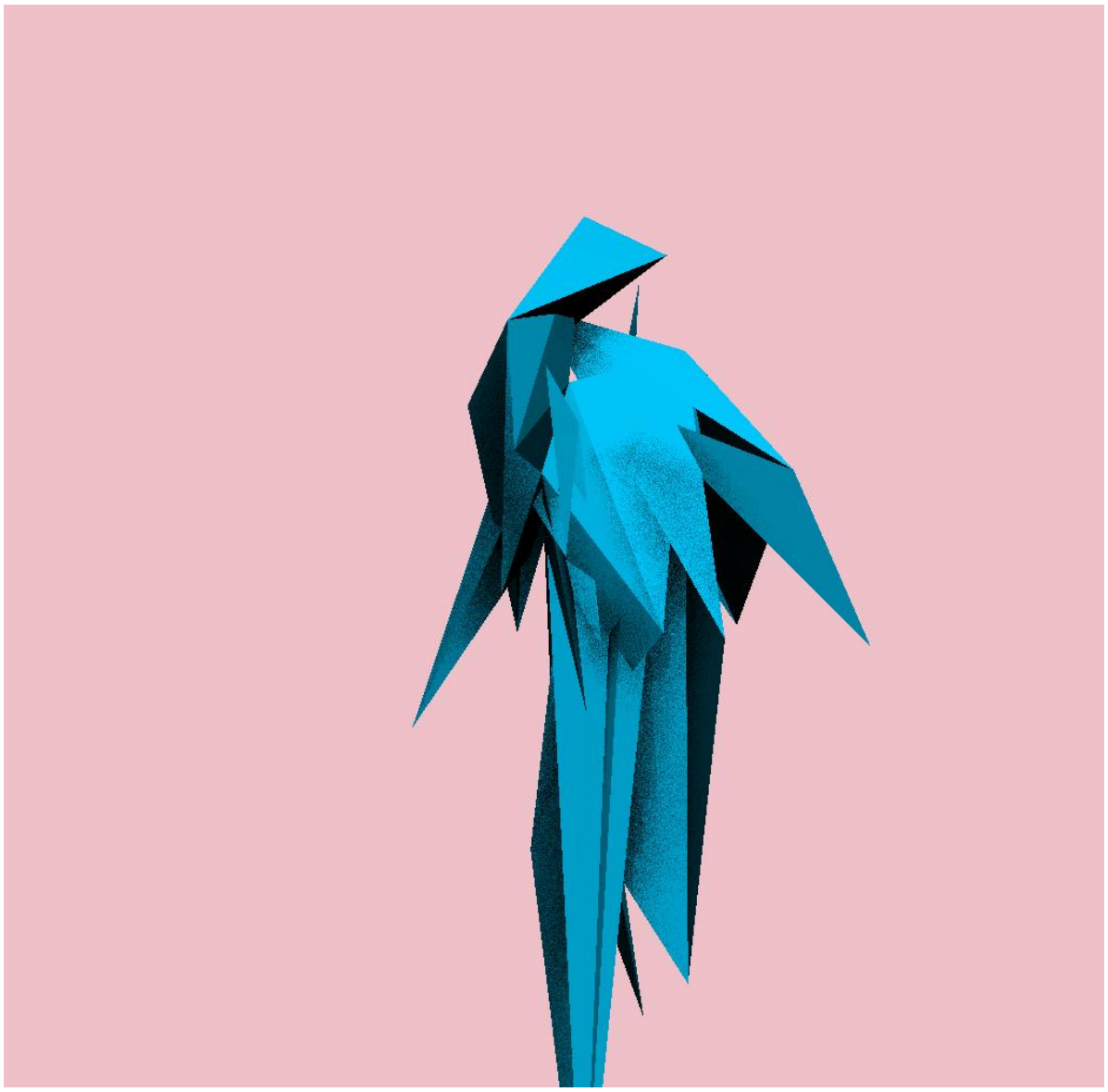


The Geometry of Language : Word Embeddings

Panagiotis Michalatos

Introduction

Word Embeddings

Today we are looking at a special type of vector space that is used to capture something of the semantic structure in language. These spaces are called “word embeddings”. The models that are used to construct and interrogate such spaces assign to each word a vector (a position in the space) such that two word-vectors will be closer together the more similar or associated the corresponding words are.

The stochastic parrot

Each new technology gives rise to scepticism and criticism especially when it is related to language and its association with the very nature of being human.

For example, the term “stochastic parrot” was coined by Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Margaret Mitchell in the paper titled “On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? 🦜”

It suggests that although large language models can generate convincing language, they may not understand the meaning of what they generate and what they are responding to.

Even if intuitively plausible such proposition would require that we know what it means for humans to know the meaning of their own words to begin with, and that’s not a given despite advances in the fields of biology, cognitive science, neuroscience, linguistics etc…

In this discussion we’ll try to avoid the polemic and the hype, and instead focus on understanding some basic principles to approach such models not as surrogates of human cognition but as microscopes that allow us to explore the structure of language and by extension of culture and society.

Plato: Phaedrus ~370 BCE

This kind of criticism of a technology of language is not new. More than 2000 years ago in Plato’s dialogue “Phaedrus”, we have a scene where Socrates discusses the merits and dangers of another technology “writing”.

Socrates tells a story about an ancient king of Egypt, Thamus that was offered writing by the god Thoth.

“**Thoth to King Thamus**: *Here, O king, is a branch of learning that will make the people of Egypt wiser and* ***improve their memories. My discovery provides a recipe for memory and wisdom****.”*

*“**Thamus to Thoth :** And so it is that you by reason of your tender regard for the writing that is your offspring have declared the very opposite of its true effect. If men learn this, it will implant forgetfulness in their souls.* ***They will cease to exercise memory because they rely on that which is written, calling things to remembrance no longer from within themselves, but by means of external marks****.”*

*“**Thamus to Thoth :** What you have discovered is a recipe not for memory, but for reminder. And it is no true wisdom that you offer your disciples, but only the*

semblance of wisdom, for by telling them of many things without teaching them you will make them seem to know much while for the most part they know nothing. And as men filled not with wisdom but with the conceit of wisdom they will be a burden to their fellows."

*"**Socrates:** You know, Phaedrus, that is the strange thing about writing, which makes it truly correspond to painting. The painter's products stand before us as though they were alive. But if you question them, they maintain a most majestic silence. It is the same with written words. **They seem to talk to you as though they were intelligent, but if you ask them anything about what they say from a desire to be instructed they go on telling just the same thing forever**."*

This last sentence by Socrates is almost identical to arguments against language models.

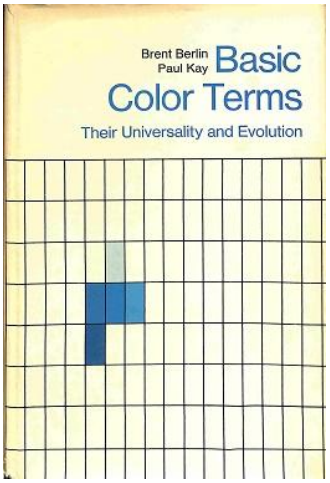
Writing didn't cause people's memories to atrophy and minds to become feeble. The irony is that the reason we have this dialogue today is because Plato wrote it down, since technologies of recording are ways to externalize memory and detach it from the constraints of a mortal body.

Where is the Blue green boundary?

When we look at the color spectrum we see a continuum without clearly demarcated regions. Conventionally we assign labels to some bands of the spectrum and call them colors. The exact wavelengths that colors correspond to are not standardized or universal. Different cultures may place boundaries on average at different locations.

Linguistic relativity and Basic Colour Terms (1969 Berlin & Kay)

The multiple ways in which different cultures over time and at different parts of the world discretized and labeled the color spectrum has led to several nowadays mostly discredited assumptions. For instance, linguistic relativity asserts that language influences the way we think. An even stronger statement was that language determines what we can perceive. The color terms we use nowadays do not map to the terms that different cultures used throughout history. Some spectral bands might not have been significant enough. A term for the color blue as we understand it today seems to be missing from many ancient languages despite occupying large swaths of our visual field (sky, sea…).



In 1969 Berlin and Kay surveyed several languages and claimed to see a pattern in relation to the existence of basic colour terms. They counted the basic color terms in each language and claimed that depending on how many terms for color a language has they are allocated progressively and almost universally to the same colors.

```
All languages have Black and White

3 terms : Red

4 terms : Green or Yellow

5 terms : Green and Yellow

6 terms : Blue

7 terms : Brown

more terms : Purple, Pink, Orange, Gray
```

A simpler version of the same hypothesis was the following:

```
1: Black and White

2: Warm(red/yellow) and Cool(green/blue)

3: red
```

This is presented as an evolutionary process where languages in their earliest forms lack the most basic terms for color. Especially a term for blue seems to always come late in a language's evolution. Blue is a color that does not exist in nature (few animals or plants are blue) and cannot be easily manufactured until the industrial revolution and synthetic dyes became available.

This model has been challenged since the time it was proposed but it does point to the difficulty in mapping a concept as simple as colour across cultures and languages.

How do words acquire meaning?

Philologists:

History, etymology, genealogy

Wittgenstein's Philosophy of language:

"the meaning of a word is its use in the language"

This question is almost as old as written sources. How do we make sense of the seemingly random sounds we make. Up until the end of the 19th century most people who studied this problem would focus on the history, etymology and evolution of words and their meaning. Philologists would trace the genealogy of words diachronically (through time) by studying texts from different periods. History and etymology were the keys to the meaning of words.

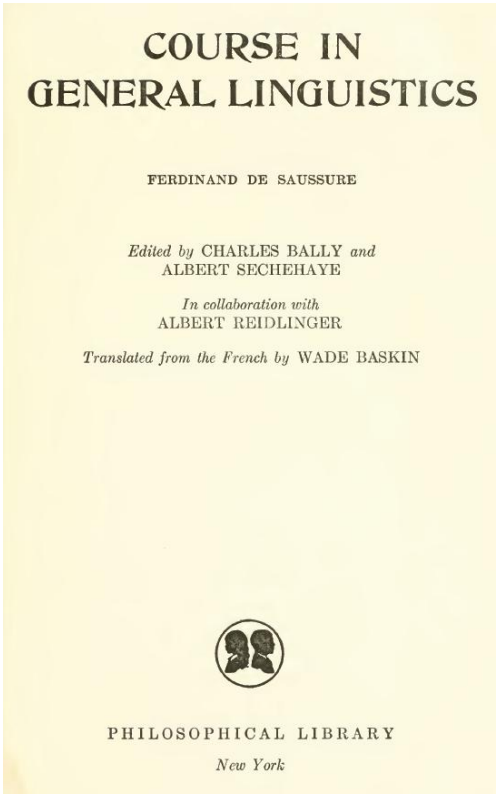
All this changed radically with the advent of modern linguistics and the work of Swiss linguist Ferdinand de Saussure (1857 – 1913).

Linguistics

Modern linguistics has come a long way and studies language in a wholistic manner, incorporating knowledge from different fields such as physiology, biology, evolutionary biology, neuroscience, information theory etc…

But here we will look at the very beginning of linguistics and the strands of it that had the biggest impact in art and culture and seem to resonate better with the simpler models we deal with in machine learning.

Early Linguistics: Ferdinand de Saussure (1857 – 1913)



Following are a few pages from the lecture notes of Ferdinand de Saussure from his "course in general linguistics"

Diachronic / Synchronic

PART TWO

Synchronic Linguistics

Chapter I

GENERALITIES

The aim of general synchronic linguistics is to set up the fundamental principles of any idiosynchronic system, the constituents of any language-state. Many of the items already explained in Part One belong rather to synchrony; for instance, the general properties of the sign are an integral part of synchrony although they were used to prove the necessity of separating the two linguistics.

To synchrony belongs everything called "general grammar," for it is only through language-states that the different relations which are the province of grammar are established. In the following chapters we shall consider only the basic principles necessary for approaching the more special problems of static linguistics or explaining in detail a language-state.

The study of static linguistics is generally much more difficult than the study of historical linguistics. Evolutionary facts are more concrete and striking; their observable relations tie together successive terms that are easily grasped; it is easy, often even amusing, to follow a series of changes. But the linguistics that penetrates values and coexisting relations presents much greater difficulties.

In practice a language-state is not a point but rather a certain span of time during which the sum of the modifications that have supervened is minimal. The span may cover ten years, a generation, a century, or even more. It is possible for a language to change hardly at all over a long span and then to undergo radical transformations within a few years. Of two languages that exist side by side during a given period, one may evolve drastically and the other practically not at all; study would have to be diachronic in the former instance, synchronic in the latter. An absolute state is defined by the absence of changes, and since language changes

101

From

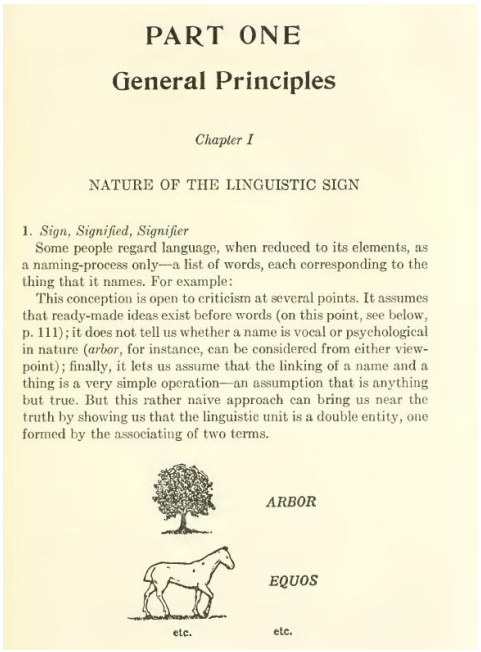
Diachronic: History, etymology, genealogy

to

Synchronic : Structure, Language as System

One of Saussure's most important gestures is to shift the attention from the diachronic study of the history of words that philologists were preoccupied with to the synchronic view of language as a system

The arbitrariness of the linguistic sign



PART ONE

General Principles

Chapter I

NATURE OF THE LINGUISTIC SIGN

1. *Sign, Signified, Signifier*

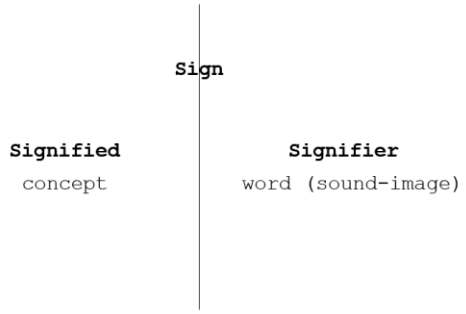
Some people regard language, when reduced to its elements, as a naming-process only—a list of words, each corresponding to the thing that it names. For example:

This conception is open to criticism at several points. It assumes that ready-made ideas exist before words (on this point, see below, p. 111); it does not tell us whether a name is vocal or psychological in nature (*arbor*, for instance, can be considered from either viewpoint); finally, it lets us assume that the linking of a name and a thing is a very simple operation—an assumption that is anything but true. But this rather naive approach can bring us near the truth by showing us that the linguistic unit is a double entity, one formed by the associating of two terms.

ARBOR

EQUOS

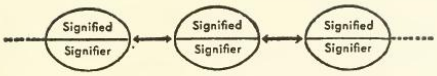
etc. etc.



Linguists even before Saussure had noticed that the words we use to describe concepts and things with rare exceptions do not in any way resemble those things. There is nothing intrinsic to the word “horse”, the sound that this word represents that is specific to the animal horse. Each language has a different word for it, and they all describe the same thing equally well.

Saussure proposes that the unit of meaning is a “sign”, but this sign consists of two parts. One is the signified (the thing that it points to, the concept) the other is the signifier (the word). One of the main features of the sign is its arbitrariness in the sense that there is no “natural” connection between the signified and the signifier but only a conventional one through their use in language by a community that speaks that language.

Value(meaning) is derived from differences



How, then, can value be confused with signification, i.e. the counterpart of the sound-image? It seems impossible to liken the relations represented here by horizontal arrows to those represented above (p. 114) by vertical arrows. Putting it another way—and again taking up the example of the sheet of paper that is cut in two (see p. 113)—it is clear that the observable relation between the different pieces A, B, C, D, etc. is distinct from the relation between the front and back of the same piece as in A/A′, B/B′, etc.

To resolve the issue, let us observe from the outset that even outside language all values are apparently governed by the same paradoxical principle. They are always composed:

(1) of a *dissimilar* thing that can be *exchanged* for the thing of which the value is to be determined; and

(2) of *similar* things that can be *compared* with the thing of which the value is to be determined.

Both factors are necessary for the existence of a value. To determine what a five-franc piece is worth one must therefore know: (1) that it can be exchanged for a fixed quantity of a different thing, e.g. bread; and (2) that it can be compared with a similar value of the same system, e.g. a one-franc piece, or with coins of another system (a dollar, etc.). In the same way a word can be exchanged for something dissimilar, an idea; besides, it can be compared with something of the same nature, another word. Its value is therefore not fixed so long as one simply states that it can be "exchanged" for a given concept, i.e. that it has this or that signification: one must also compare it with similar values, with other words that stand in opposition to it. Its content is really fixed only by the concurrence of everything that exists outside it. Being part of a system, it is endowed not only with a signification but also and especially with a value, and this is something quite different.

A few examples will show clearly that this is true. Modern French *mouton* can have the same signification as English *sheep* but not the same value, and this for several reasons, particularly because in speaking of a piece of meat ready to be served on the

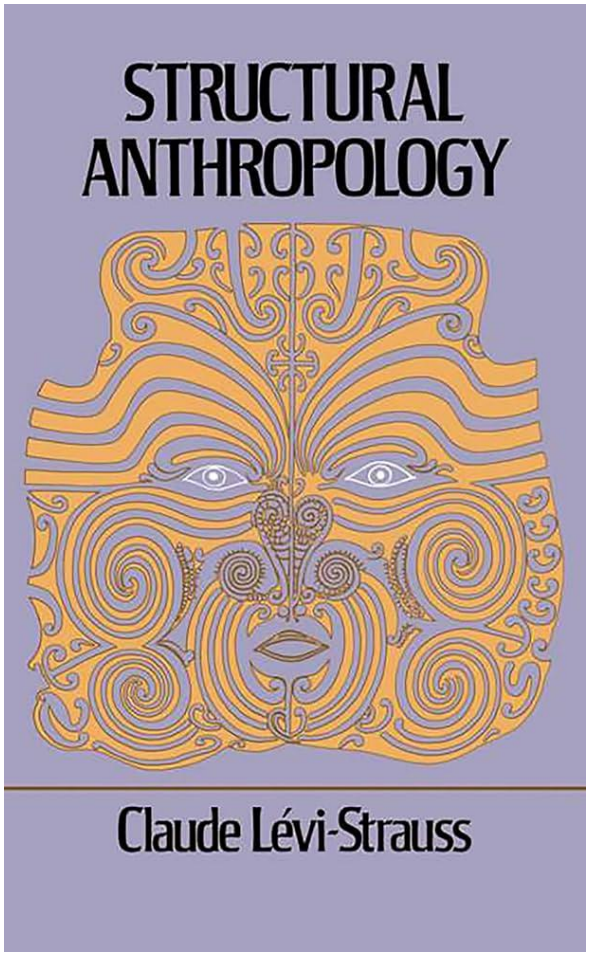
4. *The Sign Considered in Its Totality*

Everything that has been said up to this point boils down to this: in language there are only differences. Even more important: a difference generally implies positive terms between which the difference is set up; but in language there are only differences *without positive terms*. Whether we take the signified or the signifier, language has neither ideas nor sounds that existed before the linguistic system, but only conceptual and phonic differences that have issued from the system. The idea or phonic substance that a sign contains is of less importance than the other signs that surround it. Proof of this is that the value of a term may be modified without either its meaning or its sound being affected, solely because a neighboring term has been modified (see p. 115).

But the statement that everything in language is negative is true only if the signified and the signifier are considered separately; when we consider the sign in its totality, we have something that is positive in its own class. A linguistic system is a series of differences of sound combined with a series of differences of ideas; but the pairing of a certain number of acoustical signs with as many cuts made from the mass of thought engenders a system of values; and this system serves as the effective link between the phonic and psychological elements within each sign. Although both the signified and the signifier are purely differential and negative when considered separately, their combination is a positive fact; it is

When language is seen as a system (a pure relational structure) the individual signs only acquire meaning through their differences (or similarities) to other signs.

Semiotics to structuralism



Saussure's influence spread beyond linguistics especially in the humanities. Some scholars like Roland Barthes, Jean Baudrillard, Umberto Eco applied semiotics (the study of signs) to contemporary culture. Others like Claude Levy Strauss took the idea of language as a system and applied it to anthropology with his seminal work aptly named "structural anthropology"

Christopher Alexander

Architecture theory was caught in the same intellectual currents, sometimes analysing the structure of building and cities as if they were the signifiers of a social signified.

Perhaps the most well-known theorists in this field were Christopher Alexander.

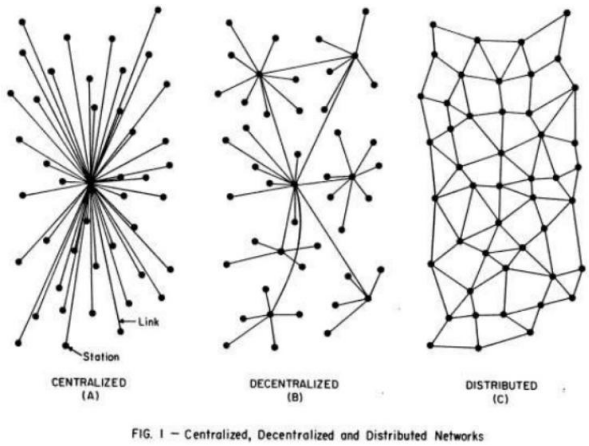
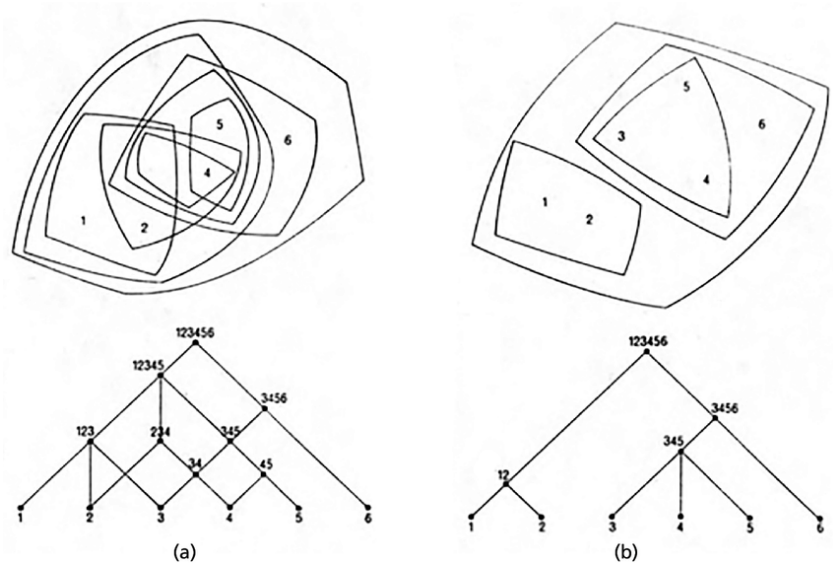


FIG. 1 — Centralized, Decentralized and Distributed Networks

By this point strucutralims has adopted vocabulary and methods (if only in somewhat metaphorical manner) from mathematicsal fields that deal with the study of relational structures, namely graph theory and abstract algebra.



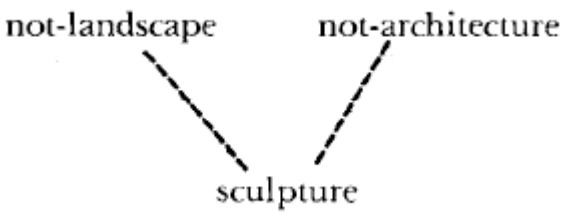
Alexander's book "a city is not a tree" is peppered with graphs. The tree graph is the archetype of the hierarchical structure where materials, functions and meanings are organized into non-overlapping zones (a modernist strawman). This is the structure of artificial cities as opposed to the "natural" cities where boundaries of materials, activities and meaning are fuzzy.

Krauss, Rosalind 1979 : Sculpture in the Expanded Field

The art theorist and critic, Rosalind Krauss in 1979 wrote the essay "sculpture in the expanded field" as a response to the increasing number of ways that artists used the term "sculpture" for works that couldn't fit any traditional sense of the word.

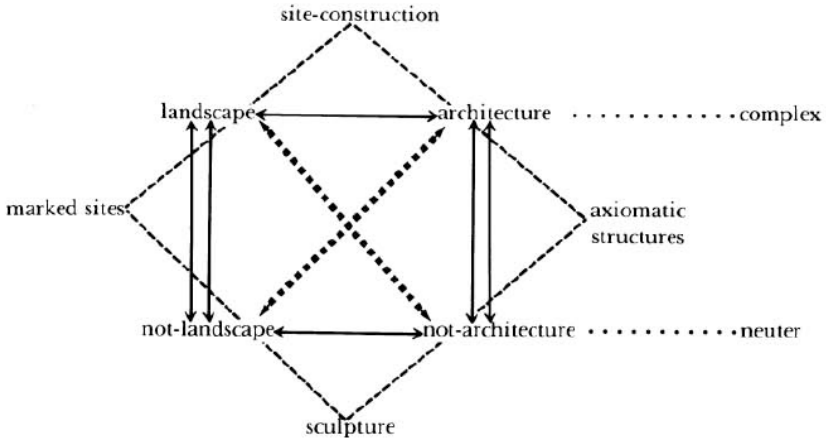
"Over the last ten years rather surprising things have come to be called sculpture: narrow corridors with TV monitors at the ends; large photographs documenting country hikes; mirrors placed at strange angles in ordinary rooms; temporary lines cut into the floor of the desert. Nothing, it would seem, could possibly give to such a motley of effort the right to lay claim to whatever one might mean by the category of sculpture. Unless, that is, the category can be made to become almost infinitely malleable."

Krauss remarks that two works by artist Robert Morris (Green Gallery installation and Mirrored boxes) one occupying a room in a building the other a garden could be thought of as not-architecture and not-landscape. That is the sculpture is everything in the room that is not part of the architecture.



She then goes on to think about a structure that could capture the semantic diversity of art objects, installations, landscapes of the 60s and 70s. She is using as her organizing principle the graph of an algebraic structure called the Klein Group.

*"The expansion to whichI am referring is called a Klein group when employed mathematically and has various other designations, among them the Piaget group, when used by structuralists involved in mapping operations within the human sciences**.* By means of this logical expansion a set of binaries is transformed into a quaternary field which both mirrors the original opposition and at the same time opens it. It becomes a logically expanded field which looks like this**:"*

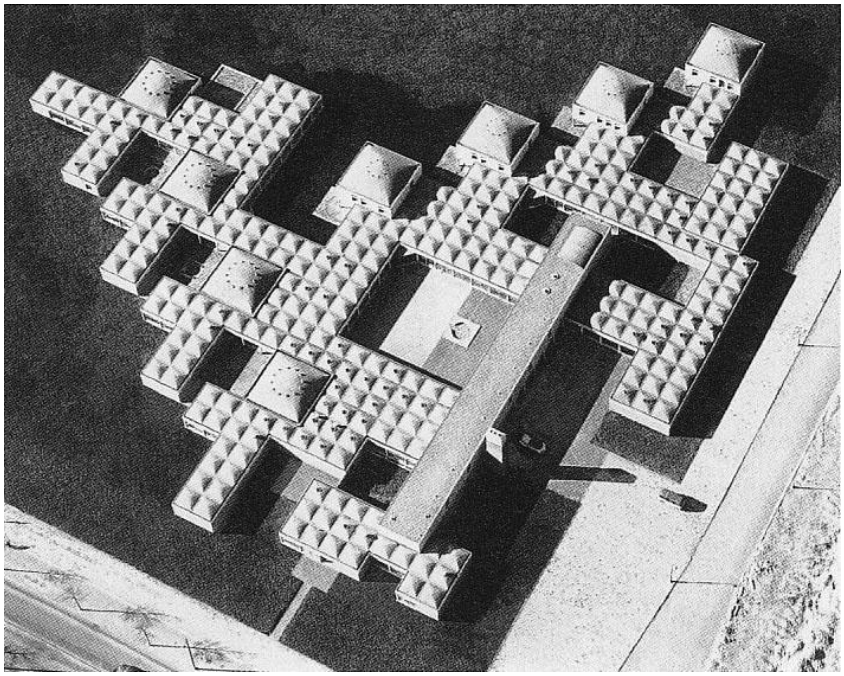


Aldo Van Eyck (1918 – 1999)

"What you should try to accomplish is built meaning. So get close to the meaning and build!" (1962)

'The city is a big house and the house is a small city'.
The Children's Home in Amsterdam (1960)

Aldo Van Eyck was a Dutch Architect who tried to apply some of the ideas floating in the structuralism infused air of his time. He thought that the role of architecture is to build a counterform to a social form (or structure).



One of the central concepts of his philosophy of design was what he called "twin phenomena". The idea here is to take concepts that appear as polar opposites and make them coexist. Things are not just big or small but big relative to something

smaller. It is the difference that gives rise to the meaning. The orphanage design consists of a constant intermixing of scales, of intimate and open spaces.

It also possesses both the “complexity” of a premodern town and the “clarity” of modernism a manifestation of another of Van Eyck’s twin-phenomena, “labyrinthine clarity”.

Language and information theory : compression

While there were various attempts to apply the linguistic, structuralist methods and concepts in the humanities and the arts, Claude Shannon established the field of information theory, which suggests a quantitative but statistical view of the world. One of the central problems in information theory is that of compression.

What is the minimum number of bits we need to encode and transmit a signal?

This is a photo of Lena, one of the most iconic images in computer graphics, computer vision and image analysis. IT has been reproduced in countless papers as it was one of those publicly available reference data sets on which all algorithms were tested.

it consists of 512x512 pixels and with 3 bytes per pixel (R,G,B) it would require 786,432 bytes to store it.

We can shrink it into a 16x16 pixels image and then it would take only 768 bytes, about 1000 times less memory.

If we get a bit cleverer and take into account that the compressed image is going to be seen by humans we can compress it in a way that destroys information that is not relevant to humans.


A Jpeg version can take as little as 16,000 bytes by eliminating noise (high frequency, hard to compress information) and aggresively smearing the color information while preserving tonal variation.

But ultimately we could just describe the image as for example:

Young woman looking at the camera over her shoulder wearing a beige hat with blue feathers

This takes only 90 bytes.

Language is an extremely efficient compression method. It can use very few tokens to encode, communicate and evoke complex scenes, thoughts and ideas.

We will see a similar effect with autoencoder models later in the semester where compression forces the system to discover structure in the data.

If difference is meaning, then how do we compare words?

Having taken a detour through 20th century linguistics, semiotics and a bit of information theory we can land on the question we started with but this time in the technological context we live in.

First, we want to measure the differences (or similarities) between words or more accurately between the concepts that words point to.

Words can be similar in different ways:

Shared Context

Opposites or Synonyms

Shared attributes

Same Function

…

One simple approach is to ask people.

Ask People : SimLex - 999

SimLex-999: Evaluating Semantic Models With (Genuine) Similarity Estimation

Felix Hill
Computer Laboratory
Cambridge University
felix.hill@cl.cam.ac.uk

Roi Reichart
Technion, IIT
roiri@ie.technion.ac.il

Anna Korhonen
Computer Laboratory
Cambridge University
anna.korhonen@cl.cam.ac.uk

In this paper the authors asked 500 native English speakers to rate the similarity between words they are shown in an online interface.

The authors note that there are two related concepts of similarity by distinguishing the relatedness of the cup and coffee from the similarity of the cup and the mug.

*"Although clearly different, coffee and cups are very much related. The psychological literature refers to the conceptual relationship between these concepts as **association**, although it has been given a range of names including **relatedness** (Budanitsky and Hirst, 2006; Agirre et al., 2009), topical similarity (Hatzivassiloglou et al., 2001) and domain similarity (Turney, 2012).*

***Association contrasts with similarity**, the relation connecting cup and mug (Tversky, 1977). **At its strongest, the similarity relation is exemplified by pairs of synonyms; words with identical referents**."*

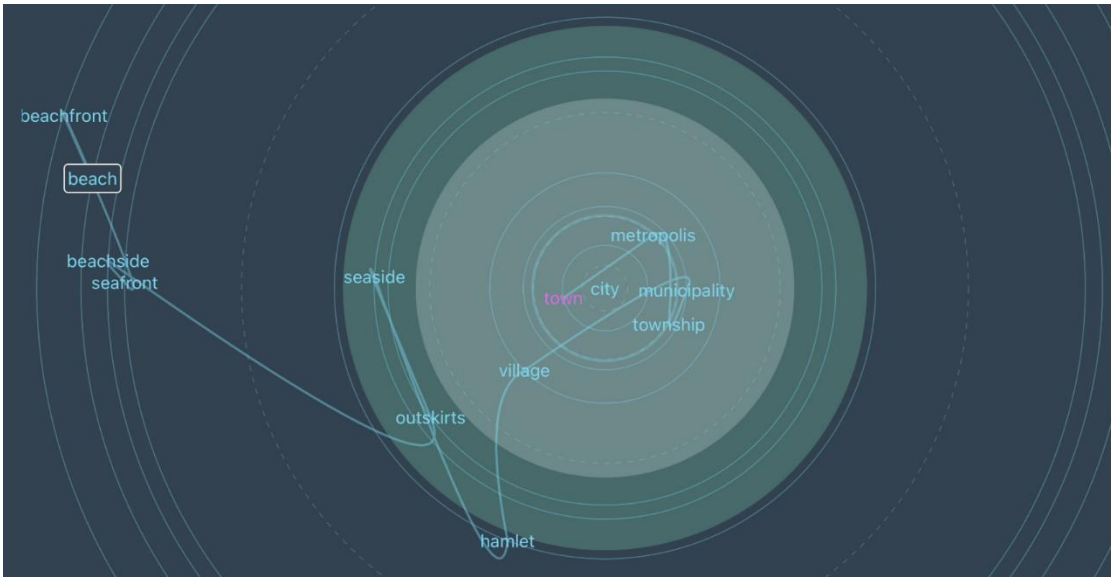
C1	C2	POS	USF*	USF rank (of 999)	SimLex	SimLex rank (of 999)
dirty	*narrow*	A	0.00	999	0.30	996
student	*pupil*	N	6.80	12	9.40	12
win	*dominate*	V	0.41	364	5.68	361
smart	*dumb*	A	2.10	92	0.60	947
attention	*awareness*	N	0.10	895	8.73	58
leave	*enter*	V	2.16	89	1.38	841

Table 2: **Top: Similarity aligns with association** Pairs with a small difference in rank between USF (association) and SimLex-999 (similarity) scores for each POS category. **Bottom: Similarity contrasts with association** Pairs with a high difference in rank for each POS category. *Note that the distribution of USF association scores on the interval [0,10] is highly skewed towards the lower bound in both SimLex-999 and the USF dataset as a whole.

Figure 5: **(a)** Pairs rated by WS-353 annotators (blue points, ranked by rating) and the corresponding rating of annotators following the SimLex-999 instructions (red points). **(b-c)** The same analysis, restricted to pairs in the WS-Sim or WS-Rel subsets of WS-353.

This method gives some insights into the quantitative structure of language and more importantly the authors create a standard that we can use to compare and evaluate artificial estimates of similarity generated by ML models.

Word Embeddings



Using humans to produce these datasets is difficult so techniques have been developed so that machines can "learn" to compute these semantic similarities by just autonomously looking at massive amounts of text.

These methods produce what we call word-embeddings. That is, they generate a dictionary where each word is associated with a vector (a list of numbers). Although these vectors individually are meaningless if we take any two of those vectors and compare them the angle between them will be proportional to their semantic distance. Another way to think about it is that the model starts with a large space and moves the words around until words that point to related concepts become closer together.

Two of these models are Word2Vec and Glove. Both are self-supervised, meaning that they can be trained by consuming text without any human explicitly encoding word relations.

Such models are now embedded in many devices and used for various tasks like text completion, sentiment analysis etc… Unlike the massive LLMs (large language models) , word embeddings have a relatively small memory footprint and are extremely fast and efficient. LLMs also do rely on semantic word embeddings to carry their work so understanding how word embeddings work is a steppingstone towards LLMs as well.

Word2Vec

You can find more information about word2vec at google:
https://code.google.com/archive/p/word2vec/

"This tool provides an efficient implementation of the continuous bag-of-words and skip-gram architectures for computing vector representations of words. These representations can be subsequently used in many natural language processing applications and for further research"

One of the important aspects of these embeddings is that the geometric relationships of word-vectors encode not only similarity in terms of proximity but also more indirect relations like analogies.

Interesting properties of the word vectors

It was recently shown that the word vectors capture many linguistic regularities, for example vector operations **vector('Paris') - vector('France') + vector('Italy')** results in a vector that is very close to **vector('Rome')**, and **vector('king') - vector('man') + vector('woman')** is close to **vector('queen')** [3, 1]. You can try out a simple demo by running **demo-analogy.sh**.

And even vectors that consists of more than one word can be used as queries:

Word Cosine distance

```
   Yangtze_River            0.667376
         Yangtze            0.644091
  Qiantang_River            0.632979
```

Yangtze_tributary 0.623527 Xiangjiang_River 0.615482 Huangpu_River 0.604726 Hanjiang_River 0.598110 Yangtze_river 0.597621 Hongze_Lake 0.594108 Yangtse 0.593442 ```

The above example will average vectors for words 'Chinese' and 'river' and will return the closest neighbors to the resulting vector.

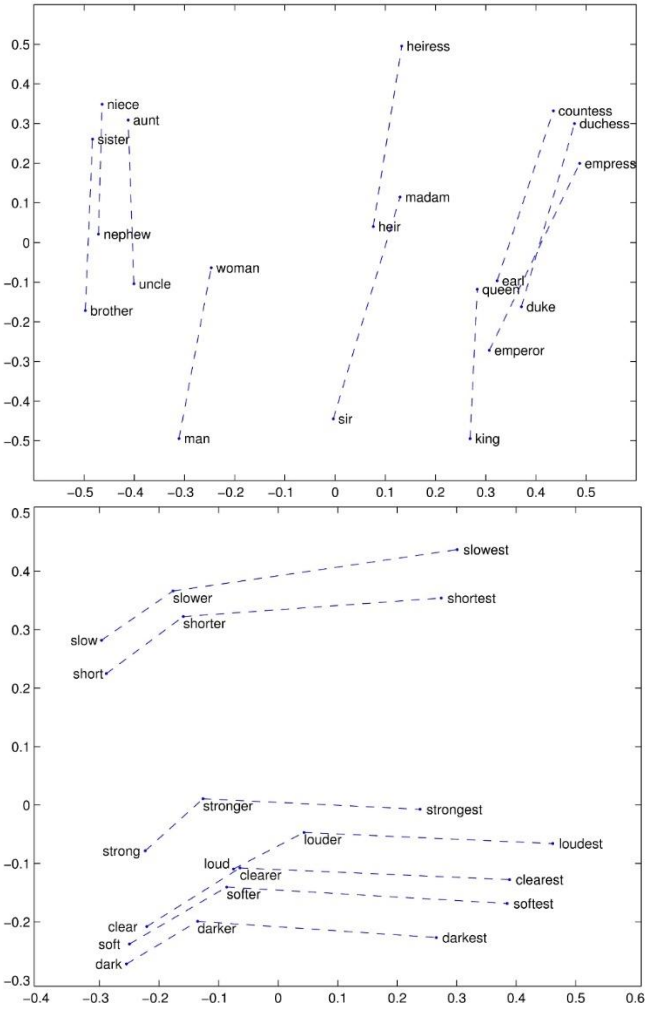
GloVe

More information about GloVe can be found here:

https://nlp.stanford.edu/projects/glove/

"GloVe is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space."

Looking at the above link we can see that Glove too can capture analogies:

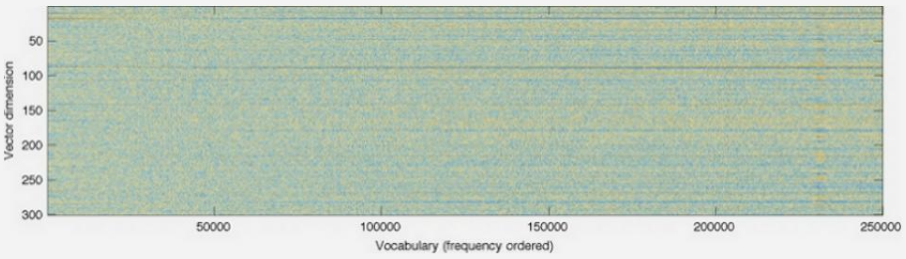


The idea is that similar relationships are parallel vectors in the embedding space.

In the same website, we can find an interesting discussion regarding patterns that appear in the visualization of the totality of the embedding (the matrix that contains the vectors for all 250k+ words)

Visualization

GloVe produces word vectors with a marked banded structure that is evident upon visualization:



The horizontal bands result from the fact that the multiplicative interactions in the model occur component-wise. While there are additive interactions resulting from a dot product, in general there is little room for the individual dimensions to cross-pollinate.

The horizontal bands become more pronounced as the word frequency increases. Indeed, there are noticeable long-range trends as a function of word frequency, and they are unlikely to have a linguistic origin. This feature is not unique to GloVe -- in fact, I'm unaware of any model for word vector learning that avoids this issue.

The vertical bands, such as the one around word 230k-233k, are due to local densities of related words (usually numbers) that happen to have similar frequencies.

Diachronic to Synchronic : Ferdinand de Saussure (1857 – 1913)

At the start of this lecture, we saw how Saussure's insight was to look at language as a system (synchronic) rather than study the history of words (diachronic).

With word embeddings we have a powerful quantitative tool to analyse the structure of language.

Synchronic to Diachronic : Word embeddings as a microscope

An embedding model is trained on a corpus of text (books, publications, online information…) and extract the language structure for this corpus. If we were to limit the training texts to a certain historical period, then we get a snapshot of language at that time and if we produce snapshots for different historical periods and compare them we can perhaps say something about that diachronic dimension.

This is what the authors of the following paper did:

Diachronic Word Embeddings Reveal Statistical Laws of Semantic Change

William L. Hamilton, Jure Leskovec, Dan Jurafsky
Department of Computer Science, Stanford University, Stanford CA, 94305
wleif, jure, jurafsky@stanford.edu

Figure 1: Two-dimensional visualization of semantic change in English using SGNS vectors.[2] **a**, The word *gay* shifted from meaning "cheerful" or "frolicsome" to referring to homosexuality. **b**, In the early 20th century *broadcast* referred to "casting out seeds"; with the rise of television and radio its meaning shifted to "transmitting signals". **c**, *Awful* underwent a process of pejoration, as it shifted from meaning "full of awe" to meaning "terrible or appalling" (Simpson et al., 1989).

Here they can trace the change in semantic value of different words through time. For example, we see the word "gay" starting in the 1900s close to cheerful, morphing into the "witty" / "bright" cluster in the 1950s before transitioning to a signifier of sexual orientation by the 1990s.

Method	Top-10 words that changed from 1900s to 1990s
PPMI	know, got, would, decided, think, stop, remember, **started**, must, wanted
SVD	harry, **headed**, **calls**, **gay**, wherever, male, **actually**, special, cover, naturally
SGNS	**wanting**, **gay**, **check**, **starting**, **major**, **actually**, touching, harry, **headed**, romance

Table 4: Top-10 English words with the highest semantic displacement values between the 1900s and 1990s. Bolded entries correspond to real semantic shifts, as deemed by examining the literature and their nearest neighbors; for example, *headed* shifted from primarily referring to the "top of a body/entity" to referring to "a direction of travel." Underlined entries are borderline cases that are largely due to global genre/discourse shifts; for example, *male* has not changed in meaning, but its usage in discussions of "gender equality" is relatively new. Finally, unmarked entries are clear corpus artifacts; for example, *special*, *cover*, and *romance* are artifacts from the covers of fiction books occasionally including advertisements etc.

Word	Language	Nearest-neighbors in 1900s	Nearest-neighbors in 1990s
wanting	English	lacking, deficient, lacked, lack, needed	wanted, something, wishing, anything, anybody
asile	French	refuge, asiles, hospice, vieillards, infirmerie	demandeurs, refuge, hospice, visas, admission
widerstand	German	scheiterte, volt, stromstärke, leisten, brechen	opposition, verfolgung, nationalsozialistische, nationalsozialismus, kollaboration

Table 5: Example words that changed dramatically in meaning in three languages, discovered using SGNS embeddings. The examples were selected from the top-10 most-changed lists between 1900s and 1990s as in Table 4. In English, *wanting* underwent subjectification and shifted from meaning "lacking" to referring to subjective "desire", as in "the education system is wanting" (1900s) vs. "I've been wanting to tell you" (1990s). In French *asile* ("asylum") shifted from primarily referring to "hospitals, or infirmaries" to also referring to "asylum seekers, or refugees". Finally, in German *Widerstand* ("resistance") gained a formal meaning as referring to the local German resistance to Nazism during World War II.

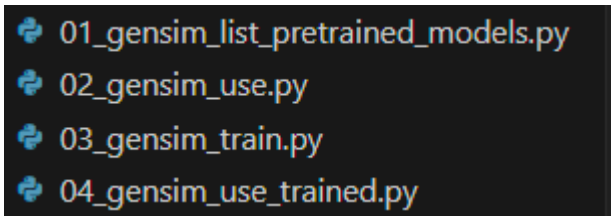
Gensim Python Library

To access such models we will use a Python library called Gensim. This will be our main tool for the tutorials today. You can find the complete documentation with rich examples and references at:

https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html

Tutorials

There is a zip file you can download from Canvas. Unzip it and open the folder in visual studio code. You should see 4 python scripts (plus some other files):



We will look at these files more closely, but before we start, we need to make sure you have Gensim installed.

To install a Python library, we are going to use the Python package a manager Pip.

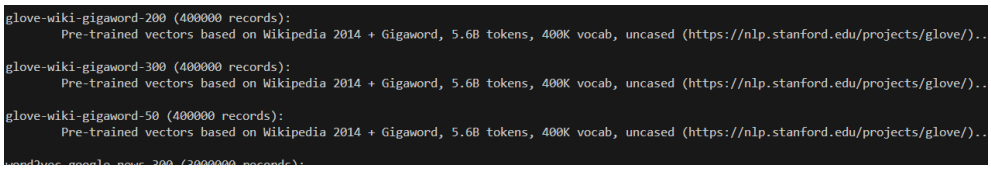
Inside Visual studio code you can open a terminal from the menu (Terminal -> New Terminal). In the terminal issue the command “pip install gensim”. If you are on MacOS you may need to write it as “pip3 install gensim”.

If you get a message that pip cannot be found, then you need to set your python installation folder as a path in your operating system’s environment variable. On windows this can be done by an option during python’s installation. On MacOs it varies depending on your system setup and whether you are using conda etc…

01_gensim_list_pretrained_models.py

This file simply calls into the gensim library and prints a list of the pretrained models that are available. These are models typically trained on millions of pages of text and contain very robust word embeddings.

Running this model, you should see something like this:



You can note down some of these models for testing. We will be using “glove-wiki-gigaword-50” because its vectors are relatively short (50 components per vector) and loads quickly, but for more robust results depending on your use case you may want to experiment with the glove or word2vec -300s.

More components per vector means more semantic directions can be distinguished in this abstract space. You can think of it as semantic resolution.

02_gensim_use

This file demonstrates basic usage of the Gensim library.

It starts with some imports:

```
import gensim
import logging
import os
import numpy as np
import math

from gensim.models import KeyedVectors
import gensim.downloader
```

it imports gensim itself as well as some extra libraries like logging (used by gensim to report progress), os (used to access the file system in your hard-drive), numpy (numerical computation library for efficient matrix, vector operations), math (contains all the math functions like sin, cos, exp etc…)

when importing numpy we use the special syntax “as np” this creates an alias for numpy so we can write **np.dot** rather than **numpy.dot**. Makes the code a bit more readable.

Finally we import the KeyedVectors type from gensim.models. This is the syntax we use in order not to have to write **gensim.models.KeyedVectors** but instead just **KeyedVectors**. KEyedVectors is the main data structure that gensim uses to encode a word embedding, which is a database that associates words with their corresponding vectors.

Finally gensim.downloader is the part of gensim responsible for downloading pre-trained models.

The imports section is followed by some more setup code:

```
#these are needed to avoid SSL certificate error when downloading models from
gensim api. Especially on MacOS
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
#this is needed to display logging information when loading the model in case
of an error
logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s',
level=logging.INFO)
this_dir = os.path.dirname(os.path.abspath(__file__))
```

this is helper boilerplate code, the ssl part prevents some errors when downloading models on macOS and the logging parts configures the error reporting in gensim. These two sections imports + setup is what we call boilerplate code. It is not part of the logic of our program but just sets up the environment and it’s usually something that you find in the documentation of a library that you are trying to use. So just

copy paste this whole section if you need to use gensim at the beginning of your script.

```
#set to the name of the pretrained model that you want to use (use the list of
available models in 01_gensim_list_pretrained_models.py as reference)
pretrained_model_name = 'glove-wiki-gigaword-50'

#loading a model (this will download the model if it is not already
downloaded, so it may be slow first time you use it)
model : KeyedVectors = gensim.downloader.load(pretrained_model_name)
```

These two lines of code actually load the model to be used. You can change the pretrained_model_name with any of the models we saw when we run the first script.

The load function resides inside the gensim.downloader module and is responsible for downloading a model. Because these models are generally large (several GBytes of data) Gensim will cache them. That means the first time you try to use a model it will be slow because gensim is downloading but all subsequent times you run your script it should be faster as gensim will use a local copy from your hard drive. There still going to be a delay at that point because even when loaded locally it takes a few seconds to rebuild the database in memory from the file.

This code is followed by some helper functions:

```
#here are a few helper methods to work with the word - vectors
#compute the dot product of two vectors
def dot(a, b):
    #we could also use numpy for this operation whicih would be faster for
large vectors
    #sum = np.dot(a,b)
    #but for pedagogical purposes we will use the manual loop here

    sum = 0.0
    for i in range(len(a)):
        sum += a[i]*b[i]

    return sum

#compute the norm of a vector as : sqrt(a.a)
def norm(a):
    return math.sqrt(dot(a,a))

#normalize a vector by dividing all its components by its norm
def normalize(a):
    na = norm(a)
    return a / na

#compute the cosine similarity between two vectors as : a.b/(norm(a)*norm(b))
def similarity(a, b):
    norm_a = norm(a)
```

```
    norm_b = norm(b)
    return dot(a,b)/(norm_a * norm_b)
```

We actually don't need these functions but they are here for pedagogical purposes. Since we are using numpy and we are going to pre-normalize all the word vectors if we wanted to compare the similarity between two vectors "a" and "b" we could just write np.dot(a,b).

However, if you didn't have a library like numpy and for some reason you wanted to compute cosine similarity using only basic arithmetics these functions show you how it would be done. For now, consider this "advanced code". If you are already familiar with python this should be straightforward, but we'll be diving into functions later.

What is important here is that we have a function called similarity that we can give it two vectors "a" and "b" and it will yield a number between +1.0 (identical) to -1.0 (opposites).

A. Vectorizing a single word

```
word = 'whale'
print(f'A Vectorizing a single word:{word}')
#check if word exists
if word in model:
    #get the raw word vector as a numpy array (list of numbers)
    word_vec = model.get_vector(word)
    print(word_vec)

    #get the normalized word vector, We will use this one for most operations
as vector norm is not important for similarity queries
    norm_word_vec = model.get_vector(word, True)
    print(norm_word_vec)
```

in this example we use the most important function in our gensim model "get_vector". This is a function that given a word will give us the vector corresponding to that word. This function can be called in two ways:

```
    word_vec = model.get_vector(word)
```

This one retrieves the "raw" vector as it is stored in the database. This vector may be scaled proportionally to the frequency of corresponding word. In general, when we are doing similarity queries the magnitude of the vector does not matter since we are interested only in the angles between vectors. So we will use a variant of this function that takes a flag (Boolean True/False value) that tells gensim to normalize the vector.

```
    norm_word_vec = model.get_vector(word, True)
```

From now on whenever we want to retrieve the vector of a word from our model, we will be using this function.

B. vectorize 3 words and check their similarity

This example first retrieves the vectors for three words and stores them in three variables:

```
cat = model.get_vector('cat', True)
dog = model.get_vector('dog', True)
mouse = model.get_vector('mouse', True)
```

the names of the variables do not have to match the words, you could name them A,B and C or whatever. It is just so for clarity.

```
print(f'similarity between cat and dog is {similarity(cat, dog)}')
print(f'similarity between cat and mouse is {similarity(cat, mouse)}')
print(f'similarity between dog and mouse is {similarity(dog, mouse)}')
```

then we use our similarity function “similarity(a,b)” that computes the dot product between the word vectors. This will yield a number in the interval [-1.0, +1.0] depending on the relationship between the words. If the similarity of two words is very close to +1.0 then the words are synonyms. If it is close to 0.0 then they are orthogonal, meaning unrelated and if it is -1.0 they point at opposite directions, so they have some oppositional attributes.

C. find the most similar words to a given word

The next example uses the built in functionality in gensim for proximity queries. We can ask Gensim to retrieve any number of the closest words to a given word or vector.

```
word = 'whale'

words = model.most_similar(positive = word, topn=10)
```

the most_similar function takes as input one or more positive terms and optionally one or more negative terms as well as the number of closest neighbours to retrieve. In our case we are getting the 10 closest words to the word “whale”.

If you print these words you will see that this function returns a list of pairs (tuples) :

[
('whales', 0.8986826539039612),
('shark', 0.8336063623428345),
('humpback', 0.8271627426147461),
('dolphin', 0.8136199116706848),
('bird', 0.7804713845252991),
('hunts', 0.7576379179954529),
('birds', 0.7554864883422852),
('mammal', 0.7482953667640686),
('sturgeon', 0.7462471723556519),
('elephant', 0.7432060837745667)
]

The words are returned from the most to the list similar (higher cosine similarity to lowest) as pairs (“word”, similarity).

So if you wanted to get the most similar word it would be **words[0][0]**, that is the first element of the pair at the first position in the list:

Words[list position][pair index]

If you wanted the similarity number for that word it would be **words[0][1]**

D. find the most similar words to a given word, but exclude another word

Here we are using the variant of the most_Similar function that takes as input both positive and negative terms. In effect we ask the model to find the closest neighbors in the direction of the positive term but away from the negative one. Internally it simply takes the vectors for the negative terms multiplies them by -1.0 reversing their direction and adding them to the vectors for the positive terms.

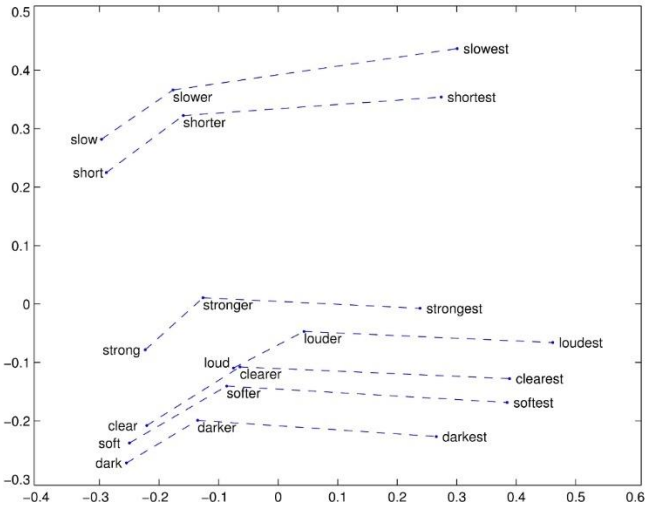
```
positive_term = 'whale'
negative_term = 'water'
words = model.most_similar(positive = positive_term, negative=negative_term,
topn=10)
print(f'the 10 most similar words to {positive_term} opposite {negative_term}
are:')
```

This can yield nonsensical results as sometimes it can land at a word-desert (a place in the embedding space where junk words or no words exist)

E. compute an analogy : Find the word that is to A what C is to B

In this example we are computing an analogy given three words A,B and C. We are trying to find a word that is to A what C is to B.

The mental model to keep in mind is the image from Glove's web site:



The idea is that similar relationships will be represented by parallel shifts in the embedding space.

So for the words A = 'good', B='bad' and C='worse' we want to move A_{good} along the B_{bad} to C_{worse} axis. This is not the axis of worsening but rather the axis of superlatives.

The expression (C_{worse}-B_{bad}) is asking of what is left if we subtract badness from the term worse. What is the semantic difference of the two terms, and that is the act of increasing intensity.

So if we have the vectors for good, bad and worse we can write:

$A_{good} + (C_{worse}-B_{bad})$

Which can be interpreted as move A along the direction B->C.

Subtracting two vectors (C-B) gives you the vector that would take you from B to C.

Then adding the vector (C-B) to A moves it along that direction.

The above expression can be written also as:

$A_{good} - B_{bad} + C_{worse}$

Here's the code:

```
a = model.get_vector(A, True)
b = model.get_vector(B, True)
c = model.get_vector(C, True)
vec = a-b+c
vec = normalize(vec)
results = model.similar_by_vector(vec, topn=10)
```

When we compute a-b+c this is vector arithmetic, so we just get a new vector. We do not know if this vector points to some valid word in the embedding space. The space of all possible vectors is continuous and vast while the words form tiny speckles within that space. There are vast semantic oceans with no words nearby, you can think of it as concepts with no single word terms associated with them.

Some of the points in this space might be approximated by compound words A+B+C… or whole sentences.

So once we have our vector "vec" we ask the model to check if there are any words nearby by using the "similar_by_vector" function. This function works the same way as the "most_similar" function we used previously but instead of taking as its first argument a word it takes a vector that may or may not correspond to a valid word.

I the given example we also use the normalize method to normalize the composite vector but that is not necessary as "similar_by_vector" internally uses cosine distance to find the closest words and therefore normalization doesn't matter.

So you can omit the line vec=normalize(vec)

F. Sort words by similarity to a given word

The final example is the most involved coding wise.

Here we are given a reference word and a list of words. We want to project the list of words on the direction of the reference word and order them by their similarity to that word.

That means we need to compute the similarity of each word from the list to the reference word and then sort the list by similarity.

```
words = ['art' , 'science', 'religion', 'politics', 'economics', 'building',
'urban', 'dog', 'cat', 'ant', 'bee', 'landscape', 'site', 'material',
```

```
'modern', 'contemporary', 'engineering', 'good', 'bad', 'ugly', 'beautiful',
'design', 'city', 'country', 'authoritarian', 'democratic']
ref_word = 'architecture'
```

we start by vectorizing the reference word:

```
ref_word_vec = model.get_vector(ref_word, True)
```

then we create an empty list called “sims” to store the similarities:

```
sims = []
```

then we create a loop that iterates over the words, and for each word w it vectorizes it and computes its similarity to the reference word:

```
for w in words:
    vec = model.get_vector(w, True)
    sim = similarity(ref_word_vec, vec)
    pair = (w, sim)
    sims.append(pair)
```

the syntax pair = (w,sim) is using the tuple constructor in Python to package a word with its similarity.

Because we want to sort the words by similarity we need to somehow store in the list both the word and its associated similarity number. So as we compute the similarity for each word we package the word and its similarity into a single tuple, and add the tuple to the list. The data structure of our list will look like this:

[(art, 0.8012039987740273), (design, 0.7926884115806803), …]

So “sims” is a list of tuples, where each tuple consists of a string (the word) and a number (its similarity to the reference word)

Now we can use the list sorting method to sort the data:

```
sims.sort(key=lambda x: x[1], reverse=True)
```

Python offers several ways to sort containers. You can find more information here:

https://docs.python.org/3/howto/sorting.html

in our case we are suing the sort method of the list objects. “sims” is a list because we created it using sims = [ ].

Because it is a list it has a method for sorting itself. However, we need to tell it what is the criteria for sorting.

The sort method is called on the list we want to sort using the dot notation sims.sort()

If our list had only numbers or strings we could simply call sims.sort(), However, since our list consists of tuples we need to tell it whether it should sort by the first or second element in each tuple (the word or the similarity).

We do this by using the “key:” argument which takes as input a function (also called a lambda). A lambda is a way to define a function or operation locally that will be applied to each element in the list and the result of which will be used for sorting.

```
lambda x: x[1]
```

is a function that takes as input a tuple x = (word,sim) pair and returns the second element x[1] of that pair. So as the sorting algorithm walks through our list it will only check the second element of each pair to decide on the ordering.

Lambdas are a more advanced feature of python and we won’t go into more detail for now. Try to change x[1] to x[0] and see how that affects the result…

Finally we can print the result :

```
for w, sim in sims:
    print(f'{w} : {sim}')
```

to get:

modern : 0.8539479281644627

art : 0.8012039987740273

design : 0.7926884115806803

contemporary : 0.7650543585538946

landscape : 0.6724785768590393

building : 0.6509127566202846

science : 0.6437532992654037

engineering : 0.6290409221117592

urban : 0.6041912163082619

economics : 0.5598428766419822

03_gensim_train

This script is needed if you want to train your own model using word2vec on a book or collection of text of your own.

The parameters you need to adjust are in this section:

```
#the input text file to train the model on
input_text_file = 'moby_dick.txt'
#the output file name for the model. the extension .kv is signifies a keyed
vector file (that is a word2vec dictionary that maps words to vectors)
output_file = 'moby_dick.kv'
#the size of the word vectors. the higher the number the more accurate the
model is (sort of more semantic resolution, differentiating distinct
direcitons in the embedding vector space), but also the more computationally
expensive and memory intensive when loaded
vector_size = 150
#the number of epochs to train the model. the training process is iterative.
```

```
#there is no hard rule for the number of epochs. the more the better, but the more epochs the longer it takes to train the model
#how many epochs you need depends on the size of the corpus. for small corpora generally you need more epochs
#start with 100 and then increase if you are not satisfied with the results
epochs = 50
```

some public domain books are already provided as txt files so you can test it with those. If you want to use your own text to train you need to coalesce all the books or text into a single txt file saved in UTF8 encoding (usually most text editors have an option for this when exporting text).

Save the txt file in the same folder as your script, then modify the input_text_file and output_file variables accordingly.

When you run the script, training will begin. The model will read through the text multiple times and adjust the vectors of all found words as it goes. At the end will save the word-vec database into the output_file.

04_gensim_use_trained

This file shows you how to use a model that you trained yourself. You can do with a trained model what we did in the 02 script with the pretrained model. They are the same data structure the only difference is where they are loaded from. So you can load your model using:

```
model : KeyedVectors = KeyedVectors.load(custom_model_name, mmap='r')
```

where custom_model_name is the filename of the model you trained in your harddrive.

Exercises

1. Semantic axis

Write a script that produces an output like below.

```
Axis: rural -> urban : building + (urban - rural)
          building:(0.82)
                 tower:(0.70)
                  buildings:(0.70)
                   design:(0.68)
                   skyscraper:(0.67)
                    towers:(0.65)
                     structure:(0.64)
                      construction:(0.63)
```

a. First load a model (e.g. 'glove-wiki-gigaword-100')

b. select a world that will form the starting central concept to be translated (e.g. ‘building’)

c. select two words to form an axis of translation (e.g. rural->urban)

d. get the vectors for the three words from the model

e. calculate the translation vector by subtracting the two axis word vectors (e.g. urban_vec – rural_vec)

f. create the translated vector by adding the translation to the central concept (e.g. building_vec + (urban_vec – rural_vec)

g. query the model using the similar_by_vector method for the 8 closest matches to the translated vector

h. print the results using the shown pattern.

i. bonus if you make the number of spaces (indentation) of each word to be related to its similarity value.

> Use your own words, give at least 3 examples with different word combinations. For each example try also to reverse the direction by changing the order of the subtracted words. (e.g. try both urban->rural and rural->urban)

2. Bi-axial semantics

For this exercise, you need to start with a central concept C and two axes X1->X2 and Y1->Y2 (total 5 words)

Form the axis vectors using the model as in assignment 1:
C_vec
X_vec = X2_vec - X1_vec
Y_vec = Y2_vec – Y1_vec

Know repeat the assignment 1 by computing the following translations:
C + 1*X + 0*Y
C + 1*X + 1*Y
C + 0*X + 1*Y
C - 1*X + 1*Y
C - 1*X + 0*Y
C - 1*X - 1*Y
C + 0*X - 1*Y

for example, for C=’architecture’ with the axes {life->death} and {natural->artificial}

we get something like the following:

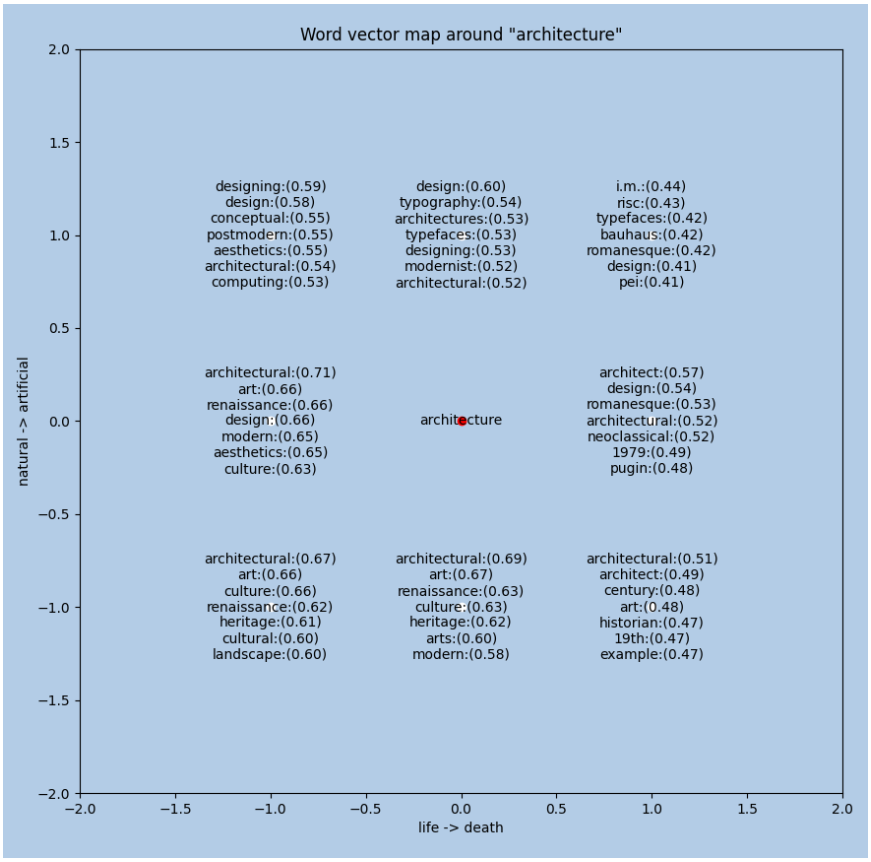
```
Direction 1.0*life -> death + 0.0*natural -> artificial:
                  architecture:(0.69)
                          architect:(0.57)
                           design:(0.54)
                            romanesque:(0.53)
                            architectural:(0.52)
                            neoclassical:(0.52)
                              1979:(0.49)
                               pugin:(0.48)


Direction 1.0*life -> death + 1.0*natural -> artificial:
                                architecture:(0.46)
                                 i.m.:(0.44)
                                  risc:(0.43)
                                  typefaces:(0.42)
                                  bauhaus:(0.42)
                                   romanesque:(0.42)
                                   design:(0.41)
                                   pei:(0.41)


Direction 0.0*life -> death + 1.0*natural -> artificial:
                  architecture:(0.71)
                        design:(0.60)
                            typography:(0.54)
                             architectures:(0.53)
                             typefaces:(0.53)
                             designing:(0.53)
                             modernist:(0.52)
                              architectural:(0.52)


Direction -1.0*life -> death + 1.0*natural -> artificial:
                  architecture:(0.70)
                        designing:(0.59)
                         design:(0.58)
```

These data conceptually allow us to explore a planar territory around a concept like in the following diagram:



Use the data that you calculated to create a diagram of the results. You can use any software or method you want for the graphical elements. The above example uses

matplotlib directly in python, but you don’t have to. You can use CAD, illustrator, hand-drawn or python. Try to give a character to the diagram that relates to the concepts that you used and the relative strength of the similarities (for example different color, opacity or font size depending on the similarity value, or slightly different placement)