages to later be able to create a guideline on how to transfer said knowledge to the study and research of ancient languages and lay a mainly theoretical foundation on how the initial proposition could be carried out.

Therefore the first step to be taken will be to have a solid knowledge on what technologies, tools and techniques are needed or used nowadays to perform tasks similar to the one we want to achieve.

1.1. Studied techniques

As we mentioned we will study many different techniques in order to understand how are Natural Language Processing (NLP) tasks performed nowadays, and to have a comprehensive understanding on the panorama. We will now get a preview of these techniques that will be later studied in depth.

We will start by understanding how do computers understand human or natural language, this is not trivial since machines are unable to work directly with plain text, they are only capable of working with numbers, more specifically with a binary system that uses 0's (zeros) and 1's (ones), and even more specifically with high or low voltages. While trying to understand how are modern machines able to work so well with human language we will cover different methods used in the computer science field to help machines understand our text, this is achieved by having a way to represent text as numbers which then can be converted to a language that machines can manipulate with ease.

Among these methods of representation we will find character encoding with ASCII or Unicode, the most used, basic and intuitive of them which consists on the assignation of a numeric value to each character we want to be able to represent, for example using ASCII code we would give the character 'a' a value of 97. This way, any given sequence of characters (words, or text in general) can be converted to a sequence of numeric values that the machine can handle, store in memory or operate with.

The second method of representation is label encoding, and it is closer to what Large Language Model (LLM)s use to handle not only the representation of words (the sequence of characters that shape the word) but it's meaning. The idea behind this is to assign these numeric values to words instead of characters, since in human languages words are most of the time the smallest unit of meaning we can find. This way a complete vocabulary or language could be represented as a list of words, like a dictionary, where each one of these words is assigned a value that will later be able to represent the word and therefore it's meaning.

When we try to apply machine learning algorithms to human language, and more specifically to the semantic plane of the languages, we hit a wall here. Since machine learning algorithms are constantly trying to find patterns within the data we provide to them, using label encoding will not be very useful, this is because by assigning random (or ordered from a non semantic perspective such as alphabetically) we would be introducing information that was initially not there. For example when assigning close values to words we are implicitly saying they are close, and if we are trying to work with their meaning, we would be saying their meaning is close, or similar.

This is why One-hot encoding and word embeddings are introduced to allow us to work with words meaning. One-hot encoding is a method that consists of encoding every element in a set of size equal to N elements, as a sparse vector, consisting of N-1 zeros and a one, in a specific position that will never be repeated by any other elements, for example the word number 1 in our label encoded vocabulary would become a vector of size N with a one in the first position and then filled with zeros. This way the issue of introducing unwanted data is solved since all vectors are at the same distance to each other.

Word embedding take this coding method one step further, and they have set a new standard in the [NLP](#) field by allowing us to obtain representations of words that actually represent their respective meaning with dense vectors. This means two related words' embedding would be closer to each other than two unrelated ones, and it creates a whole new paradigm where we could even perform mathematical operations with these embeddings, and therefore with words, for example we could add, subtract or compare words based on their semantics. This is a very interesting technique that we will later use to develop some experimentation and that will help us to start working on the goal of the entire project.

Later we will study Word2Vec which the most well known implementation of these word embeddings, developed by google researchers in 2013 [\[2\]](#) [\[3\]](#), it did set a new standard in the industry. It consists of a Artificial Neural Network (ANN) architecture and utilized different machine learning algorithms or techniques to learn these word embeddings. We will also cover with less detail some other implementations of this concept, that introduce some improvements or differences and that are used in many applications such as GloVe and fastText.

Many text processing (or pre-processing) techniques will also be studied in order to be able to understand how text can be processed in order to improve performance of all [NLP](#) applications, these include tokenization (which consist of dividing the text into smaller fragments that we want to process such as words), stopword removal (which removes words that are not useful or don't provide enough semantic information to be considered), stemming and lemmatization (which reduces the number of different words by grouping variations of words as the same base word) or even text normalization (which allows us to know what text format to expect, removing special characters, numbers or lowercasing all the text).

1.2. Document structure

This document is structured into different chapters, each one of them serves a different purpose. There are five chapters.

The first one is the introduction where we already introduced the research topic and provided background information necessary for understanding the context of the study. We also talked about the goals of the project and it's origin, and provided a preview of what will be studied in the following sections of the document.

The next chapter is the "Theoretical Study," which studies the theoretical framework of the research. It begins with an introduction to [NLP](#) theoretical concepts and then moves into a detailed exploration of the state-of-the-art techniques that are used within this field. This part covers how computers understand human language, discussing various encoding methods such as character encoding, label encoding, one-hot encoding, word embeddings, and sentence embeddings. Each method is explained in detail to provide a comprehensive understanding of their roles in [NLP](#). The section also covers various text processing techniques, including tokenization, stopword removal, stemming and lemmatization, text cleaning and normalization and regular expressions.

Following this, the section on Machine Learning (inside the theoretical framework study) introduces different approaches to [ML](#), such as supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning. The theoretical study concludes with a detailed examination of Word2Vec, including its continuous bag-of-words model and continuous skip-gram model. The creation process of Word2Vec is explained, covering it's [ANN](#) architecture, training process, and various optimization methods. The differences between Word2Vec and other models like GloVe and FastText are also highlighted.

The third chapter is called "Experimentation" and it is the practical aspect of the research. It starts with an overview of the tools used, including the programming language, package manager, environment,

and various libraries such as Numpy, NLTK, and Gensim. This section also details the coded implementations of word embeddings. For word embeddings, it discusses the use of pre-trained models and the process of building a model, which includes obtaining, exploring, and processing data, and training the model.

The document then presents the "Results" section, where the outcomes of the experiments are analyzed and discussed. This section interprets the findings made during the realization of the project.

Finally, the "Conclusion and Future Directions" section summarizes the main findings and their implications. It discusses the feasibility and challenges of the research, including a feasibility study on training a word embedding model on Middle High German corpora and collaboration with Professor Thomas Leek. The section also outlines the limitations of the study, its impact and applications, and suggests directions for future research. This includes potential improvements, addressing data scarcity, exploring new algorithms, methods, and technologies, and discussing tools like CLTK and CADE.

The document concludes with a "Bibliography," listing all the sources cited in the thesis, and an "Appendix" providing additional tools and resources that were relevant to the research.

The previously mentioned sections are accompanied by detailed explanatory figures. These appear mostly in the second section "Theoretical Study" to help understand concepts in a more visual way, or to enhance the readability and ease of understanding of different ideas. Many of them are handmade by me, but some others were found during the research phase, either on articles, papers, books, or websites. All of the non self-made figures' sources have been properly cited.

Chapter 2

Theoretical Study

2.1. Introduction

In this chapter, we will lay the foundations of [NLP](#), exploring the methodologies, algorithms, and techniques that enable machines to process language, understand grammar, and even decipher context and nuances. Understanding these basic concepts prepares the ground for understanding how semantic similarity is studied.

Next, we will dive into the scope of semantic similarity. We will discover the tools and techniques that allow computers to measure the proximity or relationship between two texts in terms of their meaning.

In summary, this chapter serves as a comprehensive guide to elucidate the theoretical landscape of semantic similarity within [NLP](#), covering its theoretical foundations, methodologies, and fundamental importance in advancing natural language understanding and processing systems.

2.2. Natural Language Processing

In this section, we will study in detail how machines process and understand text.

Today, it is difficult to separate [NLP](#) from Artificial Intelligence (AI) or, more specifically, [ML](#). This is because in recent years, [ML](#) has provided incredibly powerful tools to the field of [NLP](#). All the latest [LLM](#) such as Bidirectional Encoder Representations from Transformers (BERT) or Generative Pre-trained Transformer (GPT) are built on [ANN](#) architectures that are trained using [ML](#) techniques, such as supervised and unsupervised learning. These [ML](#) models learn from large amounts of linguistic data to understand language, perform tasks such as automatic translation, sentiment analysis, text generation, and many other [NLP](#) applications. While [ML](#) is a powerful tool for addressing [NLP](#) problems, there are approaches in the field of [NLP](#) that do not directly depend on machine learning, such as heuristic linguistic rules or knowledge-based systems. However, at the forefront of current [NLP](#) development, [ML](#) has been central to achieving significant advances in the understanding and generation of human language.

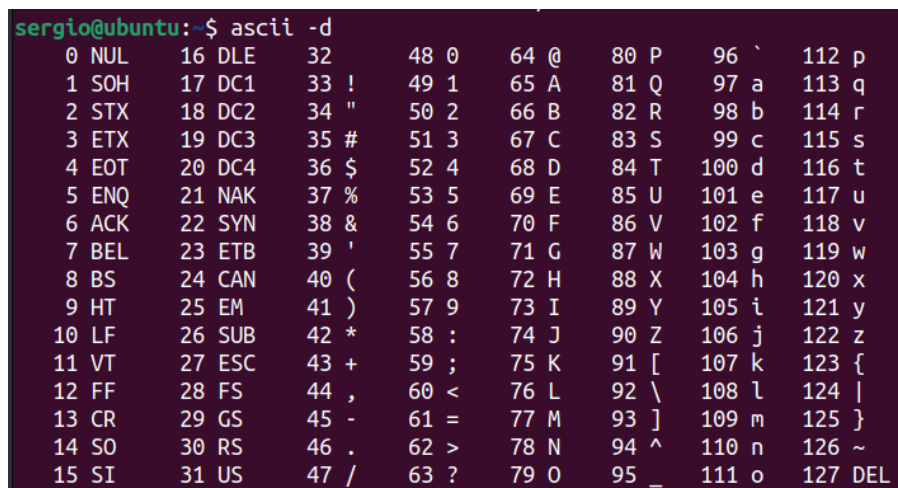
Next, we will explain the tools or techniques that are used today in the field of [NLP](#) to enable machines to interact with human language. As previously explained, we will focus solely on written text as it is the only relevant format for the development of the described work.

2.2.1. How do computers understand human language?

The first issue that arises when thinking about computers interacting with human language is understanding how they can handle characters, words, or phrases in natural language, as we know that computers are only capable of "understanding" numbers. Next, we will review the main ways of representing text through numbers that exist and are used in the field of [NLP](#).

2.2.1.1. Character encoding

There is a code called American Standard Code for Information Interchange (ASCII) that was first published in 1967 and can be used by practically all computer systems today to represent characters or symbols from numbers. The original version of [ASCII](#) is a 7-bit code that serves as a direct translation method between letters or symbols and numbers by assigning each value (between 0 and 127 since it is a 7-bit code as already mentioned) a character or symbol. Thus, in [ASCII](#) code, the letter 'a' is equivalent to the number 97 in the decimal system or 0110 0001 in binary, or the symbol '@' to 64 or 0100 0000.



```
sergio@ubuntu:~$ ascii -d
```

0	NUL	16	DLE	32		48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	\$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	
13	CR	29	GS	45	-	61	=	77	M	93]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Figure 2.1: ASCII table

Other alternatives to [ASCII](#) have appeared over time, such as extended [ASCII](#) which allows the representation of more characters by using 8-bit codes instead of 7-bit, and the universally accepted Unicode which allows the encoding of characters from several writing systems, including Latin, Cyrillic, Greek, Arabic, Chinese, Japanese, Korean, and many others instead of only English characters.

This approach of encoding characters one by one is very useful when the problem to be solved is how to receive text from the user or how to display it on the screen. However, when it comes to "understanding" the text, it is not very effective. People do not assign meaning to letters, but rather, with them, we construct words to which we do assign meaning. In other words, when we want to perform operations with text on the semantic level, having assigned a value to each letter is not useful, since it would make no sense to identify the meaning of a text fragment by summing the values of all its characters or by the number of times certain letters or symbols appear. For example, the words "earth" and "heart" share all the characters but do not share meaning, while "cat" and "feline" share a great part of their meaning but their representation is completely different.

In summary, we must separate the representation of the text from its meaning if we want a computer to be able to work on the semantic level of natural language.

2.2.1.2. Label encoding

Therefore, the next idea that arises is to apply the same system but this time assigning values to the words within a given vocabulary, instead of assigning them to each letter or symbol that may appear in our text. For example, if we have a vocabulary with 1000 words, we would start numbering them from 0 to 999 one by one, thus assigning a unique value to each word that could identify its meaning. This approach is much more similar to what is used today in systems like LLMs, and it also resembles how we humans think when trying to understand the meaning of a text: it is the words that convey the vast majority of the meaning.

word	index
animal	0
apple	1
...	...
movie	499
museum	500
...	...
zebra	998
zygote	999

Table 2.1: Example of label encoded words in alphabetical order

However, this is not the approach used today in LLMs and, as we will see below, for good reason.

The ultimate goal of representing written text in human language in a format that computers can assimilate is (besides being able to represent it and take it as input without having to interpret it as we have seen with ASCII code, and Unicode) that they can process the text, perform operations, "understand" its meaning or learn from it (we will see this later).

As mentioned earlier, a large number of ML techniques are currently used in the field of NLP, and one of the most used structures (if not the most) are artificial neural networks. Neural networks are computational models that process information (usually in large quantities) to learn patterns in that information and perform specific tasks such as classification or prediction. Part of the information these models ingest may be in numerical form. For example, a bank might try to train a model that takes input data from millions of previous customers with the goal of assigning a probability value to whether a new customer will repay a loan. Part of the relevant information given to the model about previous customers would be, for example, their age or income, and the model may (or may not, a priori we don't know exactly what patterns the model identifies) understand that a person who is 27 years old is younger than someone who is 73, or that a person with a monthly income of €1500 has half the purchasing power of a person with a monthly income of €3000. The important thing is not whether the model ends up assigning a higher probability to younger or older clients, with more or less income, or whether these parameters ultimately have no relevance to the probability of a client repaying a loan, but that these are figures that make sense in the real world and, therefore, there may be patterns that can be learned and prove useful for performing certain tasks.

Language models take natural language text as input and learn to perform various tasks. But a model will not be able to learn patterns that may be useful if the data it receives makes no real sense. When we assign numeric values in the form of integers to the words found in a given vocabulary (which could be a more or less extensive part of the Spanish language vocabulary for this case), we are creating new data, and with this data, we may give information that is not real, causing ML models to be trained on information that lacks real meaning, thus nullifying their effectiveness and their main strength, the search for patterns. Let's study this with an example: If we return to the small vocabulary of 1000 words

that appear in a set of texts, we could have assigned the possible values to the words in the vocabulary in various ways such as order of appearance, alphabetical order, a totally arbitrary order... In our small example vocabulary it could happen that if we have ordered it alphabetically, we find, for example, the case of table 2.1

If a language model we want to train with information from this set of texts received the information represented based on our vocabulary, it could identify patterns in that information and draw conclusions such as the following:

- The word "animal" has very little value, whereas the word "zygote" has great value.
- The words "zebra" and "zygote" are very similar to each other, but "animal" is very different from both.
- The value of "zebra" is double that of "movie"
- A "museum" is approximately a midpoint between an "animal" and a "zygote"

This is just an example and does not occur exactly like this in reality, but the important thing is to understand that by representing text by assigning an arbitrary value to words, we would be creating new information and causing relationships to be identified that could not be identified with the original information. It is true that some words have a much closer relationship than others, for example in this case the word "animal" and the word "zebra" are more related to each other than either of them to the word "museum", but it seems impossible to capture all the relationships we know between words by merely ordering them on a numerical scale. If we have a small vocabulary, we could try to organize it manually by placing related words close to each other, but as this vocabulary grows, and it doesn't need to grow much, it will become impossible to represent the reality of human language this way because too many variables come into play. Do we group verbs together because they all represent an action, or do we place them around words frequently used with that verb? As explained, it is impossible to represent reality through language on a numerical scale.

2.2.1.3. One-hot encoding

To solve this problem, a word representation system is introduced that starts from the same base, assigning a value to each word found in a vocabulary to uniquely identify it but without introducing false information as a side effect. It is a system that eliminates the possibility of implying that "zebra" is much more similar to "zygote" than to "animal." This is where One-hot encoding is introduced to the field of NLP. One-hot is a way of encoding values through a set of bits among which only one has the value 1 and the rest have the value 0, thus in a group of three elements numbered from 0 to 2 we could have the following example:

In table 2.2 we can visualize the encoding proposed in the previous example, a nominal scale in which each number is used as a label to uniquely identify an object or element, but the value of the number itself has no real meaning.

word	index
animal	0
movie	1
zebra	2

Table 2.2: Label encoding

In table 2.3, we can visualize the One-hot encoding, whereby we assign a vector of n elements where n is the size of our vocabulary or set of words to be encoded. In this vector, each position represents a dimension and only one position will have a value of 1, while all others will have a value of 0.

animal	movie	zebra
1	0	0
0	1	0
0	0	1

Table 2.3: One-hot encoding

This way, in a hypothetical n -dimensional space, the distance between a pair of words will be the same for any pair in the entire set since each word occupies its own unique dimension, meaning each word in our vocabulary is completely independent of the others. As a measure of distance or, rather, similarity between two vectors, cosine similarity is used, a measure that uses the cosine of the angle formed by the vectors and is widely used in the field of data science. This will be explained in more detail later, but the main idea is that since the angle between any pair of vectors in this set is 90° , the cosine similarity will always be zero. A visual representation of the previous example can be seen in figure 2.2

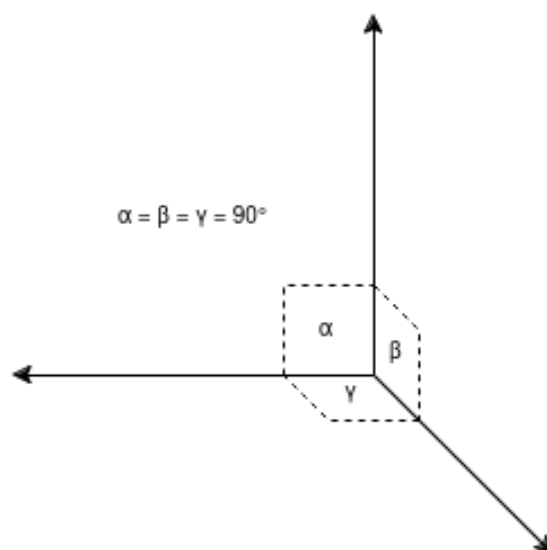


Figure 2.2: One hot encoded vectors in a 3D space

We have gone from a single integer value per word to a One-hot vector. Thus, avoiding introducing incorrect information when encoding text as numerical values. In this way, if we recover the previous vocabulary of 1000 words, each word would be a vector of length 1000, of which 99.9% are zeros, meaning they are empty. This seems like a great waste of memory for a computer. The one-hot encoding can be useful for representing words or phrases when the available vocabulary is relatively small since, although a large majority of the data is empty, if the total amount of information that the computer needs to handle is small, this may not be too much of a problem on machines as powerful as the current ones. However, it should be noted that a vocabulary of only 1000 words is really small when you want to make a machine capable of interacting with human language. If we look at the Oxford English Dictionary, considered one of the most prestigious in the English language, we can see that it contains about 600,000 definitions for approximately 273,000 words. That is, if we wanted to use one-hot vectors to encode the entire English language, each word would be a vector of length 273,000 in which more than 99.999% of the values are zeros. This system is not at all efficient in terms of memory.

2.2.1.4. Word embeddings

It is here where a way of representing text that solves this problem appears, embeddings. Embeddings are a numerical representation of text (in this case words) in the form of a vector with a previously determined length. That is, we could create embeddings for our vocabulary of 1000 words in the form of vectors of length 200, for example, and in the same way, we could encode the entire vocabulary of the English language with its 273,000 words in this same 200-dimensional space. The main idea is to compress the one-hot vectors to a more manageable size, that is, to reduce their dimensionality to a specific number of dimensions selected beforehand. The way this dimensionality reduction operation is carried out is by using neural networks. This is why embeddings are the first meeting point between [NLP](#) and [ML](#) that we will study. Embeddings are the way of representing text used by [LLMs](#) like ChatGPT today (not exactly the kind of embeddings we will see but the foundation is the same); it is a very powerful tool that provides a lot of advantages over other alternatives, such as one-hot encoding. Next, we will study in detail how the encoding works from embeddings, how they are created, and what they contribute to the field of [NLP](#), to understand how to use them and why they are so relevant today.

First, and having a general idea of what an embedding is, we will study what they contribute and why they are such a powerful tool when we try to represent text in a format understandable for a computer. An embedding is nothing more than a dense vector (unlike a one-hot sparse vector, a dense vector does not contain a large number of zeros), a representation with reduced dimensionality of a word. By having a lower dimensionality, the first advantage we find when using embeddings is clear, as we have already seen, these vectors are less costly for a machine in terms of memory, but this is not what makes embeddings so powerful, there is a feature that provides great value compared to other representations.

When we have studied one-hot encoding applied to the representation of human text in numerical format, we have seen that the main advantage it represents is that each word is encoded as a vector geometrically independent of the rest of the words in the vector space formed by the complete vocabulary. However, this characteristic is also the main difference between embeddings and one-hot encoding, and the reason why embeddings are today the most advanced and complete method of representing text in natural language in numerical format. Previously, it has been explained that it is impossible to capture the characteristics and relationships of words in human language by assigning an integer to each word or, visualizing it in a vector space, it is impossible to represent the characteristics and relationships of each word in a one-dimensional space. However, we could get much closer to the reality of how we understand language if we worked with a higher number of dimensions. This is the characteristic of embeddings that makes them stand out and place them as the main method for representing natural language so that machines can process it with a semantic approach.

To visualize how an embedding can capture different language features by having multiple dimensions, we can make an analogy. For this example, let's select a vocabulary of four words (boy, girl, man, and woman) and let's represent it in a two-dimensional vector space, that is, let's represent it based on two characteristics, the goal is to represent reality as faithfully as possible considering only the characteristics denoted by each axis in this space. The most intuitive in this case would be to place gender on one axis and age on the other, in this way, we would obtain the vector space represented in [Figure 2.3](#).

As we see, it is a representation that resembles reality and allows us to obtain more information about each word than if we had assigned an integer value to each one, or if we had encoded them as one-hot vectors. In this case, we can represent the chosen vocabulary in just two dimensions because it is a specific and reduced example, but we can check that by increasing the vocabulary we could find representations in the form of n-dimensional embeddings that resemble the reality described by the words. For example, we can expand the previous vocabulary by doubling the number of words (adding prince, princess, king,

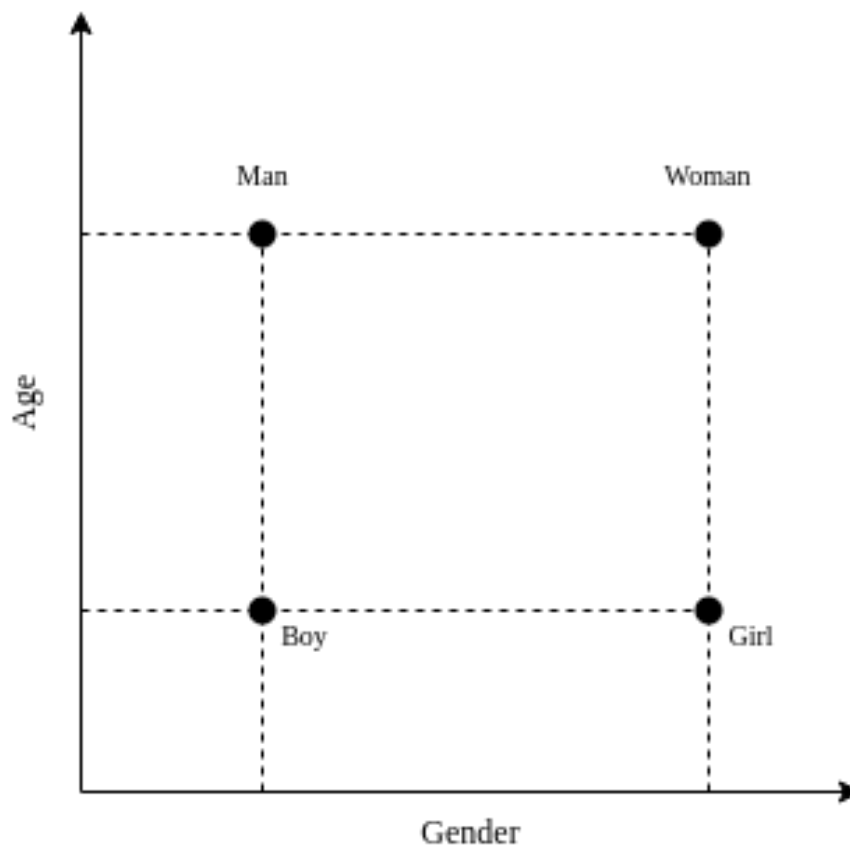


Figure 2.3: 2D analogy to embeddings

and queen) and add to our vector space a third dimension that quantifies a new characteristic, royalty. In this case, we would obtain the following representation of our vocabulary seen in figure 2.4.

Although humans can only visualize three dimensions, that does not mean that we cannot continue adding dimensions to our space and words to our vocabulary and get a more complex example. But since it is only an example and we know that humans are unable to visualize a fourth dimension, it will not be necessary.

This is the main idea of embeddings; as we will see later, it is something more complex since it would be impossible to do this manually for each word in the vocabulary, assigning a vector in relation to all the other vectors that we already know. This is why neural networks are used to perform this vector creation process; next, we will see the procedure followed and study the implications this has.

As mentioned earlier, an embedding is nothing more than a dense vector, the result of compressing or reducing the dimensionality of a sparse vector such as a one-hot vector. We also know that this dimensionality reduction is carried out with the help of neural networks, and there are two reasons why it is a good idea to rely on these structures from the ML field to perform this task.

First, neural networks are very good at compacting vectors. In fact, every time the information entered into a neural network is transmitted from one layer to the next, the dimensionality of a vector is changed, that is, if in a certain neural network we have a size in the input layer of 10 neurons and in the output layer of 2 neurons, when introducing information and receiving the output values, we have reduced a vector of size 10 to a vector of size 2.

A very clear example of this case is found in the field of computer vision when we use a Convolutional Neural Network (CNN) to classify images (although classification is not the only task where a dimension-

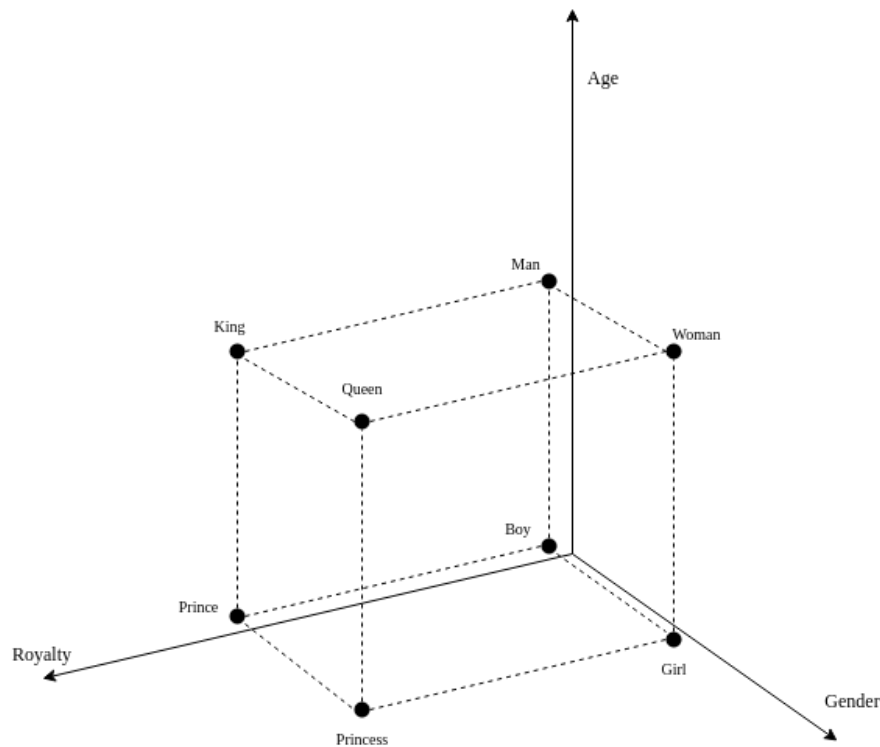


Figure 2.4: 3D analogy to embeddings

ality reduction of the input data occurs), there is continuous vector compaction in the neural network since the input will be a vector of large dimensions as it is a complete image and the output will probably be of smaller size since it will be a vector that indicates the belonging of the image to a class among a limited number of them. For example, if the task to be performed is the classification of images from the MNIST¹ dataset, the input data will be the values of each pixel of the image, that is, a vector containing 784 values (the images in this dataset have dimensions of 28x28 pixels) and the output will be a vector of dimension 10, a value to indicate the belonging to each possible class, that is, each digit from 0 to 9. This is a reduction of the dimensionality of a vector; the information represented by the input and output data is equivalent for the proposed task, but the size of this information has changed. Therefore, neural networks are capable of reducing the dimensionality of the information represented in the form of vectors.

But even more relevant is the ability of [ANNs](#) to learn. And it is that we have already talked about how difficult it would be to generate these dense vectors called embeddings manually since we should build a multidimensional vector space within which we are able to capture relationships between words through vector representations for each of them. The solution that has been reached is to let an [ML](#) model learn, through unsupervised learning techniques, to represent all the words in our vocabulary in this vector space. This implies that this model will receive as input a large amount of text written in natural language and from this, it will learn to represent each word as a vector with a previously defined dimensionality. The model responsible for generating the embeddings takes as input one-hot vectors of length n where n is the size of the vocabulary to be represented and returns as output vectors of length m where m is the desired size previously defined, and is determined by the number of neurons found in the output layer of the neural network responsible for generating the embeddings. If we were to generate bidimensional or tridimensional embeddings, we could represent them as in Figures 2.3 and 2.4, but it is

¹The MNIST database is an extensive collection of images of handwritten digits ranging from 0 to 9, which is widely used for training various image processing systems.

important to understand that, unlike in the previous examples, the dimensions do not have to correspond to any real-world attribute, in fact, we are unable to know what each dimension represents when we train a model to perform this task.

In this way, we could try to generate an embedding for each word in a certain language, for example, English. The result we will obtain after reducing the dimensionality of the one-hot vectors that represent each word of the English language and representing the new dense vectors of dimension n in an n -dimensional vector space is a space in which the semantic plane of that language is represented. That is, a space in which we could search for the vector used to represent a word and see that the words represented by the nearby vectors are words with a similar meaning to the initial word. In such a space, the vectors used to represent the words "animal" and "zebra" will be much closer to each other than any of them to the vector used to represent "movie". But this is not all, it has been shown that this way of representing language is capable of capturing semantic relationships between words, some examples are represented in a three-dimensional space in the figure 2.5.

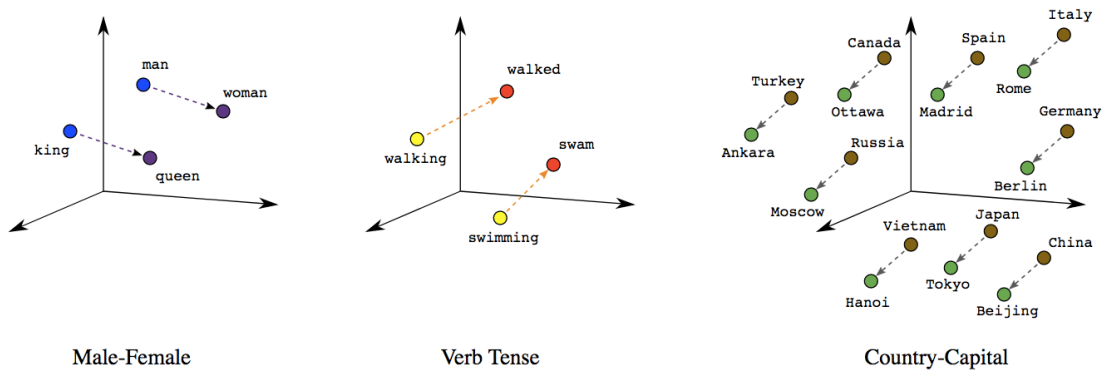


Figure 2.5: Semantic relationships in 3D space (image from [4])

This feature is incredibly powerful and allows implementing different algebras in this space. This means that in this vector space, we are able to perform operations with vectors, that is, operations with the meanings of the words. This presents a new way of understanding the relationship between machines and human language, allowing operations like the following:

$$\vec{V}(\text{king}) - \vec{V}(\text{man}) + \vec{V}(\text{woman}) = \vec{V}(\text{queen})$$

Where $\vec{V}(x)$ is the dense n -dimensional vector that represents the word x .

We have discussed the possibilities offered by embeddings as a way of representing human language; now, we will study in detail how these vectors are constructed to have a deeper understanding of their behavior. We know that these embeddings are generated by a neural network from one-hot vectors that are geometrically independent representations of words in a certain vocabulary. We also know that the model is trained in an unsupervised manner on a large amount of text (we will call it a corpus), but how can the model understand the relationships between words if it does not know their meaning? That is, how does it know that the relationship between "king" and "queen" is similar to the relationship between "man" and "woman" if these characteristics do not appear explicitly in the corpus on which it has been trained? Well, just like humans when we do not know the meaning of a word or when we are learning

a new language, the model is able to approximate the meaning of words by the context in which they appear. If the corpus used to train the model is large enough, it will be able to find semantic relationships that provide a representation of language that describes the reality reflected in the text that makes up the corpus. The process of learning these representation from context will be studied in a following section 2.4.3.3. For now we just need to understand the idea.

We have talked about how training an embedding over a corpus of texts can provide a vectorial space that represents the reality on that set of texts. However vast amounts of data (texts) is needed to accomplish a robust representation. This would mean that any development that involves the use state-of-the-art NLP techniques would come with a substantial workload as it requires gathering an extensive amount of information to generate proper embeddings. This is the reason that pre-trained embedding models such as *Word2Vec*[2][5], *GloVe*[6] or *fastText*[7] have been incredibly popular since their respective releases, being *Word2Vec* the first to be released and by far the most popular, used and studied by many for years. They are trained over huge amounts of data by researchers at big companies, experts in their field that are known for their incredible work, and they accomplish astonishing performances.

The current trend is to use these pre-trained embedding models that represent the entire vocabulary of a language. This brings a ton of advantages if we compare using pre-trained models to building your own, we will discuss these advantages:

- **Data Efficiency:** Training word embeddings from scratch often requires substantial amounts of labeled and unlabeled data. Pre-trained embeddings, having been trained on diverse and extensive corpora, alleviate the need for vast datasets, making them more suitable for scenarios with limited data.
- **Computational Efficiency:** Training word embeddings from scratch is a computationally expensive process, especially when dealing with large corpora and complex neural network architectures. Pre-trained embeddings, on the other hand, are already trained on extensive datasets using powerful hardware, saving considerable time and computational resources.
- **Transfer Learning:** Pre-trained embeddings capture general linguistic patterns and semantic relationships from vast amounts of text data. By leveraging pre-trained embeddings, models can benefit from this knowledge and adapt it to specific tasks with much smaller datasets. This concept, known as transfer learning, allows models to generalize better to new tasks and domains.
- **Fine-tuning:** Pre-trained embeddings provide a strong starting point for NLP models even when they will continue to be trained. By initializing models with pre-trained embeddings, they often achieve better performance in the early training stages compared to models starting with random embeddings. This is a specific case of transfer learning where the training of the exact same model is continued. We can think of this as being easier to reach a higher performance at working with texts in a specific domain for someone that already can use language in a general way than for someone with absolutely zero knowledge on language. This makes fine-tuning (training over a pre-trained model) with pre-trained embeddings a good alternative to training an embedding from scratch even for very specific tasks.
- **Ease of use:** Pre-trained embeddings reduce the need for extensive computational resources and infrastructure as we said, but also there is no need of knowing how to build an embedding model. This is particularly advantageous for researchers, developers, and organizations with limited time, resources and knowledge and bring NLP one step closer for anyone trying to introduce themselves in the area.

- **Community Standardization:** Pre-trained embeddings, such as Word2Vec, GloVe and fastText, have become standard benchmarks in the NLP community. This standardization facilitates collaboration, code sharing, and the comparison of model performance across different tasks.

With all of the advantages mentioned it looks like using a pre-trained embedding over training specific embeddings will always result in a better experience, but it is not always the case, there are some very specific tasks where training an specific word embedding might result in better representation of the text. While pre-trained word embeddings offer many advantages, there are also some disadvantages associated with using them compared to training your own embeddings. Here are several considerations:

- **Task-Specific Optimization:** Pre-trained embeddings are not optimized for a specific task. As we mentioned earlier in this section fine-tuning embeddings on a task-specific dataset can lead to better performance for that specific task. However when training your own embeddings, you have the flexibility to optimize them for your particular application from the first stages of the model building process, potentially improving the model's ability to capture relevant features in extremely specific tasks.
- **Vocabulary Mismatch:** Pre-trained embeddings are often generated with a fixed vocabulary size based on the training data. If your text contains out-of-vocabulary words that were not present in the training data, you might need to handle them separately (e.g., through subword embeddings²) or use unknown token³ representations. Training your own embeddings allows you to adapt to the specific vocabulary of your dataset.
- **Dependency on training algorithm:** Pre-trained embeddings are dependent on the pre-training algorithm used. Different algorithms make different assumptions about language and may capture different linguistic properties. If the assumptions made by the pre-trained embeddings do not align with the characteristics of your task, training your own embeddings might be more suitable. As all of the above this is not a common case but can happen within some very specific tasks.
- **Maintenance Over Time:** Pre-trained embeddings may become outdated as language evolves and new linguistic patterns emerge. Periodic updates or re-training might be necessary to ensure that the embeddings remain relevant and effective. When you train your own embeddings, you have more control over when and how you update them.

In summary, while pre-trained word embeddings offer a convenient and effective solution for many [NLP](#) tasks, there are scenarios where training your own embeddings provides advantages in terms of domain specificity, vocabulary adaptation, and task optimization. The choice between pre-trained and custom embeddings depends on the nature of your data, the requirements of your task, and the computational resources available. But, in general, whenever the task that the embedding is needed for is not too specific (very specific vocabulary or structures not usually found in real-world text), pre-trained embeddings are a great tool that saves time, work and usually provides better results.

2.2.1.5. Sentence embeddings

We are almost there, we already understand how [LLMs](#) understand human words nowadays, using embeddings, numerical representations of the meaning of a piece of text. And I said "piece of text"

²Using subwords is a technique that has been used in many pre-trained embedding models such as fastText, it is based on the aggregation of words that share a great part of their meaning, such as verbal forms ("helps", "helped", and "helping" are inflected forms of the same word "help") or the same verbs or singular-plural pairs of words ("dog" and "dogs"). Subword embedding may improve the quality of representations of rare words and out-of-dictionary words[7].

³Unknown token ('UNK') is a special token that is put in place of any word that is unknown for the model

on purpose, words are not the only structures that humans use to communicate, we use words with a determined meaning to build sentences and be able to communicate more complex ideas. And this is the reasoning behind the creation of sentence embeddings, the need for computers to be able to represent the meaning of not only words but complete sentences.

Building on the foundational concept of word embeddings, sentence embeddings aim to capture the semantic meaning of entire sentences, enabling a range of applications in the field of [NLP](#). In this exploration, we will delve into the basics of sentence embeddings, their methodologies, and their crucial role in determining sentence similarity.

Sentence embeddings, as for word embeddings, refer to numerical representations of sentences that encapsulate their semantic content. Unlike word embeddings, which represent individual words, sentence embeddings are designed to encode the meaning of entire sentences into fixed-size vectors. The primary purpose of sentence embeddings is to provide a condensed and meaningful representation of sentences, facilitating mathematical operations, comparisons, and various related [NLP](#) tasks.

The process of generating sentence embeddings is a critical aspect of natural language processing, influencing the performance of various downstream tasks. One prevalent approach involves leveraging pre-trained word embeddings, such as Word2Vec, GloVe or fastText, to represent individual words within a sentence and aggregate all of them to build the sentence embedding. These embeddings, obtained from extensive training on large text corpora, encapsulate semantic relationships between words. However, the straightforward aggregation of these word embeddings, (typically through averaging or summation) to create a sentence-level representation has its limitations. This method may not fully capture the contextual nuances present in the sentence, as it treats each word as an isolated entity without considering the sequential or syntactic relationships. However this word embedding aggregation approach has demonstrated a suprisingly good performance in different tasks for how simple it is.

In response to this limitation, a more sophisticated set of methods has emerged, centering around specialized models explicitly designed for sentence embeddings. These embedding models are often grouped as contextual embeddings. Unlike traditional methods that treat words in isolation, contextual embeddings consider the entire sentence, capturing the dynamic interplay of words and their meanings in different contexts. Leading the pack are models like [BERT](#), [GPT](#), and the Universal Sentence Encoder. These models represent a paradigm shift by considering the entire sentence context during the embedding generation process. Unlike traditional methods, they capture the intricate relationships between words within the sentence, offering context-aware embeddings that reflect the semantic meaning of the text. These more sophisticated methods have a much more complex implementation, but the concept of representing the meaning of the text via dense vectors remains the same.

[BERT](#), for instance, employs a bidirectional architecture to consider both left and right contexts of each word, allowing it to understand the meaning of a word in relation to its entire sentence. [GPT](#), on the other hand, is a generative model that predicts the next word in a sequence based on the context of preceding words, thus inherently learning the contextual nuances of the language.

The advantage of these specialized models lies in their ability to grasp the contextual intricacies and semantic nuances present in natural language. By considering the entire sentence during the embedding generation process, these models offer richer and more contextually informed representations. This not only enhances the quality of embeddings for downstream tasks such as sentiment analysis, text classification, and machine translation but also aligns with the evolving understanding of language as a dynamic and context-dependent system.

2.2.1.6. Conclusion

In conclusion, embeddings serve as transformative tools in natural language processing, providing a numerical representation of textual elements such as words and sentences. These representations, whether pre-trained or tailored to specific tasks, capture the semantic essence of language, enabling computational systems to understand and process textual information. By encoding complex linguistic structures into fixed-size vectors, embeddings facilitate efficient mathematical operations, semantic similarity assessments, and the development of sophisticated models for various NLP applications. Whether through traditional word embeddings like Word2Vec and GloVe or advanced contextual embeddings from models like BERT and GPT, the adoption of embeddings has revolutionized the way we approach language understanding. As NLP continues to evolve, embeddings remain a cornerstone, bridging the gap between human language and machine comprehension, and paving the way for continued advancements in the field.

2.2.2. Semantic similarity

NLP stands at the forefront of artificial intelligence, enabling machines to comprehend, interpret, and generate human language. Among the myriad tasks within NLP, measuring semantic similarity serves as a cornerstone for various applications such as information retrieval, question answering, machine translation, and text summarization. Semantic similarity, a fundamental concept within NLP, aims to quantify the likeness or relatedness between texts based on their meanings rather than their surface forms.

As we have already discussed, we are capable of obtaining numerical representations of the text based on their meaning called embeddings. These representations are dense multidimensional vectors. In mathematics and physics we are used to perform different operations over vectors, such as multiplication, addition or subtraction, embeddings are vectors, and as such we can also perform operations on them. In the end the whole point of making computers capable of representing the meaning of human texts is to allow these machines to perform all kind of tasks based on it, such as generation, summarization, etc. One of the main and simplest operations we can perform with numerical representations of text is comparison, and this is the base of the semantic similarity, one of the main subfields within NLP.

We can summarize semantic similarity as measuring how different two embeddings are, this is because, as we know already know, embeddings represent the semantics of some piece of text so by matematically calculating how different or similar these vectors are we can obtain a numerical value to quantify semantic similarity. Being able to quantify the relatedness of two pieces of text brings lots of applications, some of which we have already mentioned, but how can we calculate this similarity? Well, there are a lot of metrics that can be used to quantify the similarity of two vectors, we are going to study three of them which are very commonly used in this specific field, these are euclidean distance, dot product and cosine similarity.

- **Euclidean distance:** Euclidean distance measures the straight-line distance between two points in Euclidean space. For two vectors \vec{A} and \vec{B} in an n -dimensional space, the Euclidean distance d is calculated as:

$$d(\vec{A}, \vec{B}) = \sqrt{\sum_{i=1}^n (\vec{A}_i - \vec{B}_i)^2}$$

This metric gives the geometric distance between two points in space. Smaller distances indicate greater similarity, and larger distances indicate dissimilarity.

- **Dot product:** The dot product of two vectors \vec{A} and \vec{B} is the sum of the product of their corresponding components, the dot product is calculated as:

$$\vec{A} \cdot \vec{B} = \sum_{i=1}^n \vec{A}_i \cdot \vec{B}_i$$

The dot product reflects how much the vectors point in the same direction. It is the product of the magnitudes of the vectors and the cosine of the angle between them.

- **Cosine similarity:** Cosine similarity measures the cosine of the angle between two vectors. For vectors \vec{A} and \vec{B} , the cosine similarity $\cos(\theta)$ is calculated as:

$$\cos(\theta) = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\| \cdot \|\vec{B}\|}$$

Where $\|\vec{A}\|$ represents the Euclidean norm (magnitude) of vector A .

Cosine similarity disregards the magnitude of the vectors and focuses on the direction. It would be equivalent to dot product on a normalized space. It ranges from -1 (completely dissimilar, 180-degree angle) to 1 (completely similar, 0-degree angle). 0 indicates orthogonality.

Now let's visualize these measures to have a visual idea of what they geometrically represent:

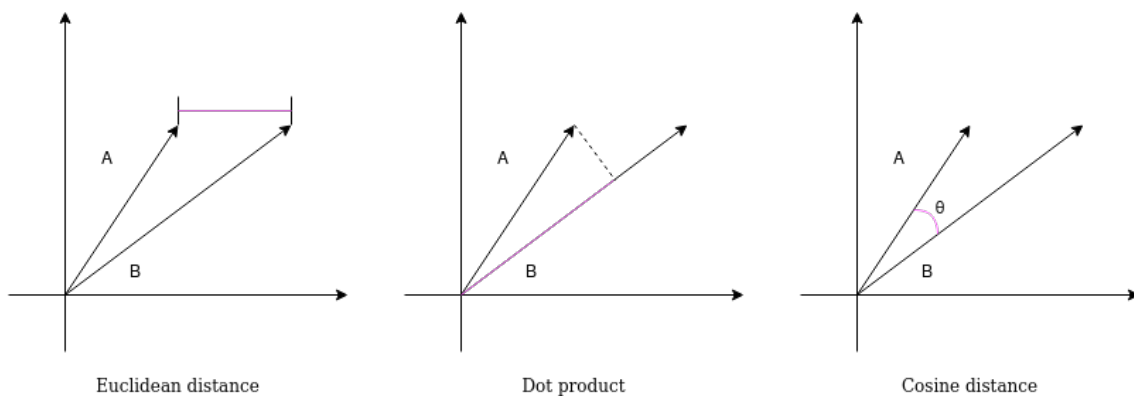


Figure 2.6: Similarity metrics for embeddings

When comparing embeddings the most used metrics are these, but cosine similarity is the most widespread among them. This is due to this metric being scale-invariant, this means cosine similarity will remain the same for two vectors regardless of their magnitude or length, meaning that it remains unchanged even if the vectors are scaled by a constant factor. This is important when dealing with embeddings that might have different scales due to the underlying algorithms used for their generation. While the dot product and euclidean distance are valid measures of similarity, their sensitivity to vector magnitude make them less suitable for tasks where the scale of embeddings might vary. Cosine similarity, with its normalization and focus on direction, addresses these concerns and has proven to be a reliable choice in many [NLP](#) and machine learning applications.

Cosine similarity has become a standard metric in many [NLP](#) tasks, including document similarity, information retrieval, and word embeddings evaluation. Its widespread use has established it as a common and accepted measure in the [NLP](#) community.

The ultimate goal of sentence similarity is to capture semantic similarity rather than mere lexical resemblance. Models like [BERT](#) and [GPT](#), designed with attention to contextual embeddings, excel in

understanding the nuanced meanings of words and phrases within a given context. However, capturing semantic similarity remains challenging due to the intricacies of natural language, contextual dependencies, and the diversity of expressions.

2.2.3. Text processing

In real world we can find text from many different sources with different formats or structures, when working with text it can be very useful to pre-process the raw data to achieve better results, this is called text processing and we will study different techniques and methods used in this field.

As we mentioned text processing is one of the fundamental steps in [NLP](#), where raw text data is transformed into a more structured format to improve the results in several aspects. In this section we will study this sub-field in depth with the goal of understanding how text processing is performed. This means we will study why and how all different approaches are used, and also what tools can be utilized to apply these techniques to text.

This is a more technical section since we can understand virtually any concept in the field of [NLP](#) without studying text processing. However, if we want to be able to work with text later, understanding all these methods will be vital to achieve satisfactory results.

We will now start by studying the main text processing techniques from the simplest of them to the most advanced ones, and will end the section with a research on the tools used to apply all of the above.

2.2.3.1. Tokenization

Tokenization is the process of decomposing a text into individual words or tokens, serving as fundamental units for any Natural Language Processing task. For instance, the sentence "I love natural language processing" can be tokenized into the following tokens: ["I", "love", "natural", "language", "processing"]. Tokenization can be achieved through simple space-based splitting or more sophisticated techniques such as regular expressions, facilitating text analysis by providing a structured text representation.

The method of tokenization can vary based on the language, task, and text complexity. Common tokenization techniques include:

- **Whitespace Tokenization:** The simplest form of tokenization, where text is split based on whitespace characters (e.g., spaces, tabs). For example, "Tokenization is important" is tokenized into ["Tokenization", "is", "important"].
- **Punctuation-Based Tokenization:** This approach splits text on both spaces and punctuation marks. For example, "I like apples, oranges, and bananas" is tokenized into ["I", "like", "apples", ",", "oranges", ",", "and", "bananas"].
- **Subword Tokenization:** Techniques like Byte-Pair Encoding (BPE) and WordPiece are employed for languages with complex morphology, such as German or agglutinative languages. These methods split text into subword units, enabling the model to handle word variations more effectively. More on [BPE](#) and Wordpiece in their respective publication papers [8], [9] (BPE) and [10] (WordPiece).
- **Language-Specific Tokenization:** Different writing systems and rules necessitate specialized tokenization. For instance, Chinese text might be tokenized into individual characters, while Japanese text might use a combination of characters and specific rules for splitting text.
- **Custom Tokenization Rules:** Specific NLP tasks may require custom tokenization rules to handle domain-specific text, such as URLs, email addresses, or code snippets.

2.2.3.2. Stopword Removal

Stopwords are common words in a language (e.g., "the," "and," "is") that do not carry significant meaning and are often removed during text processing. The removal of stopwords reduces the dimensionality of the text data, focusing the analysis on more meaningful words. The primary reasons for removing stopwords include:

- **Reducing Noise:** Stopwords can clutter data and add noise to the analysis. Removing them helps focus on the words that matter.
- **Improved Efficiency:** Cleaner datasets are easier and faster to process, enhancing the efficiency of [NLP](#) algorithms.
- **Enhanced Accuracy:** Stopword removal can improve results in tasks like sentiment analysis or topic modeling, where stopwords might skew interpretation.

For example, in the sentence "The quick brown fox jumps over the lazy dog," removing the stopwords results in "quick brown fox jumps lazy dog," where the remaining words carry more semantic weight. Although the list of stopwords can vary by language and context, their removal is a universal practice in [NLP](#), enhancing the quality and relevance of textual data and improving [NLP](#) model performance.

2.2.3.3. Stemming and Lemmatization

Stemming and lemmatization are essential preprocessing techniques in Natural Language Processing used to transform words into their root forms. This process is crucial for various [NLP](#) applications, including search engines, text mining, information retrieval, and machine learning. Both techniques aim to reduce the complexity of text data and improve the performance of [NLP](#) models by simplifying the vocabulary. Despite their common goal, stemming and lemmatization differ significantly in methodology and output.

2.2.3.3.1. Stemming is a heuristic process that reduces words to their base or root form by removing suffixes and prefixes. The primary objective of stemming is to map related words to a common base form, even if this form is not a valid word. Stemming algorithms apply a set of rules to strip affixes, and these rules are generally language-specific. Some examples can be:

- "connected" -> "connect"
- "connecting" -> "connect"
- "connection" -> "connect"
- "connections" -> "connect"

But also:

- "defensible" -> "defens"
- "defense" -> "defens"
- "defensively" -> "defens"

Or:

- "running" -> "run"
- "ran" -> "ran" (irregular form not affected by stemming)

Where we can notice its limitations, such as resulting words not necessarily being valid words ("defens") or some verb tenses being reduced to infinitive and some of them not ("ran").

2.2.3.3.2. Lemmatization is a more sophisticated process that reduces words to their base or dictionary form, known as the lemma. Unlike stemming, lemmatization considers the morphological analysis of the words and their context within a sentence. Lemmatization requires understanding the Part of speech (POS) of a word and may involve more complex algorithms and linguistic databases. Some examples can be:

- "defensively" -> "defensive" (considering its POS being an adverb derived from an adjective)
- "better" -> "good" (considering the context of comparison)
- "running" -> "run" (considering it as a verb in the present participle form)
- "mice" -> "mouse" (considering it as a plural noun)

We can easily visualize a comparison between these two approaches with an example in the figure 2.7.

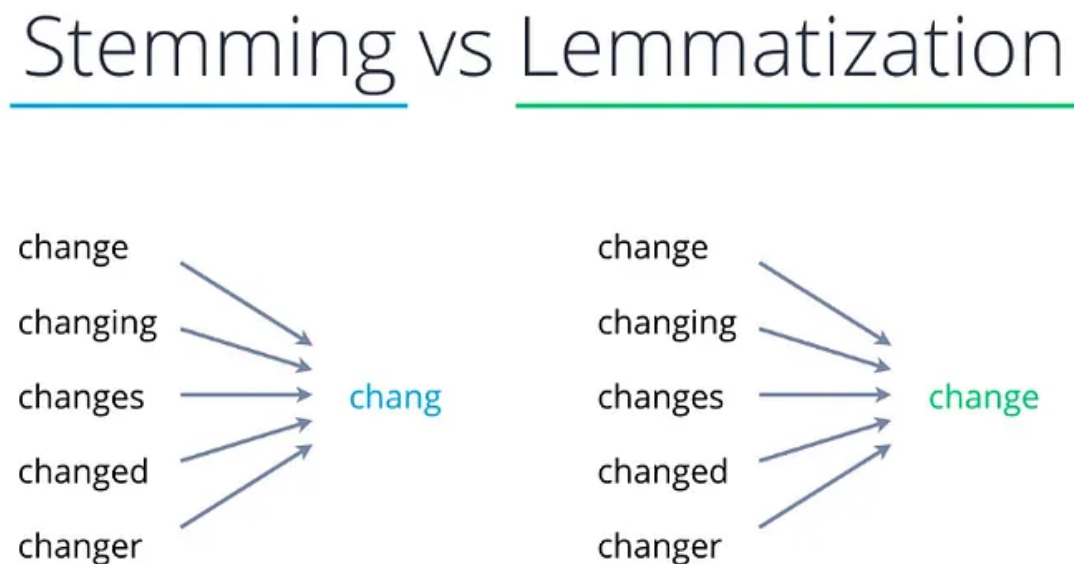


Figure 2.7: Stemming vs lemmatization by Niraj Bhoi[11]

Both stemming and lemmatization are fundamental techniques in NLP, each with its advantages and limitations. Stemming offers a fast and straightforward way to reduce words to their base form but at the cost of accuracy. Lemmatization, on the other hand, provides more accurate and linguistically valid results by considering the context and morphological structure of words, albeit with higher computational complexity.

The choice between stemming and lemmatization depends on the specific requirements of the NLP task. For applications where speed and efficiency are critical, stemming may be preferable. In contrast, for tasks that demand high accuracy and linguistic precision, lemmatization is the better choice. Understanding these differences allows practitioners to select the appropriate technique to optimize their NLP models and achieve the desired outcomes.

2.2.3.4. Text Cleaning and Normalization

Text data often contains noise, such as special characters, punctuation, numbers, and HTML tags. Text cleaning and normalization involve removing or standardizing these elements to ensure consistent and clean text data for analysis. Common techniques include:

- **Lowercasing:** Converting all text to lowercase to ensure consistency. This is done in almost every context, since it ensures that there are no duplicate tokens with the same meaning: if a corpus is not lowercased the words "computer" and "Computer" would be taken as different tokens even though they represent the same word, and especially when working in the semantic level this would lead to two different "meanings" (semantic representation) for the same word, none of which was obtained based on every context where the word appears. It would be like splitting the meaning in two parts, losing relevant information and duplicating data.
- **Handling Numbers:** Depending on the analysis, numbers can be replaced with placeholders or removed entirely. This is very dependant on the goal task.
- **Handling Contractions:** Expanding contractions (e.g., "can't" to "cannot" or "I'll" to "I will") to standardize the text. As with lowercasing this helps eliminating duplicate data since these expressions can appear with different formats through the text, providing two (or more) representations for the same piece of meaning.
- **Whitespace Normalization:** Ensuring consistent spacing between words and removing extra spaces.
- **Removing Special Characters and Punctuation:** Eliminating characters like "@", ",", and "-" that may not contribute to the analysis. As every technique in this list this is very dependant on corpora and desired task.
- **Removing HTML Tags:** Stripping HTML tags from text to focus on the content. This is necessary when extracting text from HTML websites.

These steps are critical in preparing text data for [NLP](#) tasks, enhancing the quality and consistency of the dataset for more accurate analysis and modeling.

2.2.3.5. Regular expressions

Regular expressions, often abbreviated as regex or regexp, are sequences of characters that define search patterns, primarily used for string matching and manipulation. They are powerful tools for processing text data, enabling tasks such as searching, editing, and parsing text. This makes them a good choice when we ask ourselves what tools to use to carry out the above explained text processing techniques. Regular expression can be very simple or very complicated, i have already worked with them during my degree having enough knowledge to use them, but the following books are good places to study Regular Expressions [\[12\]](#) [\[13\]](#) [\[14\]](#).

2.3. Machine Learning

Machine Learning term was coined by Arthur (Lee) Samuel[15], a renowned American pioneer in the field of computer gaming and artificial intelligence, in 1959 with the publication of his paper *"Some Studies in Machine Learning Using the Game of Checkers"*[16] at the *IBM Journal of Research and Development*. Machine learning is a dynamic and interdisciplinary field within artificial intelligence that focuses on the development and implementation of algorithms and computational models, enabling computer systems to autonomously learn patterns, relationships, and representations from data, subsequently utilizing this acquired knowledge for making predictions, decisions, or generating insights without being explicitly programmed for the task at hand. It encompasses a diverse range of methodologies, including supervised, unsupervised, and reinforcement learning, these will be discussed later on. Machine learning draws heavily from computer science, statistics, and mathematics, utilizing algorithms that iteratively refine their performance through exposure to datasets. This iterative learning process allows these algorithms to adapt to varying and evolving patterns within data, making ML a powerful tool for addressing complex problems across a multitude of domains, such as image and speech recognition, natural language processing, recommendation systems, autonomous systems, and predictive analytics. The multidisciplinary nature of machine learning integrates principles from cognitive science, data science, and domain-specific expertise, reflecting its pivotal role in the development of intelligent systems capable of sophisticated tasks and decision-making.

2.3.1. Different approaches

As mentioned above there are different approaches to ML, not every model learns using the same type of data, or using the same learning methodologies. A ML model can take advantage of supervised learning, unsupervised learning, semi-supervised learning or reinforcement learning techniques, and each one has strengths and weaknesses as well as scenarios where it can be applied and scenarios where data is not suitable for that specific approach. We will now discuss these different approaches to ML as well as their differences and main applications.

2.3.1.1. Supervised Learning

Supervised learning is a cornerstone in ML where models are trained on labeled datasets, consisting of input-output pairs. The algorithm learns the underlying patterns and relationships in the data, allowing it to generalize and make predictions on new, unseen instances. Common applications span various domains, including image classification, where models distinguish and categorize objects within images, and speech recognition, where spoken language is transcribed into text. Moreover, supervised learning finds extensive use in regression tasks, such as predicting stock prices or housing values based on relevant features. Its adaptability and effectiveness make it a vital tool in applications where explicit guidance and labeled data are available for training.

2.3.1.2. Unsupervised Learning

Unsupervised learning shifts the focus to unlabeled data, aiming to unveil inherent patterns, relationships, or structures without predefined outcomes. One primary application is clustering, where similar data points are grouped together, facilitating data organization and comprehension. Dimensionality reduction is another key application, as algorithms simplify complex datasets by extracting essential features, aiding in visualization and analysis. Unsupervised learning is instrumental in anomaly detection,

where deviations from normal patterns, like identifying fraudulent activities in financial transactions or irregularities in manufacturing processes, can be effectively identified. This versatility makes unsupervised learning crucial in scenarios where labeled data is scarce or unavailable.

2.3.1.3. Semi-Supervised Learning

Semi-supervised learning represents a hybrid approach, leveraging both labeled and unlabeled data for model training. This methodology is particularly advantageous when acquiring labeled data is resource-intensive. In applications like text classification, where limited labeled documents are available alongside a larger set of unlabeled ones, semi-supervised learning enhances the model's ability to accurately categorize new texts. Similarly, in image recognition tasks, semi-supervised learning allows models to benefit from a more extensive range of data, contributing to the development of robust models with improved generalization capabilities. This bridging of supervised and unsupervised learning principles exemplifies the pragmatic nature of semi-supervised learning in real-world scenarios.

2.3.1.4. Reinforcement Learning

Reinforcement learning constitutes a paradigm where agents learn to make sequential decisions in dynamic environments by receiving feedback in the form of rewards or penalties. Its applications are diverse, ranging from achieving superhuman performance in complex games like Go or chess to training robots for intricate tasks in robotics. Autonomous systems, including self-driving cars, employ reinforcement learning to navigate and make decisions in real-time, adapting to changing conditions. The continuous interaction between the agent and its environment allows reinforcement learning to excel in scenarios where optimal strategies must be learned through trial and error, showcasing its utility in dynamic and complex decision-making processes across various domains.

2.4. Word2Vec

In this section we will start by studying the implementation details of Word2Vec models, we have chosen Word2Vec because it is a pillar in [NLP](#) and has been an icon in this field for years due to its efficiency, simplicity, and robust performance in capturing semantic relationships within large datasets. While other (more modern and advanced) word embedding methods like GloVe (Global Vectors for Word Representation) and FastText offer unique advantages in certain contexts, Word2Vec's combination of efficiency, simplicity, and strong performance makes it a preferred choice for this project. We will then study more thoroughly the differences between all these methods to acquire knowledge about where they come from and why there are so many different methods of training word embeddings.

After having acquired knowledge on the details of the implementation of word embedding we will be ready to build our own model with custom data to understand how this is coded in python.

As we mentioned the first thing to do is study how these models are implemented. To get started we will review some contents from the section [2.2.1.4](#).

We already know what embeddings are, how they look like and what they can be used for, we are now going to learn how they are obtained. We have talked about embeddings being generated by compressing one-hot encoded vectors using Neural Network (NN)s. We already know what are one-hot vectors and how they can be used to represent words, we also know how [NNs](#) process data. But how can we use these amazing tools to generate vectors based on the meaning of words?

In the first place it is important to understand that we would never "tell" the model what words mean, this is, we will never input into the training any data that explicitly tells the model how to represent words. This is the reason why we can say these models are training utilizing unsupervised learning techniques⁴. The only source of information the models have is plain text, human text, vast amounts of natural language text.

Then how does the model learn which words are related? or how does it capture all kinds of semantic relationships as we have seen? If we go back to section 2.2.1.4 we can read that, just like humans learning new languages, these models can extract all this information from raw text just by examining the context where words appear. Therefore we can say that (in this case) Word2Vec relies on the hypothesis that neighboring words in text have semantic similarities with each other.

If we want to see it with an example we can look at two similar words, for instance, the verbs 'love' and 'like' can have similar meanings. If, during the training, the model notices that these words appear in similar context their vectors will end up being closer to each other than they are to words that don't appear in similar contexts. For example our corpus could contain lots of sentences where love and like appear indistinctly, 'I love cats' and 'I like cats' are two sentences where we can see that 'like' and 'love' share context, hence they will be considered similar.

For two words to be considered similar it is not necessary that they repeatedly appear within the corpus with the same words around. It is more like the model capturing general patterns, and as we know ANNs are very good at learning these patterns. For example the model could notice that verbs like 'love' or 'eat' often appear around words like 'I', 'you' or any other pronoun or even names. We know this happens because they share the semantic feature of being verbs, which are always accompanied by a subject, the model doesn't know this but ideally it would be able to understand that these words often appearing together means that they share some semantics, and thus they will be more related than other words that don't share this property.

Once we understand the idea of what learning from context means we will take a deeper dive on this subject by understanding how we are capable of making a NN perform this task of learning word representations based on their context within a corpus. Word2Vec models can be trained following two different (but, in fact, similar) approaches. They are Continuous Bag-of-Words (CBOW) and Skip-gram and they were introduced in the following paper by Mikolov and his team at Google: T. Mikolov, K. Chen, G. Corrado, *et al.*, «Efficient estimation of word representations in vector space», 2013. arXiv: [1301.3781 \[cs.CL\]](https://arxiv.org/abs/1301.3781). This paper was previously cited because it introduced Word2Vec for the first time.

We will now take a closer look at CBOW and Skip-gram approaches to gain a deeper understanding of Word2Vec. Our goal is to understand how a NN can learn a word representation based on the context around that word. Both CBOW and Skip-gram are neural network architectures designed to learn distributed representations of words in a continuous vector space but they have some differences. We will study both of them, but the main idea can be visualized in figure 2.8, extracted from the previously cited paper [2] which introduced Word2Vec for the first time.

When training a Word2Vec model we are training a NN to perform a task, here is where the differences between both approaches are found, the task performed by the NN is different, actually, the task performed by the NN when following the CBOW approach is the opposite as the one performed when following the Skip-gram approach (as we can see in figure 2.8).

⁴Some people call this type of learning self-supervised instead of unsupervised since the context of any given word can be considered as it's label, but since it does not involve any labeling and the tasks performed are usually found within unsupervised learning I will consider it like that

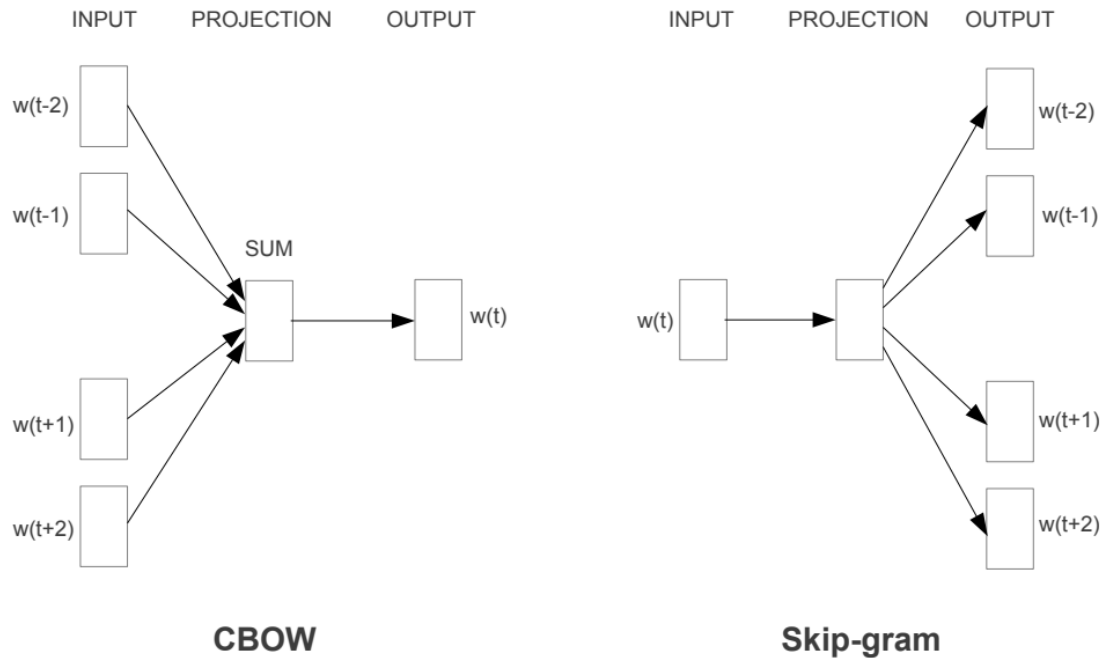


Figure 2.8: CBOW and Skip-gram architecture by Mikolov et al.[2]

2.4.1. Continuous Bag-of-Words Model

As we already know from previous sections, the **CBOW** model begins by representing words as one-hot encoded vectors. In this encoding scheme, each word in the vocabulary is represented by a vector of zeros, except for a single one at the position corresponding to the word's index in the vocabulary. For example, in a vocabulary of 10,000 words, each word would be represented as a 10,000-dimensional vector.

The task given to the **NN** in **CBOW** approach to Word2Vec is to predict a word based on its context, this can be expected since what we are trying to do is get word representations based on the different contexts where they appear through the corpus.

From a more technical point of view in the **CBOW** architecture, the input to the model consists of a sequence of context words surrounding a target word. These context word vectors are then averaged to obtain a single vector representation, which serves as the input to a neural network with a single hidden layer. This **NN**, often a simple feedforward neural network, predicts the target word based on the aggregated context vector.

During training, the CBOW model aims to minimize the prediction error between the predicted target word and the actual target word in the training data. This is typically achieved using techniques like backpropagation and Stochastic gradient descent (SGD), where the weights of the **NN** are adjusted to minimize the loss function.

2.4.2. Continuous Skip-gram Model

On the other hand we have Skip-gram approach, which starts in the same way as the [CBOW](#) model, it begins by representing words as one-hot encoded vectors, but the similarities stop here.

Unlike CBOW, which predicts a target word based on its context, Skip-gram operates in reverse by predicting the context words given a target word.

In the Skip-gram model, the input consists of a single target word represented as a one-hot encoded vector. The objective is to predict the context words surrounding the target word within a fixed window size. To accomplish this, Skip-gram employs a [NN](#) with a single hidden layer, where the target word vector serves as the input.

During training, Skip-gram aims to minimize the prediction error between the predicted context words and the actual context words in the training data. This is typically achieved through techniques like backpropagation and [SGD](#), same as [CBOW](#) (these are typical techniques when training [NN](#)). By iterating over the entire corpus multiple times, the model adjusts the neural network's weights to improve its predictive performance. As a result, Skip-gram learns to generate accurate representations of context words based on a given target word.

In short we can say that the key difference between these two architectures or Word2Vec approaches is that [CBOW](#) is context to target prediction, while Skip-gram is target to context prediction.

[CBOW](#) tends to work better for frequent words and is computationally more efficient (predicting one word is a lighter task than predicting few of them), while Skip-gram often performs better for infrequent words and captures more semantic nuances.

In practice, the choice between [CBOW](#) and Skip-gram depends on the specific task and the characteristics of the dataset being used. [CBOW](#) is faster and may be suitable for larger datasets, while Skip-gram may be more accurate for capturing intricate semantic relationships.

2.4.3. Creation process

At this point you may have noticed that both [CBOW](#) and Skip-gram neural networks are performing different tasks but we said they are interchangeable with each other and are used to get the same result, dense vectors that represent words of a vocabulary in a multidimensional space. How can they be used to obtain equivalent results if the task performed is different and the output format is not the same?

You may have also noticed that the tasks we are asking the [NNs](#) to learn are not the same as learning how to represent words as vectors. Also we haven't even compressed the one-hot vectors to a manageable dimensionality (as we said we would do in section [2.2.1.4](#)). For instance the output size of [CBOW](#) is the same as the number of words in the vocabulary, since we get a probability for each of those words, which can be huge.

This has an explanation and it is actually surprising. The following contents will be studied and explained assuming we are working with an Skip-gram based architecture, this is because it is usually preferred over [CBOW](#) due to its slightly better performance, but they are essentially the same.

2.4.3.1. The fake task

Unlike most of the [NN](#) applications, when training for word embeddings we will not use the trained [NN](#) to make predictions at all. The approach here differs from what we are used to, usually we train a [NN](#) to perform a task we will need to perform in the future, this means we put the neural network

through a training process using training data, and after that we use the [NN](#) to reproduce the results, or perform the same task with real data, real data that may have appeared during training or not. The network should have learned patterns from training data and is capable of applying them when we need to.

In the case of Word2Vec (both [CBOW](#) and Skip-gram) we will completely discard the output layer (as well as the input one) of the trained neural network after the training is completed, eliminating it's capacity to perform the task we trained it to perform. Since the output layer in a [NN](#) is designated to providing the desired output (in our case - Skip-gram architecture - this output would be the representation of the expected context around the word we provided to the input of the model) and we are removing it, we will no longer be able to predict a context based in a word using our Skip-gram [NN](#).

Instead we will just take the learned weights of the neural network as the word embeddings, this is the weight matrix between the input layer and the intermediate hidden layer (we just need to understand the concept here, in the following section we will understand it more deeply with examples). So, if we just need this weight matrix we can get rid of everything else withing the neural network, even the input layer, we don't need to input one-hot encoded vectors into the neural network to get these values and this is because the matrix multiplication between a matrix and a one-hot vector will always result in just a row of the matrix, the row in the same position as the '1' in the vector, an example could be the following operation.

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix} X \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \\ x_{41} & x_{42} & x_{43} \\ x_{51} & x_{52} & x_{53} \end{bmatrix} = \begin{bmatrix} x_{31} & x_{32} & x_{33} \end{bmatrix}$$

Where the result of the multiplication is just a vector containing the values of the corresponding row in the matrix.

This means the process of inputting words into the [NN](#) up to the hidden layer was just working as a look-up table, where each one-hot vector had one assigned row of the weight matrix, and then this row was passed to the output layer to generate the expected context representation. What we will do is take this matrix, and assign each row with it's respective one-hot encoded vector (as this is what the [NN](#)'s input and hidden layers were doing anyways) and keep it as it's word embedding. Now we have dense vectors that represent each word in the vocabulary based on the expected context where these words appear.

This is the reason why we call the task given to the neural network in the training phase a "fake task", we never wanted the [NN](#) to learn patterns and apply them later to new or unseen data, we wanted the weight matrix that resulted from the training process.

Also, even if we wanted to do so, the [NN](#) wouldn't be able to generate an expected context around a new word, this is due to the fixed input size and the fact that our inputs will always be one-hot encoded vectors. We can't generate a one-hot vector for any new word after the training process if we need to have the same size in the input vectors, this is because all possible one-hot encoded vectors for the training vocabulary size have already been generated and used during the training. For example if our vocabulary had a size of 10 words, all our input one-hot vectors would be of size 10, each one containing nine 0's and a 1 in a position that is different to the rest of them, so if after the training process we wanted to

feed the [NN](#) with new data we would need to generate a new one-hot vector with the 1 being in the 11th position of the vector, which is impossible if the size of the input layer of the [NN](#) is fixed at 10. ⁵

In summary, the training process of Word2Vec based word embedding models consists of a "fake task", which is never performed again after the training process, but provides a weight matrix that serves as representation for words in the vocabulary.

2.4.3.2. Word2Vec Neural Network architecture

We have learned the idea behind the training process of Word2Vec models, now we are going to dive into the specific architecture of the model, we will learn how to build a [NN](#) that is able to provide the results we are following.

We know [NNs](#) consist of layers, each layer can be different, there are several types of them, and they can also be of different sizes. We will break the model down to layers and study each one of them. First of all we need to know how many layers a typical Word2Vec [NN](#) has, every neural network has at least two layers, an input layer, and an output layer, in this case we will also find a hidden layer in between both of the previously mentioned. So typically we will find a 3-layer [NN](#) in every Word2Vec (either [CBOW](#) or [Skip-gram](#)) model.

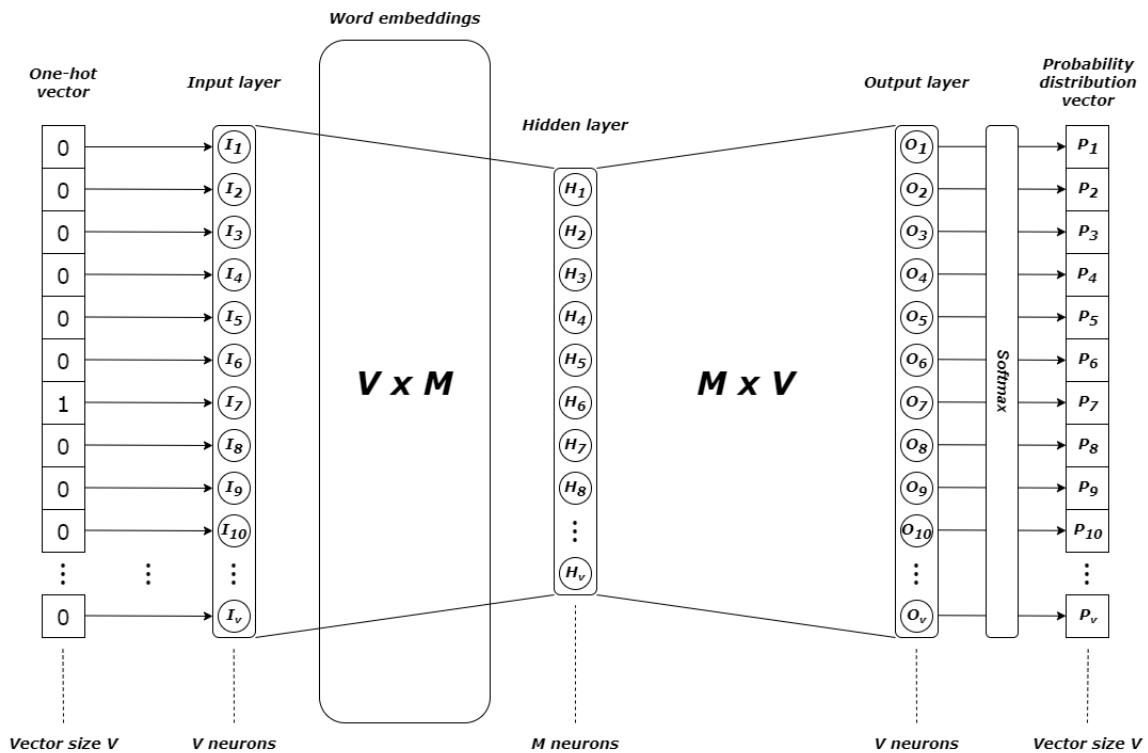


Figure 2.9: Skip-gram neural network architecture in detail

In the first place we find the input layer, this layer is common to every [NN](#) and in this case has a size of N where N is the number of words in the vocabulary, the reason to this was previously explained, it is

⁵We will later study fastText, which brings improvements to this aspect of Word2Vec, allowing us to generate embeddings for words that were not seen by the model during the training process by introducing subword information. We will later get more in detail about this but the fact that it allows us to generate embeddings for unseen or Out-of-vocabulary (OOV) words is not because the way it works is different but because the basic unit of information is subword instead of word, this means it can't generate embedding for [OOV](#) subwords, it just builds embeddings for new words based on known subwords, so the previous statements still apply. The goals of fastText and any subword based embedding technique is not being able to apply the learnt task to unseen data but to bring more flexibility to the field in terms of unseen vocabulary processing.

the fact that the inputs we use to feed this [NN](#) are always one-hot encoded vectors that represent words, and we build one vector for each word in the vocabulary, resulting in the size of these vectors having the same number of dimensions as words in the vocabulary.

The output layer is intended to represent the information we are "asking" the [NN](#). In the case of a [CBOW](#) approach each neuron in the output layer should represent the probabilities that the word associated to that specific neuron belongs to the context fed as input, on the other hand, for a Skip-gram approach it should represent the probability that a randomly chosen word in the context of the word fed as input is the word associated to that neuron. Therefore the number of neurons in the output layer has to be the same as the size of the input, this is the number of words in the vocabulary. In this output layer, *softmax* is used as activation function, *softmax* is ideal for multiclass classification, and this is why it is used in this case, we will study more details about this function later.

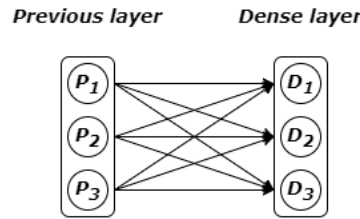


Figure 2.10: Dense or fully connected layer diagram with 3 neurons in both dense and previous layers

In between the input and the output layer we find a hidden layer, as we said before we typically only find one of these, and the size of this layer is not dependant on the specific vocabulary we are using, instead we can adjust this size, it is an hyper-parameter we can tune to try and improve the performance of our model. In the paper that gave birth to Word2Vec [\[2\]](#), google researchers used a size of 300 neurons for this layer, and this set a standard that remains today, so most trained models use a size around 300, but it is not fixed. And the reason why this is so important is because the size we pick for this layer will set the size of our word embeddings, the dimensionality of our dense vectors that represent each word in the vocabulary. The reason to this is this layer being a densely connected layer (see figure 2.10), this means every neuron in the input layer is connected to each one of the neurons in the hidden layer, knowing how [NNs](#) work we can infer that the weight matrix between these two layers will be of size $M \times N$ where M equals the size of the input layer (size of vocabulary, fixed) and N equals the size of the hidden layer, this means when we extract embeddings from this matrix for each word we will obtain M vectors of size $1 \times N$ and these N -dimensional vectors will represent words.

To visualize this, let W be the complete weight matrix found between input and hidden layers, with size $M \times N$:

$$W = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ x_{31} & x_{32} & x_{33} & \dots & x_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix}$$

As we said, M would be the vocabulary size, the number of words we wanted to learn embedding representations for, and N the desired embedding size given by the number of neurons in the hidden layer.

Then the dense vector that represents the word number one in our vocabulary (the one we encoded as $[1 \ 0 \ 0 \ \dots \ 0]$) would be:

$$[x_{11} \ x_{12} \ x_{13} \ \dots \ x_{1n}]$$

And the one for the word number i (encoded as a one-hot vector with the 1 in the i^{th} position):

$$[x_{i1} \ x_{i2} \ x_{i3} \ \dots \ x_{in}]$$

As we can see they will all be of size n , and each word in the vocabulary will have its own dense vector based on the context around it across the corpora.

Having all the information, the final architecture of the **NN** we would want to build can be seen in the figure 2.9

2.4.3.3. Training process

Once we know how to build the appropriate **NN**, let's study how can we train it so that we get the result we are expecting to find. In this subsection we will go over different topics that are relevant in the training process of Skip-gram.

2.4.3.3.1. Training algorithm

When training a **ML** model we usually want to minimize a loss function (or maximize a success probability, it's the same thing), with this purpose, we use training algorithms, there are many of them such as Difference Target Propagation (2015) [17], Hilbert-Schmidt Independence Criterion (2019) [18], or the most well known of them Backpropagation.

The training algorithm used when training a Skip-gram word embedding model is backpropagation. Backpropagation is a very common algorithm in the **ML** field, the goal of this algorithm (and other alternatives) is to help finding the weights on the **NN** that minimize the loss. In the case of backpropagation, this is achieved by applying a gradient descent algorithm, in this case **SGD** (learn more at [19] or this book J. Lu, *Gradient descent, stochastic optimization, and other tales*, 2024. arXiv: 2205.00832 [cs.LG]).

Gradient can be explained as a vector that points in the direction where given function increases (or decreases, if we take the opposite direction) the fastest, this is used by **SGD** in order to efficiently minimize the loss function values by updating the weights in the **NN**.

As we said, gradient is a complex mathematical concept that, in this case, can be interpreted as a pointer towards the direction where we should move (in the function) to reach a local minimum as fast as possible (actually, since it describes the fastest **increase** we would move to the opposite direction), in a very simplified way we could represent the gradient as follows:

$$\nabla f(x) = D[f(x)] = \left(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right)$$

This means we can think about it as a vector that contains all of the partial derivatives with respect to each variable. As mentioned it can be used as a way to minimize any given loss function.

When training the model we specify an hyper-parameter that is usually called 'epochs' and denote the number of iterations we will perform in the training process. This is the number of times the training algorithm will be used and here is where SGD helps us. When performing gradient descent, we would need to run through every sample in our dataset to perform each weight update, which is a very computationally expensive procedure. Instead when using SGD this becomes a much more lightweight task. As it's name indicates ⁶, in this case only one or a small (random) subset (which size can be adjusted as an hyper-parameter usually called batch-size) of the training sample would be used to perform a weight update in a particular iteration, achieving a much less computationally expensive task. Even though SGD can sometimes provide less accurate results, it is not usually the case, and if there is a difference it will not be great enough to worry about it, that is why SGD is most of the times the preferred method to update the weights in a NN through backpropagation, there are also research that propose that SGD is better at generalization than regular gradient descent [22].

Now that we understand gradients and SGD we can easily define backpropagation with two main steps:

- Forward propagation: This is just calculating the output for a training sample (this output will be given by the output layer and in this case an activation function that we will study down below), by passing the input through all the NN layers we can easily compute the output for that specific input, this is also the process used to make predictions once the NN is trained. After the output is obtained we would need to calculate the loss function which is basically a measure of how wrong (or correct) the NN is by comparing the real values versus the predicted values (more detail on loss function below).
- Backward propagation: This steps involves calculating the derivatives for the loss function with respect to each one of the weights, obtaining the gradient, and then using it's information to update the weights (at a fixed rate, given by an hyper-parameter called learning rate).

These steps are repeated the desired amount of times and we could track the variation of the loss function in each iteration to have an understanding of how it is decreasing (or increasing, which shouldn't be the case), and this should be an indicator of how precise the model is getting during the training process.

2.4.3.3.2. Loss function

The training objective of the Skip-gram model is to find word representations that are useful for predicting the surrounding words in a sentence or a document [3]. The formal definition of this statement would be: given a sequence of training words $w_1, w_2, w_3, \dots, w_T$, the objective of the Skip-gram model is to maximize the average log probability given by:

$$\frac{1}{T} \sum_{t=1}^T \sum_{-c \leq j \leq c} \log p(w_{t+j} | w_t)$$

Where c is the size of the training context (this is the chosen window size to generate the context of vocabulary words) and $p(w_{t+j} | w_t)$ is defined by the *softmax* function. This would be the same as minimizing the negative of the function which can be found in many documents.

⁶SGD stands for Stochastic Gradient Descent, the definition of Stochastic is: Stochastic (/stækstɪk/; from Ancient Greek (stókhos) 'aim, guess') refers to the property of being well-described by a random probability distribution. [21]

This loss function (negative of the formula above) is usually called negative log likelihood, when used along softmax (this is multiclass classification) it can be called cross-entropy, it can be plotted against the prediction accuracy and we would obtain the representation on figure 2.11.

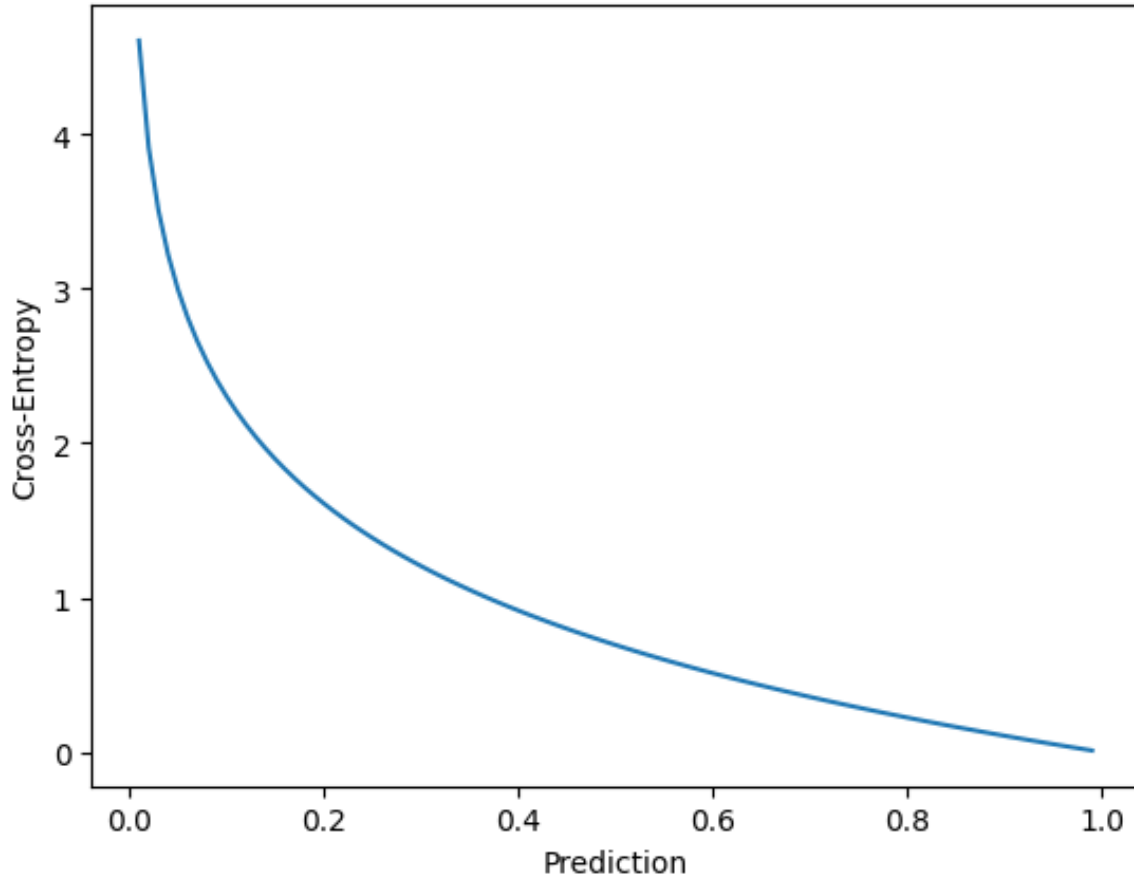


Figure 2.11: Cross entropy loss versus prediction accuracy (generated with matplotlib for python)

2.4.3.3.3. Activation function - softmax

Softmax is a vital activation function frequently employed in neural networks, particularly in the output layer, to address multiclass classification problems. Its primary function is to transform the raw output scores of a [NN](#) into a probability distribution over multiple classes. This transformation ensures that the output values represent the likelihood of each class, with the sum of probabilities across all classes equaling 1. Such probabilistic outputs enable intuitive interpretation and facilitate decision-making in classification tasks. It's applications in machine learning are countless, since multiclass classification is a very common task within the field, this applications range from classifying images into multiple different classes (for example identifying what fruit from a set of possible options appears in an image) to text sentiment analysis (it would classify the text into a set of possible sentiments).

Jumping into *softmax* formal definition, the function is expressed as:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

Where \vec{z} is the input vector and K is the number of elements in \vec{z} :

$$\vec{z} = [z_1, z_2, \dots, z_K]$$

The numerator of the given formula will always be in the range:

$$(e^{-\infty} = \frac{e}{\infty} = 0, e^{\infty} = \infty) \rightarrow (0, \infty)$$

The denominator contains the numerator itself added with K-1 other values ranging $(0, \infty)$ since it is a summation of all possible numerators, this way when the numerator tends to infinity the denominator will too, and as a result the quotient of the division will tend to 1. On the other hand whenever the numerator is equal to 0, the quotient will be 0 too. Finally, whenever the numerator is not equal to 0 or ∞ , it will be just a fraction of the denominator (since it is part of the summation). This way we can ensure that the output values of *softmax* will be always in the range $(0, \infty)$.

Also, since the denominator is just a summation of all the numerators we can state the following:

$$\sum_{i=1}^K \sigma(\vec{z})_i = \frac{\sum_{i=1}^K e^{z_j}}{\sum_{j=1}^K e^{z_j}} = 1$$

This can be read as the summation of all the values in the output of *softmax* for vector \vec{z} equals 1. And this means that all the probability scores provided by *softmax* for each value of \vec{z} add up to 1, thanks to this property we can say that *softmax* provides a valid probability distribution across multiple classes.

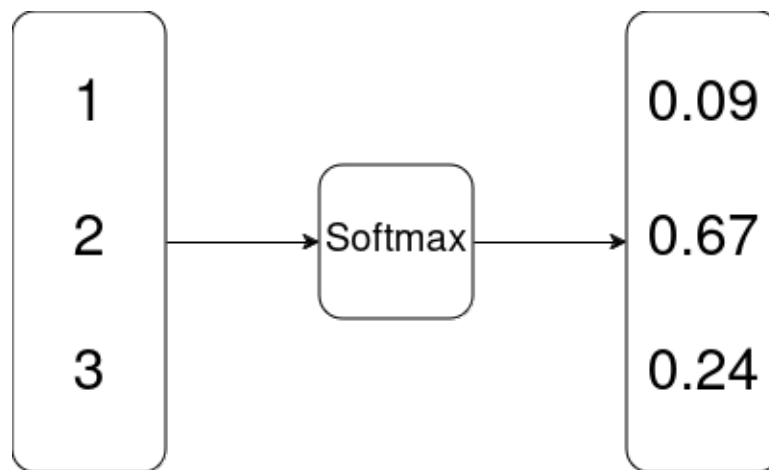


Figure 2.12: Softmax example in and out

Whenever an activation function is studied, it is useful to plot the function output values, to be able to visualize how the function behaves in different input value ranges, however, *softmax* is a multi-variable function that can't be easily plotted, instead we can see an example of input and output to this function in figure 2.12

As we can see *softmax* has a wide range of applications in fields such as computer vision, [NLP](#) and many more. In our case this output function works ideally, since the goal here is to generate a probability distribution for each word over the rest of words in the vocabulary.

2.4.3.3.4. Negative sampling

Until now, all of the contents studied in this section were based on the model architecture published in T. Mikolov, K. Chen, G. Corrado, *et al.*, «Efficient estimation of word representations in vector space», 2013. arXiv: [1301.3781 \[cs.CL\]](#) as we have mentioned before. This paper set a new standard in the industry by giving birth to the most well known word embedding technique, however shortly after the first published paper, the same researchers team at Google (with the addition of Ilya Sutskever)

published a new paper presenting some improvements made to this first Word2Vec model description T. Mikolov, I. Sutskever, K. Chen, *et al.*, «Distributed representations of words and phrases and their compositionality», 2013. arXiv: [1310.4546v1](https://arxiv.org/abs/1310.4546v1) [cs.CL].

In this section and the following one [2.4.3.3.5](#) we will go over these contents to gain a more complete idea of how these embeddings are trained.

We will start by talking about negative sampling. Negative sampling is a technique that consists on selecting a small amount of negative samples (this is word pairs that doesn't share context instead of word pairs that do share context as we have seen before) and use them for training instead of using all the words in the vocabulary. This is done by modifying our task from trying to predict a probability of being nearby for each word on the vocabulary, to trying to predict a probability for each word in our sample (small amount of negative samples and the positive sample we were using), being around the target word. This changes the task from a multiclass (thousands of classes) classification to a few binary classification problems, depending on the size of the negative sample.

An example for this would be for the pair of words ('cat','pet') that we extracted from the dataset as a pair of words, we would create other negative samples like ('cat','building'), ('cat','computer'), ('cat','fruit')... and ask the model to predict the probabilities of these pairs of words to appear together, backward propagating the loss for these chosen words and updating only the weights for the words, 'pet', 'building', 'computer', 'fruit'... instead of the weights for every word in the vocabulary.

This technique called negative sampling provides a way to train a model in a much more efficient way in terms of time and computing cost.

2.4.3.3.5. Subsampling of frequent words

Subsampling of frequent words is a technique used in the Skip-gram model to improve the quality of word embeddings and reduce computational complexity. This process helps mitigate the imbalance caused by very frequent words, which can dominate the training process and lead to less meaningful embeddings for less frequent words. This is because usually, most frequent words can carry less semantic information than less frequent ones. "For example, while the Skip-gram model benefits from observing the co-occurrences of 'France' and 'Paris', it benefits much less from observing the frequent co-occurrences of 'France' and 'the', as nearly every word co-occurs frequently within a sentence with 'the'", "To counter the imbalance between the rare and frequent words, we used a simple subsampling approach: each word w_i in the training set is discarded with probability computed by the formula:

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

where $f(w_i)$ is the frequency of word w_i and t is a chosen threshold, typically around 10^{-5} .

These are fragments from the previously cited paper that presented these improvements [3], which perfectly explain the process. Every word has a probability of being discarded depending on how many times it appeared already, making more frequent words less invasive, and less frequent words more important.

As mentioned above, many other approaches to word embedding representation have been developed in the last years, while Word2Vec is the best known technique, it is not the most advanced one. Different

approaches such as GloVe or fastText may offer different advantages (or disadvantages) compared to Word2Vec. We will now study both of these word embedding approaches in a less detailed way in order to understand what different approaches may offer or they are developed. This way we will obtain a more general vision of the field, having a deeper understanding of the advancements within this field.

2.4.4. Differences with GloVe

GloVe, or Global Vectors for Word Representation, represents a significant advancement in the realm of word embeddings within NLP. Developed by researchers at Stanford University, GloVe is a count-based model that aims to capture the global statistical properties of word co-occurrences in a corpus. Unlike previous approaches like Word2Vec, which focus on local context windows, GloVe considers the entire corpus and learns word embeddings by optimizing an objective function based on global word co-occurrence statistics.

The foundation of GloVe lies in the construction of a co-occurrence matrix X , where each element X_{ij} represents the number of times word i appears in the context of word j (or vice versa) in the corpus. This matrix encapsulates the relationships between words based on their co-occurrence patterns throughout the corpus.

An example can be seen in table 2.4, this table represents the GloVe co-occurrence matrix for the following corpus:

- 'i love coding'
- 'i love NLP'
- 'i like pizza'

Assuming our reference window to consider words co-occurrent withing the text is of size 1, this means only the previous and next words to a given word are considered co-occurring (in the first sentence 'i' and 'love' are co-occurrent since they are next to each other, but 'i' and 'coding' are not since the distance between them is two words and our window size is one).

count	i	love	coding	NLP	like	pizza
i	-	2	0	0	1	0
love	2	-	1	1	0	0
coding	0	1	-	0	0	0
NLP	0	1	0	-	0	0
like	1	0	0	0	-	1
pizza	0	0	0	0	1	-

Table 2.4: GloVe co-occurrence matrix

Then, this co-occurrence matrix is factorized by minimizing a loss function. A diagram can be seen in figure 2.13

GloVe's objective function is designed to capture these relationships by minimizing the difference between the dot product of word vectors and the logarithm of their co-occurrence probabilities. By optimizing this objective function, GloVe learns word embeddings that encode semantic similarities and relationships between words in a continuous vector space.

The main difference between Word2Vec and GloVe is that GloVe incorporates global co-occurrence statistics adding global information to the model, while Word2Vec does not have any explicit global

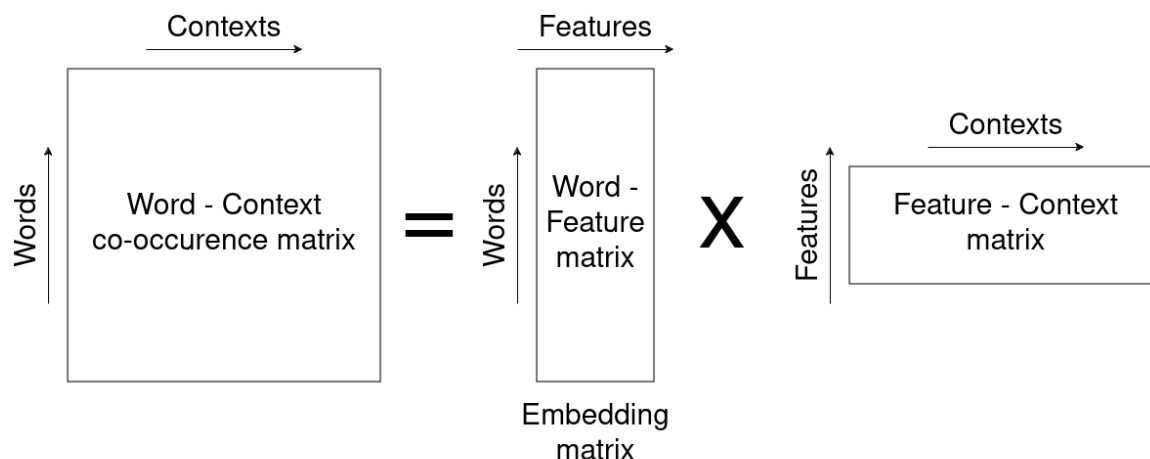


Figure 2.13: GloVe matrix factorization process (inspired by image at [23])

information embedded in it by default. GloVe creates a global co-occurrence matrix by estimating the probability a given word will co-occur with other words. This presence of global information makes GloVe ideally work better. Although in a practical sense, they work almost similar and people have found similar performance with both.

The resulting word embeddings generated by GloVe carry rich semantic information, allowing for efficient representation and manipulation of textual data. These embeddings have been shown to perform very similar to the ones provided by Word2Vec, but since they are explicitly capturing global context of the words they may capture some more global information or relationships between words in different contexts.

In summary, GloVe represents a powerful approach to learning word embeddings by leveraging global word co-occurrence statistics. Its ability to capture both local and global semantic relationships between words makes it a valuable tool in various NLP applications, particularly in scenarios involving large corpora where capturing global context is crucial.

2.4.5. Differences with FastText

FastText [7], developed by Facebook AI Research (FAIR), is an extension of the Word2Vec model that introduces a novel approach to handling OOV words and capturing word morphology. Unlike traditional word embedding models that learn embeddings only for whole words, fastText considers subword information during training, allowing it to generate embeddings for both whole words and character n-grams (subwords).

Word embedding techniques like Word2Vec and GloVe provide distinct vector representations for the words in the vocabulary. This leads to ignorance of the internal structure of the language. This is a limitation for morphologically rich language as it ignores the syntactic relation of the words. As many word formations follow the rules in morphologically rich languages, it is possible to improve vector representations for these languages by using character-level information.

At the heart of fastText is the idea of breaking words into smaller units called character n-grams, which are sequences of characters of length n . By learning embeddings for these subword units, fastText can effectively handle OOV words (i.e., fastText can find word embeddings that are not present at the time of training by building them from already learnt n-grams or subwords) and capture morphological

similarities between words. This is particularly beneficial in languages with complex morphology, where words may exhibit various morphological forms.

During training, fastText learns embeddings for both whole words and character n-grams by optimizing an objective function that combines information from both levels of representation representing words as the average of these n-gram embeddings. This enables fastText to encode semantic information at multiple levels of granularity, from individual characters to entire words, resulting in rich and informative word embeddings. Like the Word2Vec approach, fastText uses CBOW and Skip-gram to compute the vectors.

Let's visualize this with an example: Consider the word 'code' and $n = 3$, then the word will be represented by character n-grams:

'<co', 'cod', 'ode', 'de>' and '<code>' ⁷

As mentioned above in FastText, each word is represented as the average of the vector representation of its character n-grams along with the word itself. So, the word embedding for the word 'code' can be given as the average of the vectors for '<co', 'cod', 'ode', 'de>' and '<code>'. This way if the word 'coding' didn't appear in model training, but the model did learn the representations for 'code' and other words ending with '-ing', fastText would be able to represent an OOV word by averaging already known n-gram representations.

The embeddings produced by fastText find applications in a wide range of NLP tasks, including text classification, language modeling, and information retrieval. They are particularly useful in scenarios where dealing with OOV words is essential, such as in user-generated content or social media text.

In conclusion, fastText represents an extended approach to word embeddings that incorporates sub-word information to handle OOV words and capture word morphology effectively. Its ability to encode semantic information at multiple levels of granularity makes it a versatile tool in NLP, capable of addressing various challenges in text processing and analysis.

⁷(The characters '<' and '>' represent start and end of word respectively)

Chapter 3

Experimentation

After all the previous documentation phase, in this chapter we will delve into the applications of the previously explored concepts. We will see how theory is applied in the real world with examples on real pieces of text, providing us a better image or understanding on how all the previously explained is actually used. For this chapter we will study a coded implementation of a semantic similarity calculation system, that will prove that theory works the way we thought and will lay a foundation for a future system that can perform similar tasks with the needed ancient language text.

In order to code a first semantic similarity model we will need an embedding model so that we can vectorize words from text and then compare them utilizing any of the previously studied methods. This is why this section will be focused on experimenting with word embeddings.

In the first place we will study the tools needed, having a deep understanding of our working tools is very beneficial for the final result. Also we will compare some alternative tools and find the one that better fits our needs.

3.1. Tools used

The general tools used in almost every coding project is a computer with a GNU/Linux[24] operating system and a code editor. In this case we are using Ubuntu as a operating system and Visual Studio Code as a code editor.

However it is more interesting to study the tools that are very specific to our work, tools that we use to solve NLP problems.

3.1.1. Programming language

As a programming language we will use Python. Python is the most widely used programming languages for any type of tasks in the field of AI and this includes ML and NLP tasks. Python's widespread adoption in these fields can be attributed to its versatile and developer-friendly features, but the main reason for python to be above any other language is it's libraries. Python's main benefit for any developer working on AI or ML is the huge number of libraries that are available when using it, they can extend Python's base functionalities in unimaginable ways. Additionally, Python's simplicity and readability make it an ideal language for rapid prototyping and experimentation, crucial aspects in the iterative process of developing AI algorithms. Moreover, Python's capabilities in handling data make

it a natural fit for data science tasks. Libraries like Pandas, NumPy, and Matplotlib facilitate efficient data manipulation, analysis, and visualization, enabling data scientists to glean valuable insights. The wealth of resources, active community engagement, and its user-friendly syntax contribute to Python's prominence in these cutting-edge fields.

From the previous description we can extract the following key features that make using Python the best option for this experimenting phase:

- **Libraries**, they are the main reason why a researcher or developer would think of python as their first option, most times python is needed to be able to work with some specific libraries or frameworks.
- **Python's simplicity and readability**, this makes of it an ideal language for rapid prototyping and experimentation.
- **Python's capabilities in handling data**, this saves a lot of time to developers by providing fast and easy ways to perform complex operations with numeric or not numeric data.
- **Python's big and active community**, almost all developers working in [AI](#) use python, this makes the programming language to have a huge community that is building projects similar to this and this can be very helpful if we ran into any issue.

So these are the reasons why python was chose as a programming language to run these experiments.

3.1.2. Package manager

As we stated before, much of the power of python comes from its libraries, to easily install them without having to download, compile and link projects we will be using a package manager for the following work. A package manager is used to simplify the management of dependencies within a project, to easily download libraries and updating them. The main benefits for using a package manager are the following:

- **Dependency Management:** Python projects often rely on external libraries and modules. A package manager helps manage these dependencies, ensuring that the correct versions are installed and that there are no conflicts between different packages.
- **Easy Installation:** Package managers streamline the installation process for Python packages. With a simple command, users can download and install the necessary libraries, making it convenient for developers to set up their environments.
- **Update Management:** Package managers provide commands to update installed packages easily. Developers can use these commands to stay up-to-date with the latest versions of libraries, benefiting from bug fixes, new features, and security updates.
- **Automated Dependency Resolution:** Package managers automatically resolve dependencies, fetching and installing all required packages along with their dependencies. This simplifies the process for developers, reducing the likelihood of manual errors and ensuring a smooth installation experience.
- **Virtual Environments:** Package managers work well with virtual environments, allowing developers to create isolated environments for different projects. This isolation helps prevent conflicts between packages and ensures that each project has its own set of dependencies.

In general a package manager is a tool that simplifies the management of dependencies, easing the installation process, and ensuring a standardized and efficient way for developers to work with Python libraries and packages. This is the reason why we will be using *pip*[25] (pip Install Packages), it is the default package manager for python and even though there are more specialized package managers for ML projects such as *conda*[26], *pip* works well enough for this project and there is no need on using a more advanced one.

As mentioned above, one of the strengths of package managers comes when they are used along with virtual environments, these environments create isolated spaces to install libraries, this avoids a lot of dependencies conflicts and is always a good practice to use them. For this project we will use a virtual environment to install all the dependencies without having any issue with already installed packages in the system. We will use *venv* (Virtual ENVironment) in this case which is also the default virtual environment manager that comes pre-installed with python, and also does good enough for this little experimenting. Package manager *conda*[26] also comes with a virtual environment manager but as with the package manager there is no need on using the more advanced features it provides.

3.1.3. Environment

For the development of this project I will be using *Jupyter Notebooks*[27]. Jupyter Notebooks are interactive, web-based computational environments designed for data science, machine learning, and scientific computing. They provide a dynamic platform where users can create and share documents containing live code, equations, visualizations, and narrative text. Supporting multiple programming languages such as Python, R, and Julia, Jupyter enables an iterative and exploratory approach to coding, allowing users to execute code step-by-step. One of the primary advantages of Jupyter Notebooks lies in their interactive computing capability, creating a perfect environment for experimentation and quick prototyping. The integration of rich text, including Markdown support, facilitates the creation of well-documented analyses and reports. Visualizations can be seamlessly incorporated using popular libraries like Matplotlib, enhancing the ability to communicate insights effectively. Jupyter Notebooks also promote collaboration, as they can be easily shared with others, exported to various formats, and used for educational purposes. This combination of interactivity, documentation, and collaboration makes Jupyter Notebooks a versatile and widely adopted tool in the fields of data science and machine learning.

In this case the main advantage we can obtain from Jupyter Notebooks is the ability to execute pieces of code step-by-step, allowing us to check the state of the data at any moment and making it easier and quicker to conduct different experiments on the code.

3.1.4. Libraries

Most of the power of Python lies in its libraries, these being extremely useful and covering all kind of needs for many computer scientists or software developers. In this section we will cover the most important libraries we will be using when implementing code, explaining what purpose they serve and why they will be used.

3.1.4.1. Numpy

If there is one python library that stands out above the others, that is Numpy.

NumPy, short for Numerical Python, is a powerful open-source library for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection

of mathematical functions to operate on these arrays efficiently. At its core, NumPy is essential for scientific and mathematical computations, forming the foundation for many other libraries in the Python data science ecosystem. NumPy's main data structure is the `ndarray` (n-dimensional array), which allows for efficient manipulation of large datasets and enables the implementation of mathematical operations on entire arrays without the need for explicit loops. One of the significant advantages of NumPy is its performance, as most of its operations are implemented in C and Fortran, making it significantly faster than equivalent operations in pure Python. NumPy also provides functionalities for linear algebra, Fourier analysis, random number generation, and more. Its versatility and efficiency make it a fundamental tool for various applications, including machine learning, scientific research, and data analysis, providing a solid foundation for numerical computations in the Python programming language.

3.1.4.2. NLTK

Natural Language Toolkit (NLTK)[28] is a powerful and comprehensive open-source library for NLP tasks in Python. Developed by researchers at the University of Pennsylvania, NLTK serves as a valuable resource for tasks related to human language data, including text processing, tokenization, stemming, tagging, parsing, and semantic reasoning. It provides an extensive collection of libraries and tools for various linguistic and computational tasks, making it a versatile solution for researchers, developers, and educators working in the field of NLP. NLTK includes data sets, lexical resources, and text corpora, allowing users to explore and analyze language data effectively. One of the notable strengths of NLTK lies in its educational focus, as it is widely used for teaching purposes to introduce students to fundamental concepts in NLP. Its modular design and ease of use make it accessible for both beginners and experienced practitioners. With NLTK, users can perform tasks such as part-of-speech tagging, sentiment analysis, named entity recognition, and more, making it a great tool in the realm of natural language processing within the Python programming ecosystem.

3.1.4.3. Gensim

Gensim is a robust and efficient open-source library for processing raw, unstructured digital texts (“plain text”) using unsupervised machine learning algorithms in Python. Developed by Radim Řehůřek, Gensim is designed to handle large text corpora efficiently, making it particularly well-suited for applications in NLP and ML. One of its notable features is the implementation of advanced embedding techniques, such as Word2Vec, which we will be using. Gensim is renowned for its scalability, enabling the processing of large datasets and the training of models on vast amounts of text. Its modular and extensible design facilitates integration with other popular NLP libraries, making it a valuable component in the natural language processing pipeline. Gensim is widely used for applications such as document clustering, information retrieval, and semantic analysis, contributing to its reputation as a versatile and powerful library for text analysis within the Python ecosystem.

3.2. Using pre-trained models

To get started with the code we are going to run some experiments using some pre-trained word embedding model. In the next section we will explore the internal structure of three pre-trained models, allowing us to understand how these models are trained and how they learn to represent embeddings but, for now, we only need to know that the model we are going to use is a set of pre-trained 300-dimensional embedding models that were trained over corpuses of billions of words. We are going to load these models

from the Gensim library, since it provides several pre-trained word embedding models ready to use, some of them were trained using Word2Vec architecture but some others were trained using a different set of techniques such as GloVe or fastText¹. Even if they have been trained over different corpus or using a different architecture, all of them provide a very good representation for words.

The goal of this subsection is to provide a more tangible idea of what embeddings are. In the theoretical study we have learned what embeddings are mathematically (dense n-dimensional vectors that represent the meaning of words), in this section we are going to learn how they look like and how we can perform basic operations to them with code. As mentioned we will be using python for its utility and popularity but vectors are a general computer science concept that almost any popular programming language can handle.

Moving on to the code, the first thing to do once the environment is set up with the libraries already mentioned, is load them in the document to be able to use all the utilities they provide. For this first example we will only need Gensim, because it provides pre-trained word embedding models as well as some built-in functionalities such as cosine similarity, which we will be using 3.1.

```
1 import gensim.downloader as api
```

Python

Figure 3.1: Import Gensim API

After importing the library, we can use Gensim's Application Programming Interface (API) to download the pre-trained models. In this case we chose to download three of them, one was trained using Word2Vec as base, the second one used GloVe algorithms and the third one was trained using a fastText approach. The one that uses Word2Vec was trained over a corpus of about 100 billion words extracted from Google News, the one based on GloVe was trained over a corpus of around 6 billion words from Wikipedia and the last one, the one based on fastText used a 16 billion words dataset from Wikipedia as well. As we will see with the following experiments they provide different results, but despite being trained over different corpuses following different approaches they all are capable of depict the reality of the human world through their embeddings. The implementation differences between these models will be discussed later.

To download the models and store them as python objects we run the following code 3.2:

```
1 # Download the models
2 w2v = api.load('word2vec-google-news-300')
3 glove = api.load('glove-wiki-gigaword-300')
4 fasttext = api.load('fasttext-wiki-news-subwords-300')
```

✓ 4m 48.3s

[=====] 100.0% 376.1/376.1MB downloaded

Python

Figure 3.2: Download models from Gensim API

Now that we have downloaded all of the models we can start running experiments with them in order to understand how they work inside. The most basic operation we can perform to this models is retrieve their information, trying to see what they contain. As we already know these models are n-dimensional vectorial spaces where a vocabulary is represented word-by-word as n-dimensional vectors. In this case

¹A list of all available models/datasets within Gensim library can be seen on the following document: <https://github.com/piskvorky/gensim-data/blob/master/README.md>

we can refer to the documentation of Gensim to see that all of the used models contain 300 dimensions, this means any vector in these spaces will have 300 components, one for each dimension.

To get started let's ask the first model (Word2Vec approach) to provide a vector for a word. In this case the word will be 'hello' and we want to see what this word looks like in this word embedding model. We can get this information by using the `get_vector()` built-in function provided by Gensim for any of their models 3.3.

```

1 hello_vector = w2v.get_vector('hello')
2 hello_vector

```

[30] ✓ 0.0s Python

```

... array([-0.05419922,  0.01708984, -0.00527954,  0.33203125, -0.25
          -0.01397705, -0.15039062, -0.265625,  0.01647949,  0.3828125,
          -0.03295898, -0.09716797, -0.16308594, -0.04443359,  0.00946045,
           0.18457031,  0.03637695,  0.16601562,  0.36328125, -0.25585938,
           0.375,      0.171875,  0.21386719, -0.19921875,  0.13085938,
          -0.07275391, -0.02819824,  0.11621094,  0.15332031,  0.09082031,
           0.06787109, -0.0300293, -0.16894531, -0.20800781, -0.03710938,
          -0.22753906,  0.26367188,  0.012146,  0.18359375,  0.31054688,
          -0.10791016, -0.19140625,  0.21582031,  0.13183594, -0.03515625,
           0.18554688, -0.30859375,  0.04785156, -0.10986328,  0.14355469,
          -0.43554688, -0.0378418,  0.10839844,  0.140625, -0.10595703,
           0.26171875, -0.17089844,  0.39453125,  0.12597656, -0.27734375,
          -0.28125,   0.14746094, -0.20996094,  0.02355957,  0.18457031,
           0.00445557, -0.27929688, -0.03637695, -0.29296875,  0.19628906,
           0.20703125,  0.2890625, -0.20507812,  0.06787109, -0.43164062,
          -0.10986328, -0.2578125, -0.02331543,  0.11328125,  0.23144531,
          -0.04418945,  0.10839844, -0.2890625, -0.09521484, -0.10351562,
          -0.0324707,  0.07763672, -0.13378906,  0.22949219,  0.06298828,
           0.08349609,  0.02929688, -0.11474609,  0.00534058, -0.12988281,
           0.02514648,  0.08789062,  0.24511719, -0.11474609, -0.296875,
          -0.59375,   -0.29492188, -0.13378906,  0.27734375, -0.04174805,
           0.11621094,  0.28320312,  0.00241089,  0.13867188, -0.00683594,
          -0.30078125,  0.16210938,  0.01171875, -0.13867188,  0.48828125,
           0.02880859,  0.02416992,  0.04736328,  0.05859375, -0.23828125,
           0.02758789,  0.05981445, -0.03857422,  0.06933594,  0.14941406,
          ...
          -0.06225586, -0.0534668, -0.05664062,  0.18945312,  0.37109375,
          -0.22070312,  0.04638672,  0.02612305, -0.11474609,  0.265625,
          -0.02453613,  0.11083984, -0.02514648, -0.12060547,  0.05297852,
           0.07128906,  0.00063705, -0.36523438, -0.13769531, -0.12890625],
          dtype=float32)

```

Output is truncated. View as a [scrollable element](#) or open in a [text editor](#). Adjust cell output [settings](#)...

Figure 3.3: Vector for 'hello'

As we can see, the vector for the word 'hello' is a list of numbers, there should be 300 of them, as said, one per dimension on the embedding, and this is the representation of the word for our pre-trained Word2Vec word embedding. As we can see in the figure 3.4 the vector format is a numpy ndarray, this is the main data structure of Numpy library of which we already discussed in section 3.1.4.1. These arrays are used because they are much more performant than python lists due to their implementation, this makes performing any kind of mathematical operation much faster and easier when working with these arrays.

A screenshot of a Python terminal window with a dark background. The first line of code is `1 type(hello_vector)`. Below the code, there is a green checkmark, the text `0.0s`, and the output `numpy.ndarray`. The word `Python` is visible in the top right corner of the terminal window.

```
1 type(hello_vector)
✓ 0.0s
numpy.ndarray
```

Figure 3.4: Vector type

In figure 3.5 we can see the dimensions of this array or vector, it is a 300 element array, and as we already know each one of these contains the value of the vector in the respective dimension.

A screenshot of a Python terminal window with a dark background. The first line of code is `1 hello_vector.shape`. Below the code, there is a green checkmark, the text `0.0s`, and the output `(300,)`. The word `Python` is visible in the top right corner of the terminal window.

```
1 hello_vector.shape
✓ 0.0s
(300,)
```

Figure 3.5: Vector shape

This is what embedding actually look like, these dense vectors are handled as arrays in a computer and the dimensions are represented as positions in those arrays. This creates a simple way of storing and using a lot of information, and we will now see how to do this.

Once we are able to get the representation of the words in a numerical format we can start performing any kind of operation with this data to extract information as mentioned in section 2. The first operation we can perform consists of measuring distance between two vectors, this way we can determine their similarity or relatedness. We already know that since the training of these models is done in order to group similar meanings, words that are related should be closer to each other than words that don't have such a relationship.

To calculate the relatedness of two vectors we can use another function provided by Gensim with every model, the function is `similarity()` and it calculates cosine similarity between two given words. The theory of cosine similarity has been already studied in section 2.2.2, it is the cosine of the angle between two normalized vectors (and it ranges from -1 to 1, being 1 the score for two vectors that are actually the same vector). The used function receives two words as python strings and returns their cosine similarity. In order to do that it first gets the corresponding vector representation for each word and then, since all the vectors inside this space have the same dimensionality (300 dimensions in this case), it is able to compute cosine similarity between the two obtained vectors. We will use cosine similarity since it is the default metric for embedding distance within almost any NLP project, but any of the previously discussed measures would be valid and could be implemented.

With all of this information, the following experiment after being able to get the vector associated to a word within an embedding model will be calculating similarity between words and comparing them to see if these embedding work as intended. In the first place we will compute the similarity between the words 'dog' and 'cat' (Figure 3.6) two very similar words when we look at their meaning, both of them are animals, of similar sizes and features such as being very common pets for humans, so their meaning should be very close to each other. In the other hand we will calculate the similarity between the words 'dog' and 'code' (Figure 3.7), these words do not belong to the same semantic field nor share any of their meaning, they are unrelated words which meaning should be far from each other.



```
1 w2v.similarity('dog', 'cat')
✓ 0.0s
0.76094574
```

Python

Figure 3.6: Cosine similarity between 'dog' and 'cat'



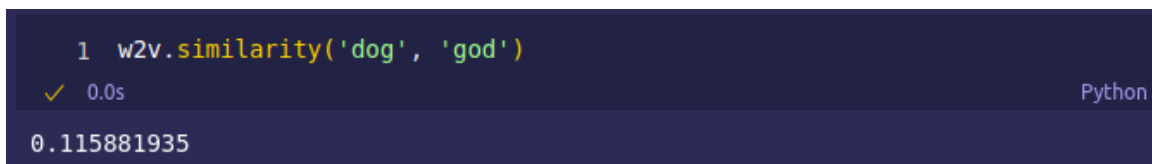
```
1 w2v.similarity('dog', 'code')
✓ 0.0s
0.13505246
```

Python

Figure 3.7: Cosine similarity between 'dog' and 'code'

By looking at the figures 3.6 and 3.7 we can see that we have a very good picture of human reality where 'dog' and 'cat' are very similar words which share a great portion of their meaning with a cosine similarity of 0.76 (very close to 1, this means their vectors are very similar since 1 is the maximum value for this measure as explained above). And we can also see that the similarity between 'dog' and 'code' is as low as 0.13 (being 0 the value for two orthogonal or independent vectors) because those words do not share any kind of meaning in the human vocabulary.

We will now perform a quick experiment to check that this similarity has nothing to do with the character that compose the words but it is all about semantics. We will check the similarity between 'dog' and 'god', two words that share all their characters but none of their meaning, this should return a similarity value similar to the one between 'dog' and 'code'.



```
1 w2v.similarity('dog', 'god')
✓ 0.0s
0.115881935
```

Python

Figure 3.8: Cosine similarity between 'dog' and 'god'

As we can see in figure 3.8 these words scored a very low similarity and that is what we expected since they share no semantics.

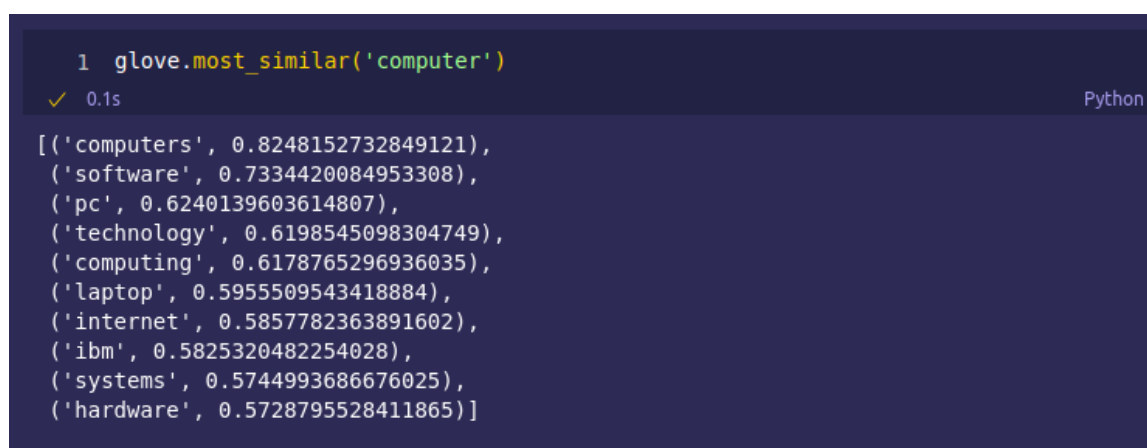
Until now we have been using only one of the models we downloaded, but all of them were trained on a big-enough corpus to be able to provide good embeddings. For the next experiments we will use the model that was trained Using a GloVe approach to see how it performs.

Another operation we can perform with embeddings is finding the most related words to one of our choice, this means we would input one word and we would be able to see what words are close to it in the model space. Gensim's models also provide a built-in function that allows us to perform this search very easily. The function is `most_similar()` and it takes a word (in the format of a python string) as input and then lists the top ten most similar words within the vocabulary to the input word. This function is based on the previous one, `most_similar()` calculates cosine similarity between input word and any other word within the vocabulary to later sort them from highest to lowest cosine similarity, by doing this we can see which words the model understand are more related to any word we want.

In this case we will search for the top ten words that are related to 'computer' 3.9, and as mentioned above we will perform this operation over the model based on GloVe instead of the one based on Word2Vec

as we have been doing. This is only to test different models, trained with different architectures over different datasets but the results are expected to be accurate with all of them. The differences between these architectures, while existing, are minimal and the main concepts remain the same. However we already discussed them in sections 2.4.4 and 2.4.5. For now the important things to keep in mind are that all of these models are trained based on context of the words and that they are all built to perform the same task, represent words' meaning as dense vectors inside multi-dimensional spaces.

Knowing this let's run the `most_similar()` function on GloVe embedding passing 'computer' as argument. What we would expect is obtaining ten words from the semantic field of computing, all related to 'computer' since we will obtain the words that the model understood (during training over a huge corpus) are most related to 'computer' based on the context these words appear.



```
1 glove.most_similar('computer')
✓ 0.1s Python

[('computers', 0.8248152732849121),
 ('software', 0.7334420084953308),
 ('pc', 0.6240139603614807),
 ('technology', 0.6198545098304749),
 ('computing', 0.6178765296936035),
 ('laptop', 0.5955509543418884),
 ('internet', 0.5857782363891602),
 ('ibm', 0.5825320482254028),
 ('systems', 0.5744993686676025),
 ('hardware', 0.5728795528411865)]
```

Figure 3.9: Top 10 most similar words to 'computer'

As we can see in figure 3.9 the the closest vectors to the vector that represents the word 'computer' are all vectors for words related to the computing field. The most similar word to 'computer' is 'computers', and it is no surprise since they are almost the same word, all other words are all related and they range from synonyms such as 'pc' or 'laptop' to related companies such as 'ibm' (IBM has historically been one of the most well-known companies in the computing sector). Next to every related word we can see the cosine similarity that words has with the input word 'computer'. The highest score is 0.82 for 'computers' so we can assume that is a very high cosine similarity and we can expect any pair of words that achieve a score that high to be little less than synonyms. We can also see that the 10th top word related to 'computer' is hardware with a score of 0.57 so we can also expect pairs of words scoring cosine similarities around 0.5 to be very similar in meaning. This provided us with a scale so that we became able to draw conclusions or extract information from these scores.

Examining word embeddings brings an interesting perspective as we observe the proximity of words within the generated embedding space. It's not just about finding synonyms but understanding the subtle semantic connections and contextual similarities between words. This aspect adds a practical layer to the study, allowing us to visually grasp how words relate to each other in a meaningful way. Exploring these relationships sheds light on the nuanced structure of language, making the investigation of word proximity a straightforward yet insightful exploration in the field of natural language processing.

But we are still missing one of the most amazing operations we can perform over embeddings. When we were studying the theory about word embeddings on section 2.2.1.4 we talked about how these methods of text representation not only could relate similar words (based on semantics), but also they were able to capture semantic relationships within human language, allowing the model to represent words in a space

where the relationship between the vectors for 'Spain' and 'Madrid' was similar to the one between 'Italy' and 'Rome'. The same happens with any kind of semantic relationships such as male-female relationships, in this case the relationship between 'man' and 'woman' should be similar to the one between 'king' and 'queen'. With this example we talked about how we could implement different algebras in these vectorial spaces. The exact example we used to enlighten this was the following:

Being $\vec{V}(x)$ the n-dimensional dense vector representing the word 'x':

$$\vec{V}(\text{king}) - \vec{V}(\text{man}) + \vec{V}(\text{woman}) = \vec{V}(\text{queen})$$

This means that if we could subtract the meaning of 'man' to 'king' and then add the meaning of 'woman' we would obtain the vector of the word 'queen'. This can be visualized in a 2 dimensional space if we look at the diagram 3.10.

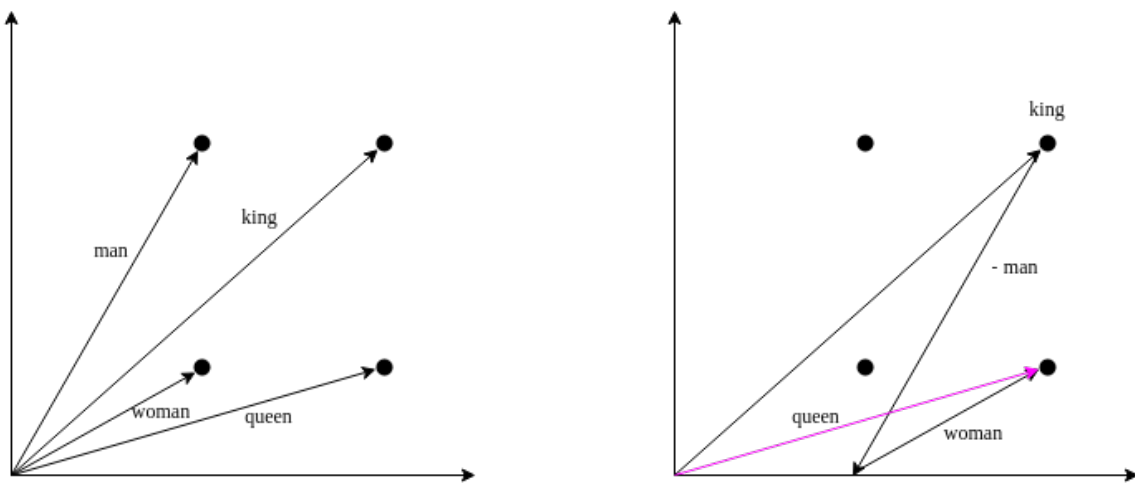


Figure 3.10: 2D embedding algebra

Here we can see how within an ideal space the model was able to capture this semantic relationship of genre. We can observe that adding the vectors of 'king' and 'woman' and the negative vector of 'man' (labeled as '- man' in the diagram) we obtain the same vector that was used for 'queen' before. If our coded implementation of word embeddings achieve similar results, this would mean it can effectively capture some semantic relationships.

The library we are using to experiment with these pre-trained word embedding models (Gensim) provides us with the possibility of performing these operations out of the box, this means we don't need to implement the code to add or subtract vectors. The way of doing it is by using two lists of words, instead of just one word. The first list called 'positive' will contain the vectors which meaning we want to add together, and the second one called 'negative' would contain all of the words which vectors we want to subtract from the other ones. In our example ($V(\text{king}) - V(\text{man}) + V(\text{woman}) = V(\text{queen})$) the positive list would contain two words: 'king' and 'woman', and the negative list would only contain 'man' which is the word which vector we want to subtract (subtracting a vector, as with numbers, is the same as adding it's negative form).

Let's now see how this looks on code, we will add 'king' and 'woman', then subtract 'man' and we will input this expression as input parameter for the function previously used `most_similar()`, in this case we will perform this operation over the Word2Vec based model first and then over the other models

to compare results. What we expect is obtaining a vector that is pretty similar to 'queen' as explained above, the results can be seen in figure 3.11.

```
1 w2v.most_similar(positive=['woman', 'king'], negative=['man'])
```

Python

```
[('queen', 0.7118193507194519),  
 ('monarch', 0.6189674735069275),  
 ('princess', 0.5902431011199951),  
 ('crown_prince', 0.5499460697174072),  
 ('prince', 0.5377322435379028),  
 ('kings', 0.5236844420433044),  
 ('Queen_Consort', 0.5235945582389832),  
 ('queens', 0.5181134939193726),  
 ('sultan', 0.5098593235015869),  
 ('monarchy', 0.5087411403656006)]
```

Figure 3.11: $V(\text{king}) - V(\text{man}) + V(\text{woman}) = V(\text{queen})$ (within Word2Vec based model's space)

As we expected, the resulting vector of adding the meaning of 'king' and 'woman' and subtracting the meaning of 'man' is a new vector whose nearest vector in the 300-dimensional embedding space is the vector for 'queen'. The similarity that the vector resulting of our operation scored with the vector for 'queen' is not equal to 1, which would be the score in our imaginary 2D example, this is because that result could only happen within a mathematically ideal embedding space where semantic relationships are perfectly understood by the model. Unfortunately that is not how things work in the real world, but we can observe a very good behavior by this pre-trained model, which is capable of understanding that the closest word to a king that is not a man but a woman is 'queen'. We can also see that the second nearest word to the vector obtained with the mentioned addition and subtraction is 'monarch' which is not far from reality and the third one is 'princess' which also has a very related meaning to the input vector.

In addition the same procedure has been applied over the GloVe and fastText based embeddings obtaining similar results. These results are showed in figure 3.12 for GloVe:

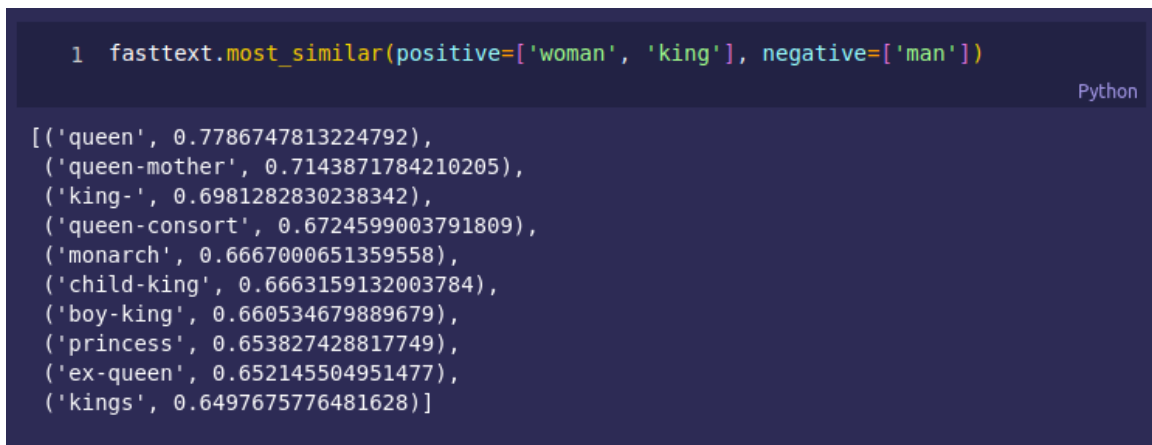
```
1 glove.most_similar(positive=['woman', 'king'], negative=['man'])
```

Python

```
[('queen', 0.7698541283607483),  
 ('monarch', 0.6843380928039551),  
 ('throne', 0.6755736470222473),  
 ('daughter', 0.6594555974006653),  
 ('princess', 0.6520534157752991),  
 ('prince', 0.6517033576965332),  
 ('elizabeth', 0.6464518308639526),  
 ('mother', 0.631171703338623),  
 ('emperor', 0.6106470823287964),  
 ('wife', 0.6098655462265015)]
```

Figure 3.12: $V(\text{king}) - V(\text{man}) + V(\text{woman}) = V(\text{queen})$ (within GloVe based model's space)

And in figure 3.13 for fastText:



```
1 fasttext.most_similar(positive=['woman', 'king'], negative=['man'])

[('queen', 0.7786747813224792),
 ('queen-mother', 0.7143871784210205),
 ('king-', 0.6981282830238342),
 ('queen-consort', 0.6724599003791809),
 ('monarch', 0.6667000651359558),
 ('child-king', 0.6663159132003784),
 ('boy-king', 0.660534679889679),
 ('princess', 0.653827428817749),
 ('ex-queen', 0.652145504951477),
 ('kings', 0.6497675776481628)]
```

Figure 3.13: $V(\text{king}) - V(\text{man}) + V(\text{woman}) = V(\text{queen})$ (within fastText based model's space)

As we can observe all of the models provide very similar and accurate results this is due to the basic concepts applied to their building being the same, the minor differences between these models have an impact on the final performance of the models, making them more flexible or accurate (as mentioned, we will study these details later) but they are still capable of performing the same tasks performing as expected. For all three of the models the closest word to the input vector was 'queen', and the following words in the list are all very related in their semantics. We can assume that different pre-trained word embedding models are capable of capturing semantic relationships between human words through a process of training over an extensive corpus of natural language texts.

These were the last experiments that will be carried out over these three pre-trained word embedding models. We have been able to understand the basics of the embeddings coded implementation in python, as well as understanding from a practical perspective what embeddings actually are.

To sum up, this section provided a practical examination of pre-trained word embeddings, aiming to demystify their essence and decode their implementation. Through various operations, we explored these models to understand their structure and the possibilities they offer. The insights gained in this section will serve as a groundwork for the subsequent sections, where we will delve into more specific questions and challenges within the [NLP](#) field. This exploration contributes to a practical understanding of word embeddings and their applications in computational linguistics which will be extended in the following sections.

3.3. Building an English model

Moving forward, we transition into a new section where our focus shifts from exploring pre-trained word embeddings to hands-on implementation. In this segment, we embark on the practical aspect of constructing a word embedding model from scratch. Unlike the previous section, where we examined existing embeddings, here, we will now craft our own model. We aim to shed light on the process of creating word embeddings, exploring the underlying mechanisms and decisions involved in the construction of such models. This hands-on approach not only enhances our comprehension of the theoretical foundations but also equips us with the practical skills to design and implement customized word embeddings.

This section will be used as a test case for the next one where we will build a model with a collection of Middle High German texts [3.4](#)

3.3.1. Obtaining data

To be able to create our own word embedding models we first need data. As we know machine learning algorithms need data to learn from it, so we will start by obtaining data. In this case we decided using Kaggle[29], one of the most well known web sites when we talk about data science in general and machine learning in specific. This website provides many tools for anyone interested in this field, ranging from datasets, to trained machine learning models, courses and they even host competitions with big money prizes. In this case we headed to the dataset section in order to find an English text corpus. The one we will be using is a dump from English Wikipedia made in 2018, which contains 7.8 million sentences². This is a big enough dataset to create a word embedding model that is able to represent semantic relationships.

3.3.2. Exploring data

In this case we have chosen a dataset that is easy to process, the format is easily readable withing a few python lines of code. The dataset is contained in one text file (’.txt’) in which each line is a sentence. We can see part of it in figure 3.14.

```

1957555 He is mainly renowned for being part of the supergroup Notting Hillbillies along with the Dire Straits frontman Mark Knopfler and Brendan Croker.
1957556 He is mainly responsible for the logistics, like transport, training facilities, hotel bookings and all other similar tasks.
1957557 He is mainly seen on BBC World News and is the main presenter of its flagship cinema program Talking Movies.
1957558 He is mainly specializing in hill climbing, taking part also to the European Hill Climb Championship .
1957559 He is Major League Soccer's third-all-time leader in regular-season goals scored with 134.
1957560 He is majorly influenced by the bands - Nirvana, Incubus, Chevelle and Rage Against the Machine.
1957561 He is making a biographical film on Pandit Gopabandhu Das, Orissa's renowned social worker, educationist and politician.
1957562 He is making an attempt to bring bandy to the Philippines.
1957563 He is making his directorial debut with the upcoming film Love Action Drama starring Mivvin Pauly and Nayanthara.
1957564 He is making his feature film directorial debut with the upcoming Malayalam film Odiyan, to be released in 2018.
1957565 He is making his Muscle Mania, MM Pro debut at MM World in Las Vegas this year.
1957566 He is making it clear in the song that he feels that there is no justice in the world because of this fact.
1957567 He is making small interventions in space.
1957568 He is Malay by ethnicity and an adherent of Sunni Islam, both of which are prerequisites for Malaysian royalty.
1957569 He is Maltese, born in Pietà, Malta in 1965, and has worked in Malta, Monaco, UK and the United States.
1957570 He is managed by ABS-CBN's Star Magic talents.
1957571 He is managed by Al Haymon.
1957572 He is managed by David Naylor & Associates and currently resides in Los Angeles, California.
1957573 He is managed by Federico Rosa.
1957574 He is managed by Gianni De Madonna.
1957575 He is managed by Gianni Demadonna and coached by Francis Songol.
1957576 He is managed by GPS management.
1957577 He is managed by James Templeton and coached by Colm O'Connell.
1957578 He is managed by Ricky Simms.
1957579 He is managed by the legendary Cameron Dunkin, and Mike Altamura.
1957580 He is managed by world heavyweight champion Anthony Joshua.
1957581 He is managed under Sony Music BMG Malaysia.
1957582 He is manager of EFL League Two club Carlisle United.
1957583 He is manager of non-league Eccleshill United.
1957584 He is Manager of Outdoor Press, and a farmer on the property known as Granite Springs, Kelvin View, Victoria.
1957585 He is manager of the Norwegian National Opera and Ballet from 2008.
1957586 He is managing director of Aquafin, a Flemish wastewater processing company.
1957587 He is managing director of Century Films, a London-based independent film and television company.
1957588 He is managing director of Computer NEXAT, which he created in 2003 after twenty years at SIR and as a head of IT in education.
1957589 He is Managing Director of Dizzijay, founder of NOCCI, and is co-organizer of Ignite Cardiff and TEDxCardiff.
1957590 He is managing director of Doha-based QALCO.
1957591 He is managing director of drama production company Lovely Day.
1957592 He is managing director of Durgish Films Pvt.
1957593 He is Managing Director of GIGAD, a research network with locations in China, the United States, Brazil and Europe, and professor at universities in Europe and China.
1957594 He is managing director of Mata Raj Kaur Institute of Engineering & Technology, Rewari, Haryana.
1957595 He is managing director of Mercury, a bipartisan public strategy firm.
1957596 He is Managing Director of ResourcesLaw International, and Executive Director of the Energy Policy Institute of Australia.
1957597 He is managing director of Soho Media Group, a production and marketing company.
1957598 He is Managing Director of the Independent TV production company, Possessed.
1957599 He is managing director of Value Partners, the management consulting multinational he founded in 1993.
1957600 He is managing director of Vera Productions, an independent production company he set up with Bremner.
1957601 He is Managing Director of VIB and Board Member of FlandersBio.
1957602 He is managing director of VisionValue, a consultancy firm running projects for multinationals and Public Institutions amongst which the European Commission.
1957603 He is Managing Editor of BBC Radio Leeds.
1957604 He is managing editor of the Forward Press monthly magazine.
1957605 He is managing editor of the Journal of Economic Perspectives, a quarterly academic journal produced at Macalester College and published by the American Economic Association.
1957606 He is Managing Member of Hanlen Real Estate Development & Funding LLC, and Hanlen Healthcare Development & Funding LLC.
1957607 He is Managing Partner and founder of the investment firm Valor Capital Group in New York and Brazil.
1957608 He is Managing Partner at Fifth Era and Helixetu Capital based in the San Francisco Bay Area.
1957609 He is Managing Partner of Applied Video Compression.
1957610 He is managing partner of BoxGroup, a seed-stage capital firm, and a co-founder of TechStars New York City.
1957611 He is managing partner of Hendricks Sports Management, L.P., and managing member of Hendricks Interests LLC, both in Houston, Texas.

```

Figure 3.14: Sample sentences from English dataset

As we can see they are all English sentences, and each one in a new line of the file, knowing this we can write a python script to read and process these sentences, in order to obtain optimal data to feed to the model.

3.3.3. Processing data

As we have seen, this data is already processed, but in order to feed it to the model and get good results we will use some of the previously studied text processing techniques.

²The dataset can be found in: <https://www.kaggle.com/datasets/mikeortman/wikipedia-sentences/data>

To get started with the code we will import the NLTK[28] library, as well as RE library, which stands for regular expression. This will allow us to use some of the NLTK tools, such as a lemmatizer and a list of stopwords, and also use regular expressions, which is a very helpful tool when processing text. Figure 3.15.

```
1 import nltk
2 import re
✓ 1.2s Python
```

Figure 3.15: Import NLTK and RE

Next we will load our stopwords. There are 179 of them in this case and some of them are words like 'or', 'is', 'do' or 'a'. Figure 3.16.

```
1 stop_words = set(nltk.corpus.stopwords.words('english'))
2 print("Length of stopwords: ", len(stop_words))
✓ 0.0s Python
Length of stopwords: 179
```

Figure 3.16: Load stopwords

Now we load the lemmatizer, and we can see an example of the result of using it with the word 'mice' which is converted into it's base form 'mouse' in figure 3.17

```
1 lemmatizer = nltk.stem.WordNetLemmatizer()
2 lemmatizer.lemmatize("mice")
✓ 0.8s Python
'mouse'
```

Figure 3.17: Load and test lemmatizer

Now that we have these tools ready to use we will start by coding a very simple data cleaning function which we will then apply to the data extracted from the text file.

In the first place we defined the function `remove_symbol` (figure 3.18) which removes any character that is not an a-z letter, the function does this using Regular Expression (RE)s and replacing any character that matches the RE `[^a-zA-Z]`³ with a space to later remove any extra spaces by replacing " +" (one or more spaces) with only one space. This means that any word containing a character that is not a letter, such as numbers or special characters, will be turned into a space.

```
1 def remove_symbol(string):
2     # Removing everything except a-z english letters
3     string = re.sub('[^a-zA-Z]', ' ', string)
4
5     # Removing extra spaces
6     clean_string = re.sub(' +', ' ', string)
7
8     return clean_string
✓ 0.0s Python
```

Figure 3.18: Function to remove anything that is not a-z characters

³This RE would match any string that contains a non-letter

Now we define the function that will perform all the text pre-processing, by reading all the sentences in a list and returning a new list with clean sentences tokenized into words. The text processing techniques applied are: tokenization to split every sentence into tokens (words), special character removing, lowercasing, lemmatizing and removing stopwords, this function can be seen in figure 3.19

```

1 def data_cleaning(data):
2     sentences = []
3     for index in range(len(data)):
4
5         for sentence in data[index]:
6
7             # Removing empty lines
8             if sentence != "":
9                 clean_sentence = []
10                # Removing anything that is not a-z characters and lowering
11                sentence = remove_symbol(sentence.lower())
12
13                # Removing stopwords and applying lemmatization
14                for word in sentence.split():
15                    if word not in stop_words:
16                        clean_sentence.append(lemmatizer.lemmatize(word))
17
18                sentences.append(clean_sentence)
19
20 return sentences

```

Python

Figure 3.19: Function to clean data

Now that we have our text processing ready it is time to extract data from the text file, and load into our coding environment, we can do this easily in Python by opening the file and using the built-in function `read()` to extract all text from it, then we use a sentence tokenizer provided by NLTK library to divide the whole text into sentences, divided by line jumps, identified as `'\n'`. In figure 3.20 we can see how this is done.

```

1 with open('data/wikisent2.txt') as file:
2     data = file.read()
3
4     tokenized_data = nltk.sent_tokenize(data)
5     print("Total number of sentences: ", len(tokenized_data))

```

✓ 2m 18.7s

Total number of sentences: 7963525

Python

Figure 3.20: Extraction of sentences from text file

Once we have a list containing all our sentences we can use our function to process them, and obtain a list of lists of words, this is a big list containing all the sentences, which are lists of words. In figures 3.21 and 3.22 we can see a test of the cleaning function, where we read a sentence from the dataset and the process it, as we can see it works as expected, tokenizing the sentence into words, removing stopwords, lowercasing and removing special characters such as the period at the end of the sentence.

```

1 print(tokenized_data[216757])

```

✓ 0.0s

Ajitpal Mangat is an Indian film director.

Python

Figure 3.21: Sentence number 216.757 after reading from text file

```

1 sample_sent = data_cleaning((tokenized_data[216757], ""))
2 print(sample_sent)
✓ 0.0s Python
[['ajitpal', 'mangat', 'indian', 'film', 'director']]

```

Figure 3.22: Sentence number 216.757 after processing it

```

1 print(sentences[4853720])
✓ 0.0s Python
['ron', 'atanasio', 'american', 'former', 'professional', 'soccer', 'player', 'played', 'forward']

```

Figure 3.23: Sentence number 4.853.720 after processing all the data

Ron Atanasio

[Article](#) [Talk](#)

From Wikipedia, the free encyclopedia

Ron Atanasio is an American former professional soccer player who played as a forward.

Figure 3.24: Sentence number 4.853.720 in Wikipedia

Then we will process all the data and pick another random number to check again the result. As we can see in figure 3.23 it is correct, and actually as a curiosity, we can still find this sentence in Wikipedia if we look for the name of the soccer player 3.24 ⁴.

Now that we have successfully prepared the data to be in a valid format to train a word embedding model, we will jump into the training, which will actually be very simple by taking advantage of Gensim, a very powerful library which will let us train a model in a very easy way.

3.3.4. Training the model

In this subsection we will showcase the training of the model and we will run some tests to check that the model works as expected, this way we will be able to confirm that the process of building a custom language model was done correctly and therefore we are ready to extrapolate the process to another dataset, which will be a Middle High German set of texts.

```

1 from gensim.models import Word2Vec
Python

1 model = Word2Vec(sentences=sentences, vector_size=200, window=4, min_count=1, workers=8)
Python

```

Figure 3.25: Training function from Gensim library

⁴Wikipedia page of the sentence: https://en.wikipedia.org/wiki/Ron_Atanasio

To begin with the training we will import a module of the Gensim library, which is `models.Word2Vec` and will provide a framework to easily train word embeddings with a Word2Vec approach [\[30\]](#). It's online documentation can be found at [\[30\]](#). Then by calling the function with our processed data as a parameter we will start the training process without the need of implementing any neural network architecture or training algorithm by ourselves. This is a very convenient way to train word embeddings, although it is less customizable than a custom implementation, but for testing purposes it is more than enough. There are also other parameters, such as the desired embedding size, or the window size, as well as an option to use multiple cores of the processor to make the training process faster.

```
1 model.save("data/model/custom_model.model")
```

Python

Figure 3.26: Save the model to disk

```
1 model = Word2Vec.load("data/model/custom_model.model")
```

✓ 13.7s

Python

Figure 3.27: Load a model from disk

After the training process is complete we can save our model to be later loaded without the need of re training it, this way the model will not be lost when the execution of the current Python kernel ends.

After having the model in memory we can perform some operation with it, as we saw in section [3.2](#). The figures [3.28](#) [3.29](#) [3.30](#) [3.31](#) show that many operations can be performed on our model and that they make sense.

```
1 model.wv.get_vector('king').shape
```

✓ 0.0s

Python

(200,)

Figure 3.28: Shape of the embedding, it is 200 as specified

```
1 model.wv.most_similar("efficiency", topn=10)
```

✓ 0.6s

Python

```
[('reliability', 0.7650973200798035),  
 ('durability', 0.7424072027206421),  
 ('responsiveness', 0.737628698348999),  
 ('productivity', 0.7375406622886658),  
 ('effectiveness', 0.7288287878036499),  
 ('flexibility', 0.7244429588317871),  
 ('stability', 0.7107709050178528),  
 ('efficient', 0.6971147656440735),  
 ('scalability', 0.6887215375900269),  
 ('accuracy', 0.68172687292099)]
```

Figure 3.29: Top 10 most similar words to 'efficiency'

```
1 model.wv.n_similarity('computer','programming')
✓ 0.0s Python
0.8844001
```

Figure 3.30: Cosine similarity between 'computer' and 'programming' embeddings

```
1 model.wv.doesnt_match(["dog", "cat","monkey","car"])
✓ 0.0s Python
'car'
```

Figure 3.31: Word that doesn't match from 'dog', 'cat', 'monkey' and 'car'

Even though it will be further discussed in chapter 4 after these results we can assume that the process of training a custom embedding over a corpora of text is a feasible task with the tools we have. Knowing this we will now do something new, something that we haven't been able to find before. We will try to obtain a word embedding model for the Middle High German language.

3.4. Building a Middle High German model

3.4.1. Obtaining data

For the training of the [MHG](#) model we will use a dataset provided by Professor Thomas Leek, it is a Middle High German corpus of texts that can be found here [\[31\]](#) and it contains more than one thousand PDF files with text written in this ancient language. With help of Professor Thomas Leek we are going to try and understand the format of this dataset, to be able to work with it.

3.4.2. Exploring data

After comprehensively examining the dataset, we found out that it contains three copies of each PDF file, each one of them is in a different format, these are **Diplomatic** [3.32](#), **Modern** [3.33](#) and **Normalised** [3.34](#).

M001: Ad equum errehet	Diplomatischer Lesetext
M001-66_02,00	Ad equū errehet
M001-66_02,01	Man gieng after wege.
M001-66_02,02	zoh lîn Rof inhandon.
M001-66_02,03	do begagenda imo min trohtin
M001-66_02,04	mit lînero arngrihte.
M001-66_02,05	“wefman gestu
M001-66_02,06	zune rideftu.”
M001-66_02,07	“waz mag ih riten.
M001-66_02,08	min rof iſt errehet.”
M001-66_02,09	“nu ziuſ ez da bifiere.
M001-66_02,10	tu runeimo in daz ora.
M001-66_02,11	drit ez anden cefewen fuoz.
M001-66_02,12	fo wirt imo def erræhtenbûz”
M001-66_02,13	Paī nr̄. & terge crura ei⁹ & pedef dicenf. “alfo ſciero werde
M001-66_02,14	difemo cui⁹cūg colorif ſit. rot. ſuarz. blanc. ualo. grifel. feh.
M001-66_02,15	roffe def erræhtenbûz fâmo demo got dafelbo bûzta.”

Figure 3.32: Diplomatic format fragment of corpora

M001: Ad equum errehet	Modernisierter Lesetext
M001-66_02,00	Ad equum errehet
M001-66_02,01	Man gieng after wege.
M001-66_02,02	zoh sin Ros in handon.
M001-66_02,03	do begagenda imo min trohtin
M001-66_02,04	mit sinero arngrihte.
M001-66_02,05	“wes man ges tu
M001-66_02,06	z u ne rides tu.”
M001-66_02,07	“waz mag ih riten.
M001-66_02,08	min ros ist errehet.”
M001-66_02,09	“nu ziuh ez da bi fiere.
M001-66_02,10	tu rune imo in daz ora.
M001-66_02,11	drit ez an den cesewen fuoz.
M001-66_02,12	so wirt imo des erracheten buoz”
M001-66_02,13	Paten nren. & terge crura eius & pedes dicens. “also sciero werde
M001-66_02,14	disemo cuiuscunq coloris sit. rot. suarz. blanc. ualo. grisel. feh.
M001-66_02,15	rosse des errachoten buoz samo demo got da selbo buozta.”

Figure 3.33: Modern format fragment of corpora

M001: Ad equum errehet	Normalisierter Lesetext
66_02,0 {10r,13}	[!] [!] erræhet.
66_02,1 {10r,13}	man gienc after wege,
66_02,2 {10r,14}	zôch sin ros in handen.
66_02,3 {10r,14}	dô begegente ime mîn truhtîn
66_02,4 {10r,14}	mit sînere êrengrehte:
66_02,5 {10r,15}	“wes man gês dû?
66_02,6 {10r,15}	zuo iu ne rites dû?
66_02,7 {10r,15}	”waz mac ich riten?
66_02,8 {10r,15}	mîn ros ist erræhet.
66_02,9 {10r,16}	“nû ziuch ez dâ bi viere,
66_02,10 {10r,16}	dû rûne ime in daz oere,
66_02,11 {10r,17}	trit ez an den zeswen vuoz;
66_02,12 {10r,17}	sô wirdet ime des erræhet buoz.
66_02,13 {10r,18}	[!] [!], [!] [!] [!] [!] [!] [!] [!]:”alsô schiere werde
66_02,14 {10r,19}	diseme, [!] [!] [!], rôr, swarz, blanc, vale, grisel, vêch,
66_02,15 {10r,20}	rosse des erræheten buoz, same deme got dâ selbe buozte.

Figure 3.34: Normalised fragment of corpora

Following the indications of Professor Thomas Leek we chose the normalised format, since it should provide better consistency to the project. As we can see this format presents a token, which stands for some words in the text, this is because these sentences or fragments of text are not written in **MHG** but Latin, preventing us from introducing incorrect data, or having to manually check for inconsistencies across the corpora, which would be a very tedious task.

After analyzing some of these files, we know the sentences are not in a format where one line means one sentence, instead they should be divided where we find a period. These are very important observations, since they will define how we read the text from the files to convert it into sentences.

3.4.3. Processing data

The idea is the same as with the English corpus, but in this case we need to extract text from a PDF file, which is not as straightforward as with plain text files. Anyways there is a python library that is called *pypdf*, and serves the purpose of extracting text from PDF files, this library was used in this case.

```
1 import pypdf
2 import re
3 import os
```

Figure 3.35: Libraries used to extract and process [MHG](#) text

```
8 # Write each sentence in sentences (all sentences in a page) to txt file
9 def write_to_txt(sentences):
10     f = open(FILENAME, "a")
11     for sentence in sentences:
12         f.write(sentence + "\n")
13     f.close()
```

Figure 3.36: Function that writes any number of sentences to a plain text file

```
15 # Removes first and last lines of a page since they are not actual mhg text
16 def remove_first_and_last_lines(text):
17     ind1 = text.find('\n')
18     ind2 = text.rfind('\n')
19     return text[ind1+1:ind2]
20
```

Figure 3.37: Function that, given the contents in a page, removes first and last lines (metadata)

```
21 # Removes all punctuation from each sentence and fixes spaces
22 def clean_sentences(sentences):
23     clean_sentences = []
24     for sentence in sentences:
25         sent_no_punct = re.sub('[.,:;="\'@#!?!&$\[\]\{\}]+', ' ', sentence)
26         # Remove extra spaces
27         sent_no_extra_spaces = re.sub(' +', ' ', sent_no_punct)
28         clean_sentence = sent_no_extra_spaces.strip()
29         if clean_sentence != '':
30             clean_sentences.append(clean_sentence)
31
32     return clean_sentences
33
```

Figure 3.38: Function that processes each sentence removing special characters and extra spaces

```

34 def main():
35     directory = '/home/sergio/Documents/TFG/mhg-lesetexte'
36     for file in os.listdir(directory):
37         # Create pdf reader for file
38         filename = os.path.join(directory, file)
39         reader = PyPDF2.PdfReader(filename)
40         text = ""
41         # Extract text from every page
42         for page in reader.pages:
43             page_text = page.extract_text()
44             page_text = remove_first_and_last_lines(page_text)
45             # Add to complete file text
46             page_text = ' '.join(s for s in page_text.split() if not any(c.isdigit() for c in s))
47             text = text + "\n" + page_text
48
49         # Create sentences
50         # Remove line jumps, there should be one per page jump in original document
51         text = text.replace("\n", " ")
52         # Divide sentences by period (".") or period + space (". ")
53         text = text.replace(". ", ".")
54         sentences = text.split(".")
55         sentences = clean_sentences(sentences)
56         write_to_txt(sentences)
57
58
59 if __name__ == '__main__':
60     main()
61

```

Figure 3.39: Main function that uses all of the previous to extract, process, and save the data

As we can see in figures 3.35, 3.36, 3.37, 3.38 and 3.39 this time we used a script that would extract and process all the text from the PDF files, which are not very easy to use because of the extra information, into a text file that contains only the actual MHG sentences. This is useful since now we have a full processed corpus ready to use any number of times without having to extract any information from the PDF files. The process can be seen in the previously referenced figures. The main differences when comparing with the processing of the English text are the following:

- We can't just remove any character that is not an a-z letter, this is because Middle High German contains characters that modern English doesn't. Instead we created a set of symbols or special characters to be removed after examining many of the files.
- In this case the sentences are divided by a period ('.') instead of a new line ('\n').
- We had to remove extra information from the PDF files, this is, the first and last lines of the file (which contain metadata not useful for our purpose) and the information added next to each line of text (which is also not MHG text)

With all of the previous considerations, we iterated through all the PDF files in a directory, which contains the normalised version of each file in the corpus, then for each page in each one of these PDF files, the text was processed and saved into a text file. This text file contains each sentence in a line, and can be later used along with a sentence tokenizer to extract the ready-to-use corpus. We can have a look at the text file in figure 3.40.

```

5912 sines kūnnes ne schol niht mêre wahren in dere erde
5913 die gisele hiez er üz vüeren
5914 die houbete sie in abe sluogen
5915 Genelunen sie bunden mit vuozen unde mit handen wilden rossen zuo den zagelen
5916 durch dorne unde durch hagene an deme bûche unde an deme rûgge brâchen sie in ze stücken
5917 sô wart diu untriuwe geschendet
5918 dâ mite sich daz liet verendet
5919 dô quam Falsaron
5920 von der erden Dathan unde Abiron was er verre gevaren
5921 einen güldinen aren vuorte er ane deme schilte
5922 vor der schare er spilte
5923 von sineme helme dâ schein ein liechter karfunkelstein unde ander werc wæhe
5924 er was rîche und mære
5925 er sprach bist dû hier Olivir mir ist dicke gesaget von dir dû sîst der kristene vorvehtære
5926 ich hân hier guote knehte sehs unde zweinzic tûsent manne
5927 nû rît dû helet under minen vanen
5928 ich hilfe dir gerne hinnen wilt dû helet dîngen dâr zuo deme kûninge Marsilien
5929 dû ne maht iz niht gewideren
5930 behalt lîp unde ruom

```

Figure 3.40: Fragment of the text file containing all the processed sentences

Since I am not able to understand any text written in [MHG](#), these results were discussed with Professor Thomas Leek to make sure these sentences make sense, and ensure the process and the results are correct. Also as we mentioned earlier there was a previous study to understand the structure and format of the files for me to be able to work with them.

3.4.4. Training the model

The process followed to train the model is the same as the one followed with the English dataset, and is shown in figure 3.25. In figure 3.41 we can see that the number of sentences used to train this [MHG](#) model is 50.524, this means in this case we will work with only 50 thousand samples versus the 7.9 million used for the English model, this is a huge issue when working with [ML](#) algorithms and will be discussed later.

```

1 print("\nTotal number of sentences: ", len(sentences))
✓ 0.0s Python
Total number of sentences: 50524

```

Figure 3.41: Number of sentences used to train the [MHG](#) model

Once the word embedding model has been trained we check that we are able to obtain vectors for words, and compare them, and in figure 3.42 we can see that the score for cosine similarity between the words 'Krist' and 'got' ('Christ' and 'god') is approximately 0.8, which makes sense for now.

```

1 model.wv.similarity('Krist','got')
0.80083835 Python

```

Figure 3.42: Cosine similarity between 'Christ' and 'got' within [MHG](#) custom model

In this section we will not study the results, that will be done in chapter 4, but this is a way to check that the training was successful from the technical point of view, demonstrating that it is able to turn words into vectors, compare them and that the generated embeddings are not completely random. Also this is considered a success since the goal of this project was not to train accurate word embedding

models, but studying the state of the [NLP](#) field and the feasibility of applying these methodologies to ancient languages, which don't have nearly as much work on them as modern languages nowadays.

Chapter 4

Results

In this section, we present the results of building two word embedding models, one for the English language, and one of the Middle High German language. Our primary goal was to take a first step towards a tool that would enable us to relate text fragments across different ancient languages in a quick and easy way, with this in mind the first step has been a big research phase which was summarized and explained in chapter 2, this research has provided a great understanding of what it is used to accomplish this goal nowadays and what is not. This way we have been able to select the most useful and basic techniques and study them in depth to have a good understanding of the panorama.

After this study, we came to the conclusion that embeddings were a needed first step to take in the direction of semantic similarity. Then we have jumped into the experimentation, with both pre-built and custom ML word embedding models applying these technologies to real data with help of libraries that made it easy, since they are specifically built for that purpose (Gensim, NLTK).

After we built our own models we have used Cosine Similarity (section 2.2.2) to evaluate its performance and with help of Professor Thomas Leek we have run several tests on MHG pairs of words, comparing its results to those offered by pre-built or even custom models on modern languages (English), the capability of offering high scores for related words, and low scores for unrelated words felt inferior. We will now see some examples and then expose our thoughts on them.

To get started we have seen that the pre-trained models that were trained by experts using huge amounts of data, all performed very similar, offering good results, these can be seen in 3.2. The results we want to discuss are the relationships made by our own models, the ones trained by ourselves, both English and MHG.

For the English model we have seen that it is capable of relating words like 'computer' and 'programming' with a very high score 3.30, it is also capable of providing synonyms of 'efficiency' if we look at the top 10 words with highest similarity to it 3.29, and even identify the word that doesn't match from 'dog', 'cat', 'monkey' and 'car' 3.31. Therefore we can confidently say that the model has been able to learn many relationships extracted from the corpus provided in an effective way, utilizing the described algorithms to learn word representations from the context they appear in across the text.

It is important to mention that the following results were obtained in cooperation with Professor Thomas Leek, and it wouldn't have been possible without him to test, understand and learn from the following results.

Now going with the MHG model we have already seen that it scores 'Krist' (Christ) and 'got' (god) at a 0.8 using cosine similarity between their respective embeddings 3.42, this can make sense but we will

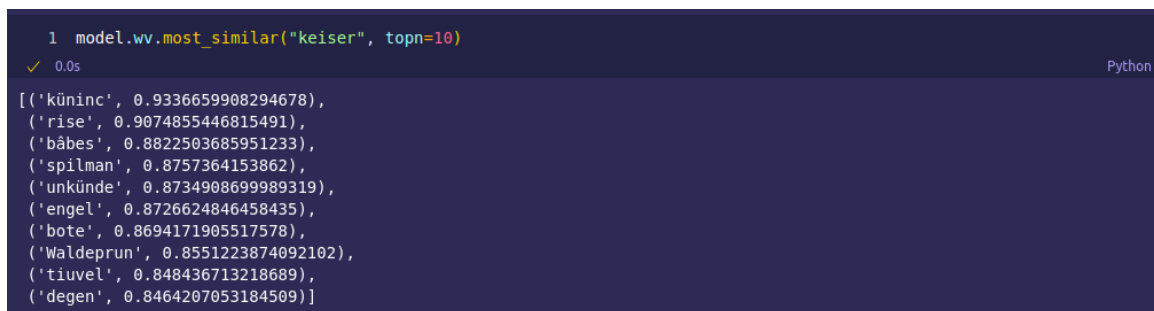
see other examples. For example in figure 4.1 we can see that the similarity score for the words 'hêrre' (lord) and 'got' (god) is very low, and we didn't expect it to be low, instead we expected these words to be scored as synonyms.



```
1 model.wv.similarity('hêrre','got')
✓ 0.0s
0.6453244
```

Figure 4.1: Cosine similarity between 'hêrre' and 'got' within MHG custom model

This is an unexpected behaviour and it doesn't seem accurate. But if we take a look at other examples we can find relationships that may not seem accurate at all for us, but if we take a historical and linguistic perspective we could look at it from a different angle. In figure 4.2 we can see that among the most similar words to the word 'keiser' (emperor), we found words like 'spilman' (musician/poet), which for us won't make sense, but if we think about the context of these text, the time and place they were written and read, we can see that they both can appear in the context of songs, e.g, the 'spilman' sings about the 'keiser'. We can also find words related to religion, such as 'engel' (angel) or 'tiuvel' (devil/demon), this can be due to the emperor being much closer to the heaven in pre-modern Catholic thinking than it is today for us.



```
1 model.wv.most_similar("keiser", topn=10)
✓ 0.0s
[('kūninc', 0.9336659908294678),
 ('rise', 0.9074855446815491),
 ('bābes', 0.8822503685951233),
 ('spilman', 0.8757364153862),
 ('unkūnde', 0.8734908699989319),
 ('engel', 0.8726624846458435),
 ('bote', 0.8694171905517578),
 ('Waldeprun', 0.8551223874092102),
 ('tiuvel', 0.848436713218689),
 ('degen', 0.8464207053184509)]
```

Figure 4.2: Ten most similar words to 'keiser' within MHG custom model

We have also found word relationships that are interesting and show a certain degree of 'understanding' such as the words 'hunger' (hunger) and 'durste' (thirst) scoring a 0.9, or 'arzât' (doctor) being the third closest word to 'jude' (Jew).

Since we have found many accurate relationships and some that seem like they are not, we understand that the model needs improvement, refinement and optimization. In the following section 5.2 we will present our thoughts on the continuation of the project.

In conclusion, we have been able to find that our MHG model is, to a certain extent, capable of capturing relationships between words, even though there are results that we can't explain. Therefore we have accomplished the initial goal of successfully applying NLP modern techniques to Ancient Languages in order to evaluate semantic similarity.

Chapter 5

Conclusion and future directions

After all the research, study and experimentation this last section aims to tie together all the elements on this document and extracting the necessary conclusions from all the work that was done. This way we will have a comprehensive understanding of what was done, what was found and what learning we can obtain from it.

This section aims to serve several important purposes including the following:

- **Summarize Key Findings:** This section should provides a concise summary of the main findings and results of the research. This includes highlighting the significant points that were discussed in the document and providing a complete idea linking all of them together. This ensures we finish with an idea of the whole document, having complete understanding of the different topics and how they relate to each other.
- **Reflect on the Research Objectives:** This section connects the findings back to the original objectives and aims of the study, showing how the research has met these goals and contributed to the understanding of the topic.
- **Discuss the Implications:** We will also elaborate on the broader implications of the findings. Including theoretical implications, practical applications, and the potential impact on the field of study.
- **Identify Limitations:** One of the goals of this section is to be able to acknowledge the limitations of the work done. In this case, the coded implementations provided serve the purpose of experimenting with the algorithms and measuring feasibility of the desired tool, they are not and don't claim to be final, optimized or usable results in any way. Therefore the limitations here will be many, this sections aims to identify and acknowledge these weaknesses providing our thoughts on the steps or tools necessary to address such limitations.
- **Suggest Future Research:** As mentioned before, one of the main goals of the following section is to provide recommendations for future research based on the findings and limitations of the current study. This should helps to guide subsequent studies and indicate areas where further investigation is needed, as well as encouraging the continuation of the work. This is specially important because, as we already know, this project is just a part of a bigger one, a first step to dilucidate the panorama and explore what paths could guide to a bigger goal, providing a solid knowledge base on which to continue the work.

- **Providing a final thought:** We will end with a concluding subsection providing a final thought about the whole document.

5.1. Conclusions

We will start by going over different topics to explain what the conclusions of the realization of this project are.

5.1.1. Feasibility and Challenges

As emphasized throughout this document, the main goal of this project was to lay a solid foundation for future work. Since this is a preliminary study, our primary aim was to understand how to begin working in this field, identify key challenges, and evaluate the feasibility of the project.

To accomplish this, we thoroughly explored the current technological landscape and assessed how modern natural language processing techniques could be applied to the study of ancient languages. We also conducted some experimentation to be able to study the feasibility of the future work and to identify practical challenges.

By addressing these foundational aspects, this project aims to provide a clear starting point for future research and development. Our goal was to ensure that subsequent efforts are based on a solid theoretical and methodological framework, paving the way for the successful creation of a tools capable of identifying related text fragments across different ancient languages.

5.1.1.1. Feasibility Study

The initial phase of our project was a comprehensive study aimed at identifying the most effective approach for analyzing semantic similarities in ancient texts. Through this investigation, we evaluated various methodologies and techniques, accounting for their applicability to low-resource languages like Middle High German. Our findings clearly indicated that word embeddings, which capture semantic relationships between words by representing them in a continuous vector space, were particularly well-suited for this task. These embeddings provide a robust framework for evaluating semantic similarity, even with limited data, making them the optimal choice for developing tools to identify and analyze related text fragments in ancient languages.

5.1.1.1.1. Training a Word Embedding Model on Middle High German Corpora was one of they key components to assess the feasibility of this project. Middle High German, being a low-resource language, presents unique challenges due to the limited availability of digitized texts and the linguistic complexity of the corpus. Despite these obstacles, we successfully trained a word embedding model specifically tailored to this ancient language.

We utilized a substantial corpus of Middle High German texts, (which included literary works, legal documents, and religious texts)

By employing state-of-the-art techniques such as Word2Vec, we were able to generate meaningful embeddings that captured the semantic relationships between words within the corpus. These embeddings serve as the foundational layer for evaluating semantic similarity between text fragments.

5.1.1.1.2. Collaboration with Professor Thomas Leek from the University of Wisconsin-Stevens Point was instrumental in evaluating the effectiveness of the trained word embedding model. Professor Leek's expertise in historical linguistics and ancient languages provided valuable insights into the specific linguistic features and nuances of Middle High German.

Together, we conducted a series of evaluations to test the model's performance. This involved:

- **Qualitative Analysis:** We performed a qualitative analysis of the embeddings by examining the nearest neighbors of selected words and assessing the semantic coherence of these relationships. Professor Leek provided expert feedback on the linguistic accuracy of these associations.
- **Quantitative Evaluation:** We employed standard metrics to quantitatively evaluate the model's performance. This included intrinsic evaluations such as cosine similarity scores between word pairs.
- **Comparative Studies:** We compared the performance of our Middle High German embeddings with those trained on modern languages to understand the impact of the unique characteristics of ancient texts on the model's effectiveness.

The results of these evaluations were promising. The word embedding model demonstrated some degree of semantic accuracy, effectively capturing several relationships between words in the Middle High German corpus. This success highlights the potential of adapting modern natural language processing techniques to ancient languages, even with the inherent challenges of limited data and linguistic complexity.

It is worth mentioning that the project (including python scripts and trained models) are currently hosted in a GitHub [\[32\]](#) shared repository, and a GitHub codespace (synchronized with the repository) containing a virtual environment with all the dependencies installed on it. This allows Professor Thomas Leek to experiment with the models from his web browser, easily (he doesn't need to install anything such as Python, VSCode or any python library in his institutional machine), and having immediate access to any update from my side. This has played and will continue to play a crucial role for the collaboration between both of us.

5.1.1.2. Challenges

While the model was able to capture some relationships, it was not always accurate, throwing some result that don't make any sense from a semantic perspective. Some of these results can be seen in Chapter 4.

We attribute this (to some extent) to data scarcity, the main problem that we can face when working with small amounts of data in [ML](#).

As mentioned, the primary challenge that was identified is the scarcity of data in ancient languages. Unlike modern languages, which continually generate new data through various channels like social media, publications, and online content, ancient languages are limited to historical texts and inscriptions. This scarcity presents a significant hurdle in applying modern [NLP](#) and especially [ML](#) techniques, which (almost) always perform better with large datasets, since data is their source for learning. Despite this, we have explored various methods to mitigate this challenge, such as transfer learning and data augmentation techniques, which will be further explored in section [5.2 Future Directions](#).

5.1.1.3. Conclusion

The successful training and evaluation of the word embedding model for Middle High German (despite its currently low accuracy) underscore the feasibility of extending this approach to other ancient languages. Our collaboration with Professor Leek exemplifies the importance of interdisciplinary efforts in enhancing the accuracy and applicability of these models. Moving forward, similar collaborative frameworks can be established to further refine these techniques and apply them to a broader range of ancient languages.

By addressing these foundational aspects, we have not only demonstrated the practical feasibility of the project but also provided a robust starting point for the continued development of tools aimed at identifying and analyzing related text fragments across different ancient languages. This groundwork will be crucial in guiding future research and ensuring the successful realization of the project's objectives.

5.1.2. Technology and Methods

In this study, we successfully applied state-of-the-art techniques, methods, and technologies that we had never used before to both modern and ancient languages. Our work demonstrated the effectiveness and robustness of these approaches, contributing valuable insights to the NLP field. This endeavor is considered a success, highlighting the potential for further advancements through continued interdisciplinary collaboration and innovation.

5.1.3. Impact and Applications

The successful development of this tool has the potential to significantly advance the field of ancient language studies. By enabling faster and more accurate identification of related text fragments, researchers can uncover previously unnoticed translations, adaptations, and connections between ancient texts. This capability not only enhances our understanding of historical linguistic relationships but also provides insights into cultural and historical contexts.

Such a tool will streamline the research process, allowing scholars to quickly identify and analyze relevant texts. This can lead to new discoveries in historical linguistics, archaeology, and cultural studies. By facilitating the identification of relationships between texts, the tool can help reconstruct lost languages, understand the spread of cultural and religious ideas, and trace the evolution of linguistic features over time.

5.2. Future directions

With the groundwork laid by this project, we have identified several promising paths for future research and development. These directions aim to improve our current results, overcome the challenges we encountered, and extend its use to more ancient languages. By focusing on these areas, we can continue to improve our methods and enhance our understanding of working with ancient texts and their connections. This section outlines the main future directions that will guide the next steps of this research.

5.2.1. Improvements

In the previous section 4 we gathered and presented the results we obtained after training our own word embeddings models. After analyzing and studying these results we will now discuss how these results could be improved in the future, so that the continuation of the work will be more efficient and effective.

To our understanding, and after considering different options, there are two main ways in which the current result could be greatly improved, and that have been already studied by other researchers, and that have enough work around them to be considered here.

The first one of these two paths to be followed would be to directly address data scarcity through various different methods which we will discuss later. The other presented path would be to adopt a different approach on training embeddings, one specific approach designed for low-resource languages, and we would be able to work with these techniques because they are often variations of the basics studied in this document, optimized for the specific task of learning the highest quality embeddings possible assuming very low amounts of data.

In the following sections we will discuss about how can these methods help improving our results, what are the advantages and disadvantages of one versus the other, and we will also provide some guidance on how to approach the different methods.

5.2.1.1. Addressing data scarcity

Starting with the first proposed path to improve current results we will study what are the tools or methods we can find to address or "fight" data scarcity.

Data scarcity is a significant issue in ML because these models rely heavily on large and diverse datasets to learn patterns and make accurate predictions. When data is limited, the models struggle to generalize, often resulting in poor performance and overfitting¹. This challenge is particularly pronounced in specialized fields like ancient language processing, where available data is inherently scarce.

To mitigate the effects of data scarcity (within the whole domain of data science and ML, not only for NLP field), a substantial body of research has focused on techniques such as data augmentation, transfer learning, and synthetic data generation. Data augmentation involves creating modified versions of existing data to increase the dataset's size and variability. Transfer learning leverages pre-trained models on larger datasets from related domains, which are then fine-tuned on the smaller target dataset. Synthetic data generation creates new data points through simulations or generative models. These methods collectively aim to enhance the robustness and accuracy of ML models, even when faced with limited data.

A preliminary study has been done in order to understand how these improvements could be carried out, here are the main ways to improve performance of ML models in a context of small amounts of data:

- **Collecting more data:** The first and more obvious way to address data scarcity is to collect more data. This is not always possible but it has to be mentioned that the best way to deal with an overfitted model is always to provide more data. In our case new sources of text could be included into the training of the model, improving it's capacity to generalize by having more and different text sources.

¹Overfitting is a common issue in machine learning where a model learns the training data too well, including its noise and outliers, to the point that it performs poorly on new, unseen data. Essentially, an overfitted model captures not only the underlying patterns but also the random fluctuations and idiosyncrasies of the training data. This results in high accuracy on the training dataset but low accuracy on validation or test datasets, indicating poor generalization.

- **Data augmentation:** Moving on to the first real [ML](#) technique to address data scarcity, it is performing what is known as data augmentation. Data augmentation is usually used in [ML](#) to artificially increase the size and diversity of a dataset by applying various transformations or modifications to the existing data. In our case some of these variations could be the following:
 - **Synonym Replacement:** By identifying and substituting words with their synonyms within the Middle High German language, new text variations can be generated. This could require a comprehensive synonym database specific to the language.
 - **Back-Translation:** This technique involves translating a Middle High German text into a modern language (e.g., German or English) and then translating it back. The back-translated text will have slight variations from the original while retaining the same meaning.
 - **Paraphrasing:** Creating paraphrased versions of existing texts can introduce variability without modifying the general context in which words appear.

All these could be done manually by language experts or using automated tools trained on Middle High German if available.

Data augmentation is a very common practice [ML](#) field, but in our judgement this can be very costly in terms of effort for this particular application, since an expert would need to process (or oversee in the case that automated tools were used) vast amounts of text. Anyways it is very common and has demonstrated its effectiveness many times over time, so it could be further considered and studied.

- **Synthetic data generation:** Very similarly to data augmentation, synthetic data generation could be helpful to deal with data scarcity. In this case completely new data would be generated instead of it just being a variation of already existing data. However this approach faces the same issues as data augmentation with the addition of the fact that generating new good-quality data for an ancient language can be incredibly challenging, therefore it wouldn't be the first recommended option.
- **Transfer learning:** Transfer learning is the last technique that we considered potentially relevant in this case. Transfer learning is a [ML](#) technique where a model trained on one task is reused or adapted for another related task. Instead of training a model from scratch on the new task, transfer learning leverages the knowledge learned from the source task to improve the performance of the model on the target task. This approach is particularly useful when the target task has limited data or resources compared to the source task, which is the case here. Transfer learning has become a widely used technique in [ML](#) and can be potentially useful here. Also transfer learning has already been used on low-resource languages word embeddings (with different goal) in the following paper: T. Q. Nguyen and D. Chiang, *Transfer learning across low-resource, related languages for neural machine translation*, 2017. arXiv: [1708.09803 \[cs.CL\]](#).

5.2.1.2. Using a different approach

The second proposed path to improve current results would be to assume the scarcity of data (instead of trying to fight it) and try to work with it by applying algorithms and tools that have been developed specifically for this purpose.

As we already know data scarcity is the main issue when working with ancient languages in the [ML](#) field. But this is not an issue specific to only ancient languages, in fact, according to *SIL International* [\[34\]](#) (formerly known as the Summer Institute of Linguistics) there are more than 7100 living languages around

the world to this days, this is languages that are currently being used. Most of them are spoken by very few people and even "over 40% of these languages are endangered". Still they are living languages, most of which are even more low-resource than some ancient languages such as Latin. Actually any modern language excluding the most used ones (such as English, Spanish or Mandarin) could be considered as low-resource, depending on the context (previously mentioned numbers were published in: Eberhard, D. M., G. F. Simons, *et al.*, Eds., *Ethnologue: Languages of the world. twenty-seventh edition*, Dallas, Texas, 2024. [Online]. Available: <http://www.ethnologue.com>).

This means that being able to work with low resource languages is interesting not only for ancient language researchers, but for anyone that want to apply NLP state-of-the-art techniques to less common modern languages. This is why a lot of research in this direction has already been done.

The idea here would be to work on studying and applying a different approach to generate our word embeddings, one that was specifically designed to perform effectively with low-resource languages such as the following C. Jiang, H.-F. Yu, C.-J. Hsieh, *et al.*, «Learning word embeddings for low-resource languages by PU learning», in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, M. Walker, H. Ji, and A. Stent, Eds., New Orleans, Louisiana: Association for Computational Linguistics, Jun. 2018, pp. 1024–1034. DOI: [10.18653/v1/N18-1093](https://doi.org/10.18653/v1/N18-1093). [Online]. Available: <https://aclanthology.org/N18-1093>. This approach is based in GloVe, or more specifically on it's co-occurrence matrix adding some variations to deal with the small amounts of data.

Another interesting approach is described in V. Hangya, S. Severini, R. Ralev, *et al.*, *Multilingual word embeddings for low-resource languages using anchors and a chain of related languages*, 2023. arXiv: [2311.12489](https://arxiv.org/abs/2311.12489) [cs.CL], and includes relating different low-resource languages embeddings to each other, which is one of the main goals for the project.

5.2.2. New methods and technologies

In this subsection we will go over a few ideas that came up during the development of the project, and that could be useful to dive deeper and understand how they work in order to utilize them in the future. In this case, opposing to the improvements mentioned before, these are techniques or algorithms that haven't been used here and are new to the project, we will briefly explain the general idea of them.

5.2.2.1. Contextual embeddings

In the first place, we understood the basics of word embeddings, but we also know that most modern LLMs don't utilize them in the same way we do. Even tough the idea behind them is the same, these refined models introduce some improvements, and will continue to introduce more with time, since these language models are in constant evolution, and they change a lot in small intervals of time. One of the differences are contextual embeddings.

Contextual embeddings are a more sophisticated approach to representing words or phrases in NLP. Unlike traditional word embeddings (studied in depth in this document), which assign fixed vectors to each word regardless of it's context (here we mean the context where these words appear after training, when generating the word embedding after the training phase), contextual embeddings capture the meaning of a word based on its surrounding context within a sentence or document.

One of the most popular models for generating contextual embeddings is the Transformer architecture, which has revolutionized NLP since its introduction. Transformer-based models like BERT or GPT have gained significant attention for their ability to produce high-quality contextual embeddings.

The process of generating contextual embeddings involves passing the input text through multiple layers of neural network transformers. Each layer refines the representation of the input tokens based on their contextual relationships with other tokens in the sequence. This iterative refinement process allows the model to capture intricate linguistic nuances and semantic dependencies, leading to more accurate embeddings. This means we can't just store a vectorized vocabulary with all the embeddings (as we have done here), instead we need to constantly run a model that generates new embeddings for each query.

Contextual embeddings have several advantages over traditional word embeddings:

- **Context Sensitivity:** Contextual embeddings capture the meaning of words based on their context within the input sequence. This means that the same word can have different embeddings depending on its usage in different sentences or contexts.
- **Semantic Richness:** By considering the surrounding context, contextual embeddings can capture a wider range of semantic information, including word sense disambiguation and syntactic relationships. This results in more nuanced and informative representations of the input text.
- **General Improved Performance:** Models that utilize contextual embeddings have demonstrated superior performance across various [NLP](#) tasks. Their ability to leverage contextual information enables them to better understand and process natural language text.

Despite their effectiveness, contextual embeddings also pose some challenges, including computational complexity, storage requirements, and the need for large amounts of training data. However, ongoing research efforts continue to address these challenges and improve the scalability and efficiency of contextual embedding models.

In summary, I think this can be a path to research later, even though I am not sure it will be possible to generate these kind of embeddings for such low resource languages, it would be a great addition to understand how they work and experiment with them.

5.2.2.2. Vectorial spaces alignment

One of these ideas is related to the ability of utilizing word embeddings across different languages, this is a future challenge we haven't faced yet but we have found some pieces of information on alignment of word embeddings. This means we could try to align word embedding spaces trained over different corpora to be able to relate embeddings from one model to the other. This has been done to relate models trained over different contexts, and it has been researched[38] [39], our intuition is that we could try aligning two models from different languages, we will see a tool that could be useful to do this later.

5.2.2.3. Sentence embeddings

We are also decided to explore sentence embedding in depth, trying to apply them to ancient languages in a way that allows us to relate pieces of text, which was one of the original main goals of the project. Even though we have run some tests using the English languages, we are not ready yet to apply this techniques to ancient languages, in part due to the lack of resources. Therefore this will be further investigated for sure.

5.2.3. Tools

To finish the section, we will mention two tools we found and we think will be useful in the future, to improve our current results and to implement new techniques that may provide more and better functionalities.

5.2.3.1. CLTK

The Classical Language Toolkit (CLTK) [40] [41] is an open-source software library designed for working with classical languages (probably inspired from NLTK), such as Ancient Greek, Latin or [MHG](#) which is useful for us. It provides a comprehensive suite of tools and resources to facilitate the analysis, processing, and study of texts in these languages. CLTK aims to bring the capabilities of modern computational linguistics and [NLP](#) to the domain of classical philology.

The CLTK offers some very good tools that could be used in the future to improve our text processing methods, providing simple and powerful tools that could do a better job since they are developed by experts in the field. Some of them can be tokenization, lemmatization, text normalization or datasets.

CLTK is a community-driven project, with contributions from scholars, linguists, and developers around the world. It is designed to be extensible, allowing users to add new languages, tools, and resources as needed. The open-source nature of the project encourages collaboration and sharing of knowledge and resources within the classical studies community.

The Classical Language Toolkit (CLTK) is a valuable resource for anyone interested in the computational analysis of classical languages. By providing a wide range of tools and resources, CLTK bridges the gap between classical philology and modern computational techniques, enabling new forms of research and exploration in the field of classical studies.

5.2.3.2. CADE

CADE is a tool to align distributional vector spaces. CADE is the more general name for another framework Temporal Word Embeddings with a Compass (TWEC). TWEC was originally implemented by Valerio Di Carlo during his masters' thesis and published at the AAAI2019 Conference [42] [43]. Eventually, they changed the name due to the more general nature of the alignment provided by the model.

The tool is based on a compass to help align the models and can be a good starting point to try and align word embedding models, since it has already been used in different applications. Much more information can be found on their website [44] their GitHub repository [45] and this blog [46] where they were cited.

5.3. Final conclusion

As a last conclusion we could remark the success on understanding and applying state-of-the-art techniques to both modern and ancient languages, studying the panorama in the [NLP](#) field and working in an interdisciplinary collaboration, that we expect to continue and grow, exploring the fascinating world of Natural Language Processing and advancing the research with respect to the application of the latest technology to ancient text.

Bibliography

- [1] *University of wisconsin-stevens point website*, <https://www.uwsp.edu/>.
- [2] T. Mikolov, K. Chen, G. Corrado, and J. Dean, «Efficient estimation of word representations in vector space», 2013. arXiv: [1301.3781](https://arxiv.org/abs/1301.3781) [cs.CL].
- [3] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, «Distributed representations of words and phrases and their compositionality», 2013. arXiv: [1310.4546v1](https://arxiv.org/abs/1310.4546) [cs.CL].
- [4] *Google for developers ml course*, <https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space>.
- [5] *Word2vec available code*, <https://code.google.com/archive/p/word2vec/>.
- [6] J. Pennington, R. Socher, and C. D. Manning, «Glove: Global vectors for word representation», in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: <http://www.aclweb.org/anthology/D14-1162>.
- [7] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, «Enriching word vectors with subword information», 2017. arXiv: [1607.04606](https://arxiv.org/abs/1607.04606) [cs.CL].
- [8] P. Gage, «A new algorithm for data compression», *The C Users Journal archive*, vol. 12, pp. 23–38, 1994. [Online]. Available: <https://api.semanticscholar.org/CorpusID:59804030>.
- [9] *Bpe article*, <http://www.pennelynn.com/Documents/CUJ/HTML/94HTML/19940045.HTM>.
- [10] R. Sennrich, B. Haddow, and A. Birch, «Neural machine translation of rare words with subword units», in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, K. Erk and N. A. Smith, Eds., Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. DOI: [10.18653/v1/P16-1162](https://doi.org/10.18653/v1/P16-1162). [Online]. Available: <https://aclanthology.org/P16-1162>.
- [11] *Stemming vs lemmatization in nlp - niraj bhoi at medium*, <https://nirajbhoi.medium.com/stemming-vs-lemmatization-in-nlp-efc280d4e845>.
- [12] J. E. F. Friedl, *Mastering Regular Expressions*, 3rd ed. Beijing: O'Reilly, 2006, ISBN: 978-0-596-52812-6. [Online]. Available: <https://www.safaribooksonline.com/library/view/mastering-regular-expressions/0596528124/>.
- [13] J. Goyvaerts and S. Levithan, *Regular expressions cookbook, [detailed solutions in eight programming languages]*, English, 2. ed., rev. and updated. Beijing [u.a.]: O'Reilly, 2012, XIV, 594 S, ISBN: 9781449319434.
- [14] M. Fitzgerald, *Introducing Regular Expressions - Unraveling Regular Expressions, Step-by-Step*. O'Reilly, 2012, pp. I–XI, 1–136, ISBN: 978-1-449-39268-0.
- [15] *Arthur samuel's wikipedia page*, [https://en.wikipedia.org/wiki/Arthur_Samuel_\(computer_scientist\)](https://en.wikipedia.org/wiki/Arthur_Samuel_(computer_scientist)).

- [16] A. L. Samuel, «Some studies in machine learning using the game of checkers», *IBM Journal of Research and Development*, vol. 44, no. 1.2, pp. 206–226, 2000. DOI: [10.1147/rd.441.0206](https://doi.org/10.1147/rd.441.0206).
- [17] D.-H. Lee, S. Zhang, A. Fischer, and Y. Bengio, *Difference target propagation*, 2015. arXiv: [1412.7525](https://arxiv.org/abs/1412.7525) [cs.LG].
- [18] W.-D. Ma, J. Lewis, and W. Kleijn, *The hsic bottleneck: Deep learning without back-propagation*, Aug. 2019.
- [19] *Sgd wikipedia page*, https://en.wikipedia.org/wiki/Stochastic_gradient_descent.
- [20] J. Lu, *Gradient descent, stochastic optimization, and other tales*, 2024. arXiv: [2205.00832](https://arxiv.org/abs/2205.00832) [cs.LG].
- [21] *Stochastic definition*, <https://en.wikipedia.org/wiki/Stochastic>.
- [22] I. Amir, T. Koren, and R. Livni, *Sgd generalizes better than gd (and regularization doesn't help)*, 2021. arXiv: [2102.01117](https://arxiv.org/abs/2102.01117) [cs.LG].
- [23] *Glove matrix factorization image*, <https://machinelearninginterview.com/topics/natural-language-processing/what-is-the-difference-between-word2vec-and-glove/>.
- [24] *Gnu/linux information at wikipedia*, <https://en.wikipedia.org/wiki/Linux>.
- [25] *Pip website*, <https://pypi.org/project/pip/>.
- [26] *Conda website*, <https://conda.io/projects/conda/en/latest/index.html>.
- [27] *Jupyter notebooks*, <https://jupyter.org/>.
- [28] *Natural language toolkit (nltk)*, <https://www.nltk.org/>.
- [29] *Kaggle*, <https://www.kaggle.com/>.
- [30] *Gensim word2vec module documentation*, <https://radimrehurek.com/gensim/models/word2vec.html>.
- [31] T. Klein, K.-P. Wegera, S. Dipper, and C. Wich-Reif, *Referenzkorpus mittelhochdeutsch (1050–1350), version 1.0*, <https://www.linguistics.ruhr-uni-bochum.de/rem/>, ISLRN 332-536-136-099-5, 2016.
- [32] *Github*, <https://github.com/>.
- [33] T. Q. Nguyen and D. Chiang, *Transfer learning across low-resource, related languages for neural machine translation*, 2017. arXiv: [1708.09803](https://arxiv.org/abs/1708.09803) [cs.CL].
- [34] *Sil international wikipedia page*, https://en.wikipedia.org/wiki/SIL_International.
- [35] Eberhard, D. M., G. F. Simons, and C. D. Fennig, Eds., *Ethnologue: Languages of the world. twenty-seventh edition*, Dallas, Texas, 2024. [Online]. Available: <http://www.ethnologue.com>.
- [36] C. Jiang, H.-F. Yu, C.-J. Hsieh, and K.-W. Chang, «Learning word embeddings for low-resource languages by PU learning», in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, M. Walker, H. Ji, and A. Stent, Eds., New Orleans, Louisiana: Association for Computational Linguistics, Jun. 2018, pp. 1024–1034. DOI: [10.18653/v1/N18-1093](https://doi.org/10.18653/v1/N18-1093). [Online]. Available: <https://aclanthology.org/N18-1093>.
- [37] V. Hangya, S. Severini, R. Ralev, A. Fraser, and H. Schütze, *Multilingual word embeddings for low-resource languages using anchors and a chain of related languages*, 2023. arXiv: [2311.12489](https://arxiv.org/abs/2311.12489) [cs.CL].

- [38] M. Izbicki, «Aligning word vectors on low-resource languages with Wiktionary», in *Proceedings of the Fifth Workshop on Technologies for Machine Translation of Low-Resource Languages (LoResMT 2022)*, A. K. Ojha, C.-H. Liu, E. Vylomova, *et al.*, Eds., Gyeongju, Republic of Korea: Association for Computational Linguistics, Oct. 2022, pp. 107–117. [Online]. Available: <https://aclanthology.org/2022.loresmt-1.14>.
- [39] A. Diallo and J. Fürnkranz, *Unsupervised alignment of distributional word embeddings*, 2022. arXiv: 2203.04863 [cs.CL].
- [40] K. P. Johnson, P. J. Burns, J. Stewart, T. Cook, C. Besnier, and W. J. B. Mattingly, «The Classical Language Toolkit: An NLP framework for pre-modern languages», in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing: System Demonstrations*, Online: Association for Computational Linguistics, Aug. 2021, pp. 20–29. DOI: 10.18653/v1/2021.acl-demo.3. [Online]. Available: <https://aclanthology.org/2021.acl-demo.3>.
- [41] K. P. Johnson, P. Burns, J. Stewart, and T. Cook, *Cltk: The classical language toolkit*, 2014–2021. [Online]. Available: <https://github.com/cltk/cltk>.
- [42] V. Di Carlo, F. Bianchi, and M. Palmonari, «Training temporal word embeddings with a compass», in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 6326–6334.
- [43] F. Bianchi, V. Di Carlo, P. Nicoli, and M. Palmonari, «Compass-aligned distributional embeddings for studying semantic differences across corpora», *arXiv preprint arXiv:2004.06519*, 2020.
- [44] *Cade website*, <https://federicobianchi.io/cade/>.
- [45] *Cade github repository*, <https://github.com/vinid/cade>.
- [46] *Cade cited in a post explaining compass concept*, <https://federicobianchi.medium.com/aligning-temporal-diachronic-word-embeddings-with-a-compass-732ab7427955>.
- [47] *Visual studio code*, <https://code.visualstudio.com/>.
- [48] *Python*, <https://www.python.org/>.
- [49] *Git*, <https://git-scm.com/>.
- [50] *The latex project*, <https://www.latex-project.org/>.
- [51] *Overleaf website*, <https://www.overleaf.com/>.

Appendix A

Tools and resources

The tools used to build this project have been:

- Personal computer
- GNU/Linux operating system [\[24\]](#)
- Visual Studio Code source code editor [\[47\]](#)
- Python programming language [\[48\]](#)
- Git control version [\[49\]](#)
- GitHub cloud Git repository [\[32\]](#)
- L^AT_EX markup language [\[50\]](#)
- Overleaf L^AT_EX online editor [\[51\]](#)

Universidad de Alcalá
Escuela Politécnica Superior



ESCUELA POLITECNICA
SUPERIOR



Universidad
de Alcalá