Natural Language Processing and Large Language Models

Corso di Laurea Magistrale in Ingegneria Informatica

Lesson 5

Word Embeddings

Nicola Capuano and Antonio Greco

DIEM – University of Salerno

Outline

- Limitations of TF-IDF
- Word Embeddings
- Learning Word Embeddings
- Word2Vec Alternatives
- Working with Word Embeddings

Limitations of TF-IDF

Limitations of TF-IDF

TF-IDF counts terms according to their **exact spelling**

- Texts with the **same meaning** will have completely **different TF-IDF vector** representations **if they use different words**

Examples:

- *The movie was amazing and exciting*
- *The film was incredible and thrilling*
- *The team conducted a detailed analysis of the data and found significant correlations between variables*
- *The group performed an in-depth examination of the information and discovered notable relationships among factors*

Term Normalization

Techniques like **stemming** and **lemmatization** help normalize terms

- Words with **similar spellings** are collected under a **single token**

Disadvantages

- They **fails to group most synonyms**
- They may group together words with **similar/same spelling** but **different meanings**
  - *She is **leading** the project* **vs** *The plumber **leads** the pipe*
  - *The **bat** flew out of the cave* **vs** *He hit the ball with a **bat***

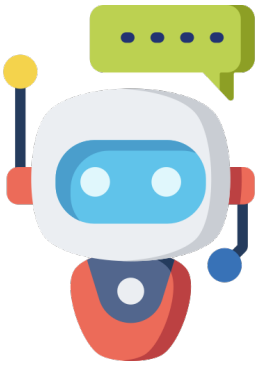
TF-IDF Applications

TF-IDF is **sufficient for many NLP applications**

- Information Retrieval (Search Engines)
- Information Filtering (Document Recommendation)
- Text Classification

Other applications require a **deeper understanding of text semantics**

- Text generation (Chatbot)
- Automatic Translation
- Question Answering
- Paraphrasing and Text Rewriting



Bag-of-Words (recap)

Each word is assigned an **index** that represents its **position in the vocabulary**

- the 1st word (e.g., **apple**) has index **0**
- the 2nd word (e.g., **banana**) has index **1**
- the 3rd word (e.g., **king**) has index **2**
- …

Each word is then represented by a **one-hot vector**

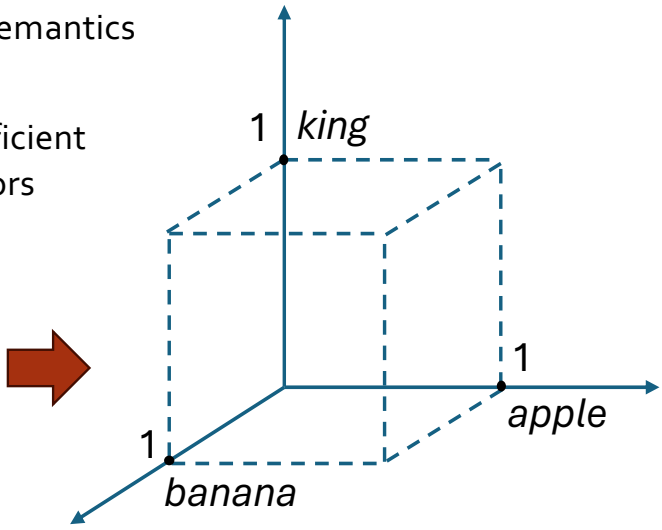
- **apple = (1,0,0,0,…,0)**
- **banana = (0,1,0,0,…,0)**
- **king = (0,0,1,0,…,0)**

Bag-of-Words (recap)

With this encoding, **the distance between any pair of vectors is always the same**

- It does not capture the semantics of words
- Furthermore, it is not efficient since it uses sparse vectors

Note: the figure shows only **three dimensions** of a space where dimensions equals the **cardinality of the vocabulary**



Word Embeddings

Word Embeddings

A technique for representing **words with vectors** (A.K.A. **Word Vectors**) that are:

- **Dense**
- **With dimensions much smaller than the vocabulary size**
- **In a continuous vector space**

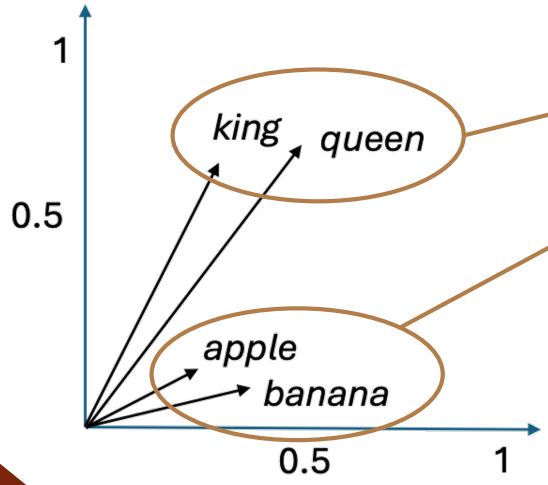
Key feature:

- Vectors are generated so that **words with similar meanings are close to each other**
- The **position** in the space represents the **semantics** of the word

Word Embeddings

Example:

- Apple = (0.25,0.16)
- Banana = (0.33,0.10)
- King = (0.29,0.68)
- Queen = (0.51,0.71)



king and **queen** are close to each other

apple and **banana** are close to each other

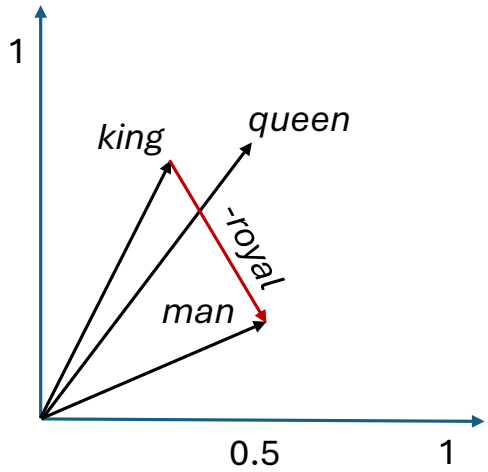
The words of the **first group** are far from those of to the **second group**

Word Embedding: Properties

Word embeddings enable **semantic text reasoning** based on **vector arithmetic**

Examples:

- Subtracting **royal** from **king** we arrive close to **man**



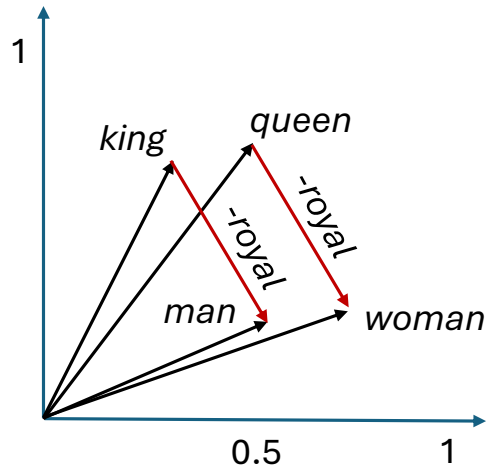
king – royal

Word Embedding: Properties

Word embeddings enable **semantic text reasoning** based on **vector arithmetic**

Examples:

- Subtracting **royal** from **king** we arrive close to **man**
- Subtracting **royal** from **queen** we arrive close to **woman**



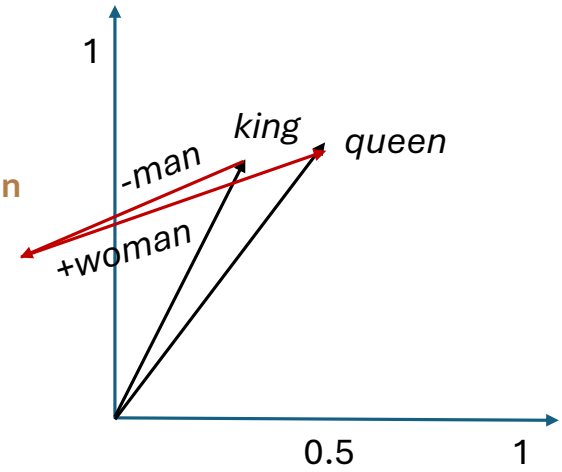
queen – royal

Word Embedding: Properties

Word embeddings enable **semantic text reasoning** based on **vector arithmetic**

Examples:

- Subtracting **royal** from **king** we arrive close to **man**
- Subtracting **royal** from **queen** we arrive close to **woman**
- Subtracting **man** from **king** and adding **woman** we arrive close to **queen**



king – man + woman

Semantic Queries

Word embeddings allow for searching words or names by **interpreting the semantic meaning of a query**

Examples:

- Query: "Famous European woman physicist"

```
wv['famous'] + wv['European'] + wv['woman'] + wv['physicist']
≈ wv['Marie_Curie'] ≈ ['Lise_Meitner'] ≈ …
```

- Query: "Popular American rock band"

```
wv['popular'] + wv['American'] + wv['rock'] + wv['band']
≈ wv['Nirvana'] ≈ wv['Pearl Jam'] ≈ …
```

Analogies

Word embeddings enable answering **analogy questions** by leveraging their semantic relationships

Examples:

- Who is to physics what Louis Pasteur is to germs?

```
wv['Louis_Pasteur'] - wv['germs'] + wv['physics']
≈ wv['Marie_Curie']
```

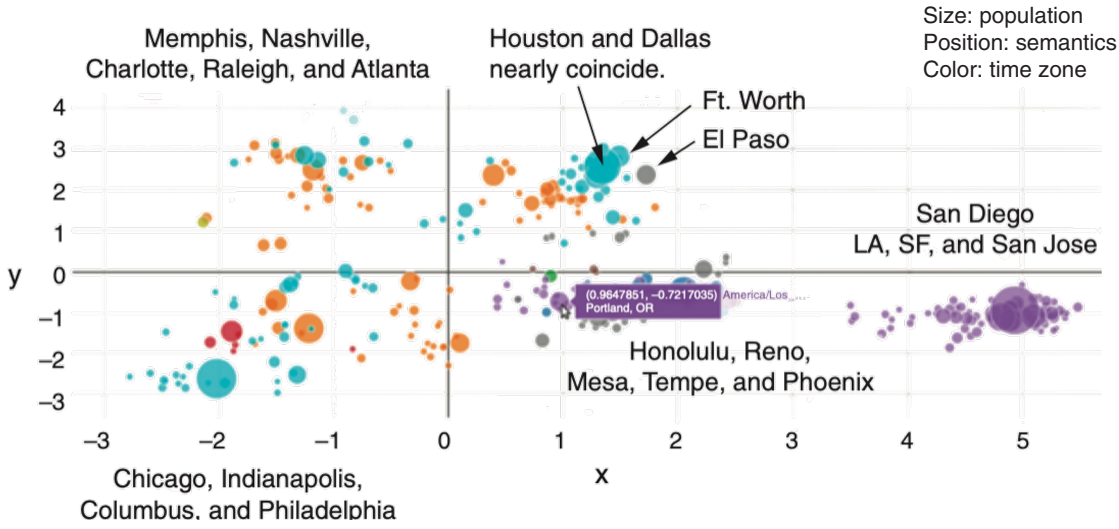
- Marie Curie is to science as who is to music?

```
wv['Marie_Curie'] - wv['science'] + wv['music']
≈ wv['Ludwig_van_Beethoven']
```

- Legs is to walk as mouth is to what?

```
wv['legs'] - wv['walk'] + wv['mouth']
≈ wv['speak'] or wv['eat']
```

Visualizing Word Embeddings



Google News Word2vec **300-D vectors** projected onto a 2D map using PCA

Semantic proximity approximates **geographical proximity**

Learning Word Embeddings

Word2Vec

Word embeddings was introduced by **Google** in **2013** in the following paper

- T. Mikolov, K. Chen, G. Corrado, and J. Dean, *Efficient estimation of word representations in vector space* in 1st International Conference on Learning Representations, ICLR 2013

The paper defines **Word2Vec**

- A methodology for the **generation of word embeddings**
- Based on **neural networks**
- Using **unsupervised learning** on a large **unlabeled textual corpus**

Word2Vec

Idea: words with **similar meanings** are often found in **similar contexts**

- **Context: a sequence of words in a sentence**

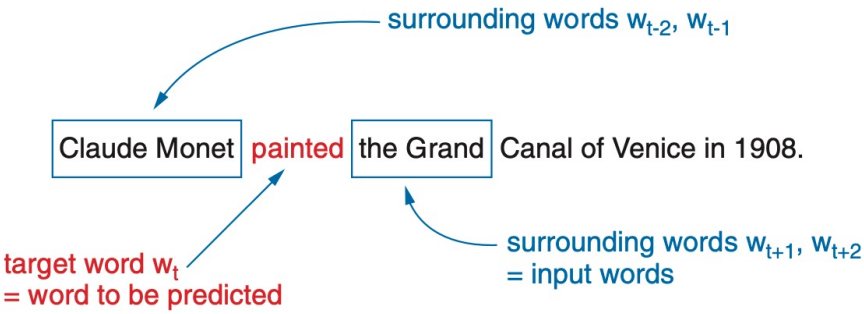
Example:

- Consider the sentence ~~Apple~~ **juice is delicious**
- Remove one word
- The remaining sentence is **____ juice is delicious**
- Ask someone to **guess the missing word**
  - Terms such as **banana**, **pear** or **apple** would probably be suggested
  - These terms have **similar meanings** and used in **similar contexts**

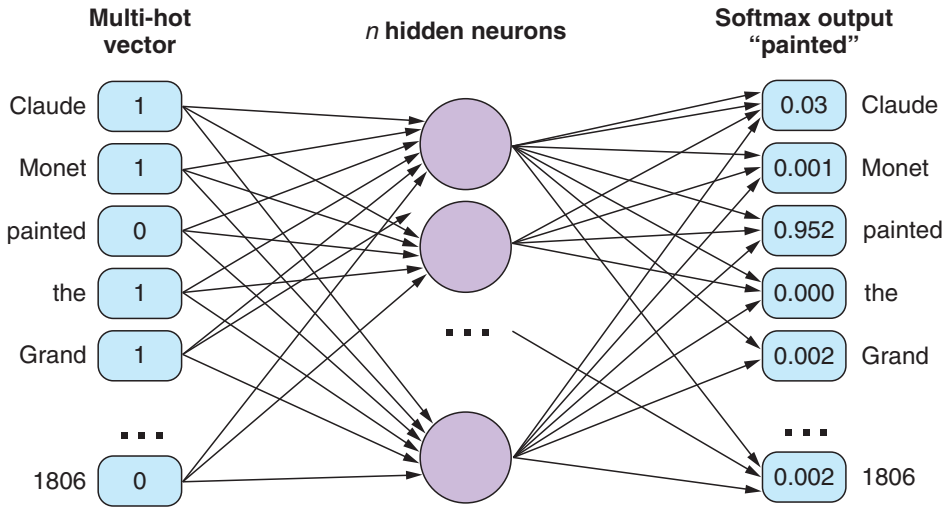
Continuous Bag-of-Word

A neural network is trained to **predict the central token of a context of m tokens**

- **Input:** the bag of words composed of the sum of all one-hot vectors of the **surrounding tokens**
- **Output:** a probability distribution over the vocabulary with its maximum in the **most probable missing word**
- **Example:** *Claude Monet **painted** the Grand Canal in Venice in 1806*



Continuous Bag-of-Word



$|V|$ **input** and **output** neurons where **V is the vocabulary**

n hidden neurons where **n is the word embedding dimension**

Continuous Bag-of-Word

Ten **5-gram examples** from the sentence about Monet

Input word w_{t-2}	Input word w_{t-1}	Input word w_{t+1}	Input word w_{t+2}	Expected output w_t
		Monet	painted	Claude
	Claude	painted	the	Monet
Claude	Monet	the	Grand	painted
Monet	painted	Grand	Canal	the
painted	the	Canal	of	Grand
the	Grand	of	Venice	Canal
Grand	Canal	Venice	in	of
Canal	of	in	1908	Venice
of	Venice	1908		in
Venice	in			1908

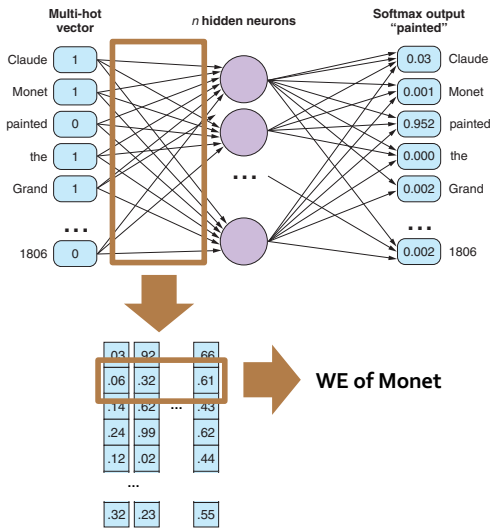
Continuous Bag-of-Word

After the training is complete **the output layer of the network is discarded**

- Only the **weights of the inputs to the hidden layer** are important
- They represent the **semantic meaning of words**

Similar words are found in similar contexts …

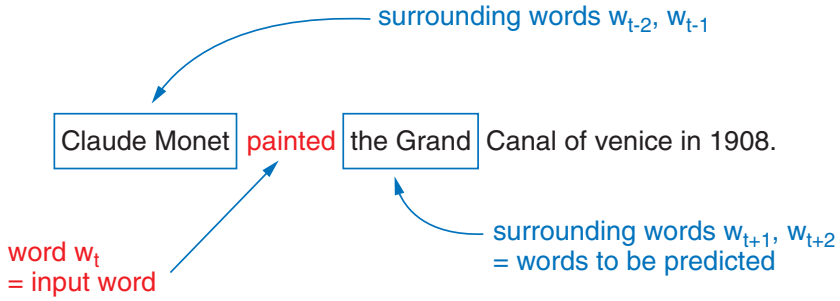
- … their **weights to the hidden layer adjust in similar ways**
- … this result in **similar vector representations**



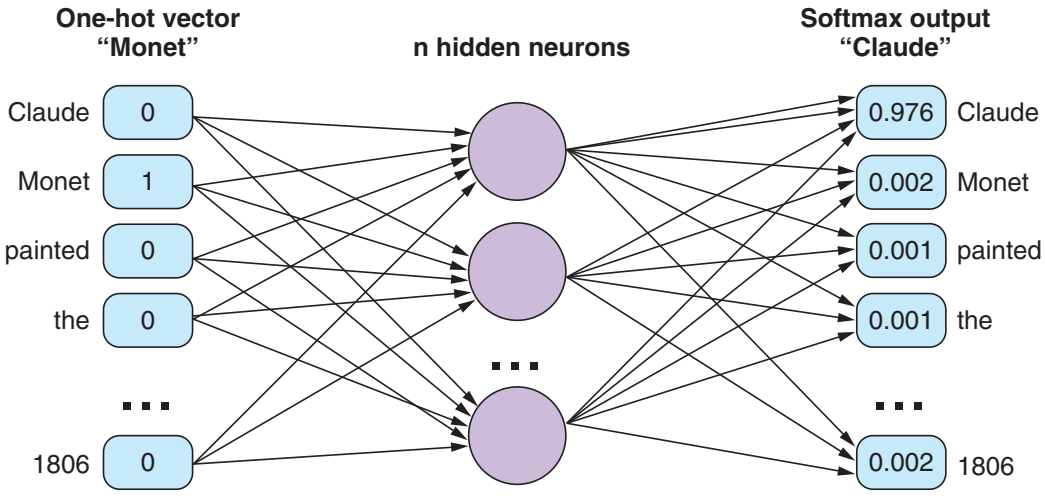
Skip-Gram

Alternative training method for **Word2Vec**

- A neural network is trained to **predict a context of m tokens based on the central token**
- **Input:** the one-hot vector of the **central token**
- **Output:** the one-hot vector of a **surrounding word** (one training iteration for each surrounding word)



Skip-Gram



$|V|$ **input** and **output** neurons where **V is the vocabulary**

n hidden neurons where **n is the word embedding dimension**

Skip-Gram

Ten **5-gram examples** from the sentence about Monet

Input word w_t	Expected output w_{t-2}	Expected output w_{t-1}	Expected output w_{t+1}	Expected output w_{t+2}
Claude			Monet	painted
Monet		Claude	painted	the
painted	Claude	Monet	the	Grand
the	Monet	painted	Grand	Canal
Grand	painted	the	Canal	of
Canal	the	Grand	of	Venice
of	Grand	Canal	Venice	in
Venice	Canal	of	in	1908
in	of	Venice	1908	
1908	Venice	in		

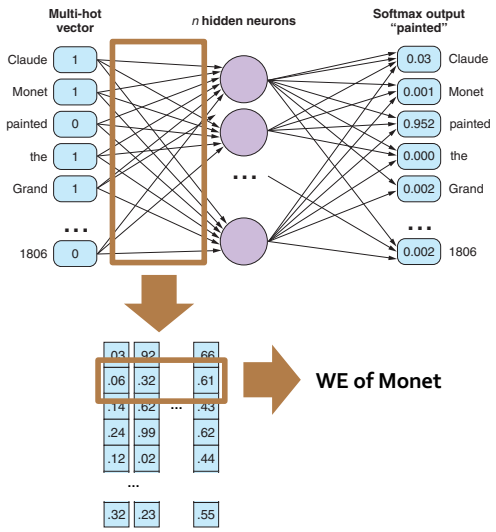
Skip-Gram

After the training is complete **the output layer of the network is discarded**

- Only the **weights of the inputs to the hidden layer** are important
- They represent the **semantic meaning of words**

Similar words are found in similar contexts …

- … their **weights to the hidden layer adjust in similar ways**
- … this result in **similar vector representations**



CBOW vs Skip-Gram

CBOW

- Higher accuracies for frequent words, much faster to train, suitable for large datasets

Skip-Gram

- Works well with small corpora and rare terms

Dimension of Embeddings (n)

- Large enough to **capture the semantic meaning of tokens** for the specific task
- Not so large that it results in **excessive computational expense**

Improvements to Word2Vec

Frequent Bigrams

- Some words often **occur in combination**
  - **Elvis** is often followed by **Presley** forming a **bigram**
  - Predicting Presley after Elvis **doesn't add much value**
- To let the network focus on **useful predictions** **frequent bigrams and trigrams are included as terms** in the Word2vec vocabulary
- **Inclusion criteria**: **co-occurrence frequency** greater than a threshold

$$score(w_i, w_j) = \frac{count(w_i, w_j) - \delta}{count(w_i) \times count(w_j)}$$

Examples:

- **Elvis_Presley**, **New_York**, **Chicago_Bulls**, **Los_Angeles_Lakers**, etc.

Improvements to Word2Vec

Subsampling Frequent Tokens

- **Common words** (like **stop-words**) often don't carry significant information
- Being **frequent**, they have a big **influence on the training process**

To reduce their effect…

- During training (**skip-gram method**), words are sampled in **inverse proportion to their frequency**
- **Probability of sampling:** $P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$
- The effect is like the **IDF effect on TF-IDF vectors**

Improvements to Word2Vec

Negative Sampling

- Each training example causes the network to **update all weights**
- With thousands or millions of words in the vocabulary, this makes the process **computationally expensive**

Instead of updating all weights…

- Select 5 to 20 **negative words** (words not in the context)
- **Update weights** only for the **negative words** and the **target word**
- Negative words are **selected based on their frequency**
  - Common words are chosen more often than rare words
- The **quality of embeddings** in maintained

Word2Vec Alternatives

GloVe

Global Vectors for Word Representation

- Introduced by researchers from **Stanford University** in 2014
- Uses classical optimization methods like **Singular Value Decomposition** instead of neural networks

Advantages:

- **Comparable precision** to Word2Vec
- Significantly **faster training times**
- Effective on **small corpora**

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

FastText

Introduced by **Facebook** researchers in 2017

- Based on **sub-words**, predicts the surrounding **n-character grams** rather than the surrounding words
  - **Example:** the word **whisper** would generate the following 2- and 3-character grams: **wh, whi, hi, his, is, isp, sp, spe, pe, per, er**

Advantages:

- Particularly **effective for rare or compound words**
- Can handle **misspelled words**
- Available in **157 languages**

https://fasttext.cc/

Static Embeddings

Word2Vec, **GloVe**, **FastText** are **Static Embeddings**

- Each word is represented by a **single static vector** that captures the **average meaning** of the word based on the training corpus
- Once trained, vectors **do not change** based on context
- This does not account for **polysemy** and **homonymy**

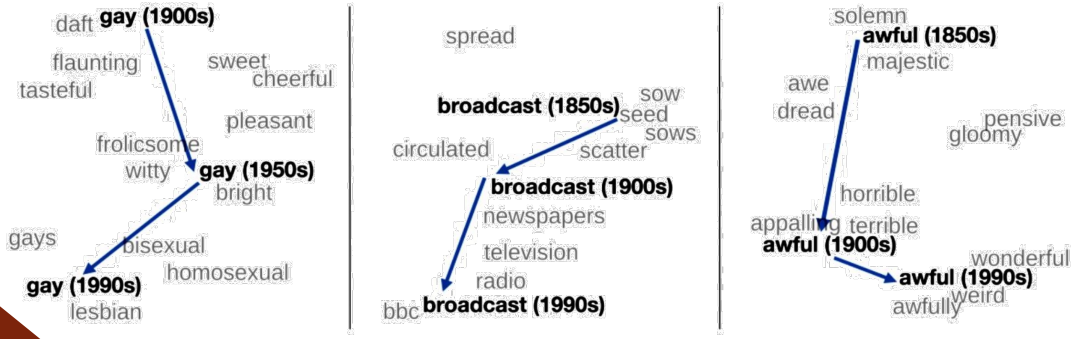
Example:

- The word **apple** could refer to the **fruit**, the **tech company**, or even a **popular song (ABBA)**
- A word embedding would **blend these meanings into a single vector**, failing to capture the specific context

Semantic Drift

The **meanings of words changes over time**, posing a challenge for **static embeddings**

- **Gay** once meant **cheerful** but now primarily refers to **homosexuality**
- **Broadcast** shifted from **casting out seeds** to **transmitting signals** with the advent of radio and TV
- **Awful** changed from **full of awe** to **terrible or appalling**



Social Stereotypes

Word embeddings can perpetuate and amplify **societal biases** present in the training data

- **Man is to Doctor as Woman is to… Nurse**

Examples of **gender**, **racial**, and **religious** biases in analogies generated from word embeddings trained on the **Reddit** data from users from the **USA**

Black is to Criminal as Caucasian is to Police: Detecting and Removing Multiclass Bias in Word Embeddings

Gender Biased Analogies	
man → doctor	woman → nurse
woman → receptionist	man → supervisor
woman → secretary	man → principal
Racially Biased Analogies	
black → criminal	caucasian → police
asian → doctor	caucasian → dad
caucasian → leader	black → led
Religiously Biased Analogies	
muslim → terrorist	christian → civilians
jewish → philanthropist	christian → stooge
christian → unemployed	jewish → pensioners

Other Issues of WEs

Out-of-Vocabulary words

- Traditional WEs cannot handle **unknown words**
- They are limited to the **words present in the training data**
- Models based on **sub-words** (like **FastText**) can handle this

Lack of transparency

- It can be difficult to interpret the **meaning of individual dimensions** or word vectors
- Difficult to **analyze** and **improve** the model, ensure its **fairness**, and **explain its behavior** to stakeholders

Contextual Embeddings

Contextual Embeddings can be updated based on the **context of surrounding words**

- Context is used both during **training** and **usage**
- Effective for applications that need deep **language understanding**

  The embedding for **not happy** is closer to **unhappy** than in static embeddings

Examples:

- **ELMo** (Embeddings from Language Model) – 2018
- **BERT** (Bi-directional Encoder Representations for Transformers) – 2020
- Many others, based on **transformers…** we will see them **later**

Working with Word Embeddings

Load Pre-trained WEs

Gensim is a popular Python library for NLP supporting various **word embedding models**

- https://radimrehurek.com/gensim/
- `pip install gensim`


```python
import gensim.downloader as api

# Download word embeddings pre-trained on a part of the Google News dataset
# The model contains 300-dimensional vectors for 3 million words and phrases
# about 1.6 Gb data will be downloaded (only the first time)

wv = api.load('word2vec-google-news-300')

# print the first 10 words in the model
for i, word in enumerate(wv.index_to_key[:10]):
    print("Word {} is {}".format(i, word))
```

```
Word 0 is </s>
Word 1 is in
Word 2 is for
Word 3 is that
Word 4 is is
Word 5 is on
Word 6 is ##
Word 7 is The
Word 8 is with
Word 9 is said
```

Get a Word Vector

```
wv['king'] # 300-dimensional vector
```

```
array([ 1.25976562e-01,  2.97851562e-02,  8.60595703e-03,  1.39648438e-01,
       -2.56347656e-02, -3.61328125e-02,  1.11816406e-01, -1.98242188e-01,
        5.12695312e-02,  3.63281250e-01, -2.42187500e-01, -3.02734375e-01,
       -1.77734375e-01, -2.49023438e-02, -1.67968750e-01, -1.69921875e-01,
        3.46679688e-02,  5.21850586e-03,  4.63867188e-02,  1.28906250e-01,
        1.36718750e-01,  1.12792969e-01,  5.95703125e-02,  1.36718750e-01,
        1.01074219e-01, -1.76757812e-01, -2.51953125e-01,  5.98144531e-02,
        3.41796875e-01, -3.11279297e-02,  1.04492188e-01,  6.17675781e-02,
        1.24511719e-01,  4.00390625e-01, -3.22265625e-01,  8.39843750e-02,
        3.90625000e-02,  5.85937500e-03,  7.03125000e-02,  1.72851562e-01,
        1.38671875e-01, -2.31445312e-01,  2.83203125e-01,  1.42578125e-01,
        3.41796875e-01, -2.39257812e-02, -1.09863281e-01,  3.32031250e-02,
       -5.46875000e-02,  1.53198242e-02, -1.62109375e-01,  1.58203125e-01,
       -2.59765625e-01,  2.01416016e-02, -1.63085938e-01,  1.35803223e-03,
       -1.44531250e-01, -5.68847656e-02,  4.29687500e-02, -2.46582031e-02,
        1.85546875e-01,  4.47265625e-01,  9.58251953e-03,  1.31835938e-01,
        9.86328125e-02, -1.85546875e-01, -1.00097656e-01, -1.33789062e-01,
       -1.25000000e-01,  2.83203125e-01,  1.23046875e-01,  5.32226562e-02,
       -1.77734375e-01,  8.59375000e-02, -2.18505859e-02,  2.05078125e-02,
       -1.39648438e-01,  2.51464844e-02,  1.38671875e-01, -1.05468750e-01,
        1.38671875e-01,  8.88671875e-02, -7.51953125e-02, -2.13623047e-02,
        1.72851562e-01,  4.63867188e-02, -2.65625000e-01,  8.91113281e-03,
        1.49414062e-01,  3.78417969e-02,  2.38281250e-01, -1.24511719e-01,
       -2.17773438e-01, -1.81640625e-01,  2.97851562e-02,  5.71289062e-02,

       ...
```

Similarity Between Words

```
pairs = [
    ('car', 'minivan'),   # a minivan is a kind of car
    ('car', 'bicycle'),   # still a wheeled vehicle
    ('car', 'airplane'),  # ok, no wheels, but still a vehicle
    ('car', 'cereal'),    # ... and so on
    ('car', 'communism'),
]
for w1, w2 in pairs:
    print("{:<6} {:<12} {:.2f}".format(w1, w2, wv.similarity(w1, w2)))
```

```
car    minivan      0.69
car    bicycle      0.54
car    airplane     0.42
car    cereal       0.14
car    communism    0.06
```

Compute similarity between words

Operations with WEs

```
# most_similar find the top-N most similar keys. Positive keys contribute
# positively towards the similarity, negative keys negatively.
# Uses cosine similarity between word vectors.

# king + woman - man = queen
print(wv.most_similar(positive = ['king', 'woman'], negative = ['man'], topn = 2))

# Louis_Pasteur is to science as ... is to music
print(wv.most_similar(positive = ['Louis_Pasteur', 'music'], negative = ['science'], topn = 2))

# Words most similar to 'cat'
print(wv.most_similar('cat', topn = 3))

# Which key from the given list doesn't go with the others (doesnt_match)?
print(wv.doesnt_match("breakfast cereal dinner lunch".split()))
```

```
[('queen', 0.7118192911148071), ('monarch', 0.6189674735069275)]
[('guitarist_Django_Reinhardt', 0.48056140542030334), ('Lester_Lanin', 0.47759246826171875)]
[('cats', 0.8099378347396851), ('dog', 0.760945737361908), ('kitten', 0.7464984655380249)]
cereal
```

Change the WE Model

Gensim comes with several **pre-trained models**

- fasttext-wiki-news-subwords-300
- conceptnet-numberbatch-17-06-300
- word2vec-ruscorpora-300
- word2vec-google-news-300
- glove-wiki-gigaword-50
- glove-wiki-gigaword-100
- glove-wiki-gigaword-200
- glove-wiki-gigaword-300
- glove-twitter-25
- glove-twitter-50
- glove-twitter-100
- glove-twitter-200

You can also import an **external model**

```
# Download Italian WE from https://mlunicampania.gitlab.io/italian-word2vec/ (700Mb)
# Trained on Italian Wikipedia, contains 300-dimensional vectors for 618.224 words

from gensim.models import KeyedVectors
wv_ita = KeyedVectors.load("W2V.kv")
```

Using Italian WEs

```
pairs = [
    ('automobile', 'suv'),        # a suv is a kind of car
    ('automobile', 'bicicletta'), # still a wheeled vehicle
    ('automobile', 'aereo'),      # ok, no wheels, but still a vehicle
    ('automobile', 'cereali'),    # ... and so on
    ('automobile', 'comunismo'),
]
for w1, w2 in pairs:
    print("{:<12} {:<12} {:.2f}".format(w1, w2, wv_ita.similarity(w1, w2)))
```

```
automobile   suv          0.59
automobile   bicicletta   0.53
automobile   aereo        0.45
automobile   cereali      0.06
automobile   comunismo    0.16
```

Compute similarity between words

Train a WE Model

```
import nltk
from nltk.corpus import reuters
from gensim.models import Word2Vec

# Dwonload the Reuters corpus
nltk.download('reuters')

# Extract and tokenize training documents (we use the "words" method)
training_ids = [id for id in reuters.fileids() if id.startswith("training")]
training_corpus = [list(reuters.words(id)) for id in training_ids]

# Training a word2vec model on the Reuters corpus
model = Word2Vec(sentences = training_corpus, vector_size = 100, window = 5, min_count = 5, workers = 4)

# print the first 10 words in the model
for i, word in enumerate(model.wv.index_to_key[:10]):
    print("Word {} is {}".format(i, word))

# Save the model
model.save("reuters_w2v")
```

Dimensions of word vectors

Context window

Min occurrences of a word to be considered

Number of threads

Out-of-Vocabulary Words

Before using a **WE** you should **check if the token is present in the vocabulary**

- Usually **out-of-vocabulary words are discarded**

```
def check_print(model, word):
    print(model.wv[word][:10] if word in model.wv else "Not present.")

# Check if some words are present in the model
check_print(model, 'nation')
check_print(model, 'kingdom')
check_print(model, 'king') # not present
```

```
[ 0.4143482   0.23261763 -0.1324786  -0.17583102 -0.44117942 -0.61939853
  0.28938904  0.72226125  0.15422714 -0.8243338 ]
[ 0.12139592  0.10290138 -0.03114202  0.00311482 -0.15570888 -0.23540567
  0.14787677  0.26050693  0.017613   -0.27656934]
Not present.
```

Using FastText

FastText can generate embeddings for **unknown words**

- Each word is treated as an **aggregation of sub-words**
- WEs are generated based on the **word's morphological structure**

```
from gensim.models import FastText

# Training a FastText model on the Reuters corpus
model_ft = FastText(sentences = training_corpus, vector_size = 100, window = 5, min_count = 5, workers = 4)

# Check if some words are present in the model
check_print(model_ft, 'nation')
check_print(model_ft, 'kingdom')
check_print(model_ft, 'king') # generated by FastText
```

```
[-0.5036252  -2.6121807  -0.86733997  2.1884098  -2.1722329   0.251731
  0.6423502  -1.6802678   0.1404863  -1.4411132 ]
[-0.05382872 -0.387665   -0.19749434  0.42324948 -0.20067666  0.23863423
  0.19949934 -0.37634277  0.30481192 -0.41476622]
[-0.08562545 -0.28502145 -1.2610829   1.4116249  -1.3126792   0.484129
  1.5019886  -0.6024879   1.8014698  -1.800453  ]
```

WEs in spaCy

spaCy uses Word Vectors behind the scenes

```
import spacy

sentence = "Leonardo da Vinci began painting the Mona Lisa at the age of 51."

# The "small" language models include simple low-dimensional word vectors
nlp_sm = spacy.load("en_core_web_sm") # load the language model
doc = nlp_sm(sentence)
print(doc[0].vector[:12], "...") # word vector of the first token (Leonardo)
print(doc[0].vector.shape) # word vector shape

# The "medium" and "large" models include 300-dimensional GloVe word vectors
# python -m spacy download en_core_web_md
nlp_md = spacy.load("en_core_web_md") # load the language model
doc = nlp_md(sentence)
print(doc[0].vector[:12], "...") # word vector of the first token (Leonardo)
print(doc[0].vector.shape) # word vector shape
```

```
[-0.9564945  -0.23288721  0.11667931  0.92122364 -0.7411676  -0.4035676
 -0.24098495  1.854048   -0.89303887 -0.62927604  0.5505939   1.3656751 ] ...
(96,)
[-2.2694  -1.1013   0.90876 -1.3653  -1.2757  -0.60016  2.99    -0.94047
  2.6915   0.62172 -1.6861  -0.21656] ...
(300,)
```

Document Similarity

Can Word Vectors be used for **document similarity**?

- **Idea:** compute the **average of word vectors** in a document to obtain a **single vector** representing it
- **spaCy** uses this method through the `Doc.similarity` method

Example:

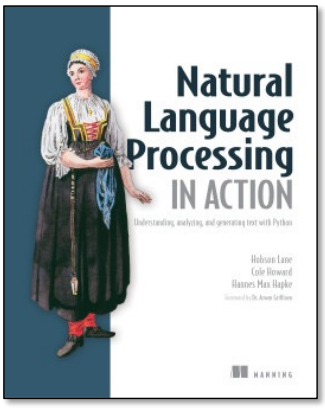
```
doc1 = nlp_md("The CEO addressed the staff about the upcoming \
               changes in the company's strategy.")
doc2 = nlp_md("The executive leader informed the team about the \
               forthcoming adjustments in the firm's approach.")
doc3 = nlp_md("Leonardo was born in Vinci, Italy, in 1452.")

print (doc1.similarity(doc2), doc1.similarity(doc3), doc2.similarity(doc3))
```

```
0.9758518010802842 0.5724102714465288 0.6102447525623739
```

References

Natural Language Processing IN ACTION
Understanding, analyzing, and generating text with Python

Chapter 6



Further Readings…

Gensim documentation

https://radimrehurek.com/gensim/auto_examples/index.html#documentation

Natural Language Processing and Large Language Models

Corso di Laurea Magistrale in Ingegneria Informatica

Lesson 5

Word Embeddings

Nicola Capuano and Antonio Greco

DIEM – University of Salerno