Machine Learning Applications

Lecture 4: Natural Language Processing

Lecture Notes

M.Sc. in Financial Markets

ACADEMIC YEAR 2021-2022

Dr. Dominic O'Kane and Jean-Michel Maeso

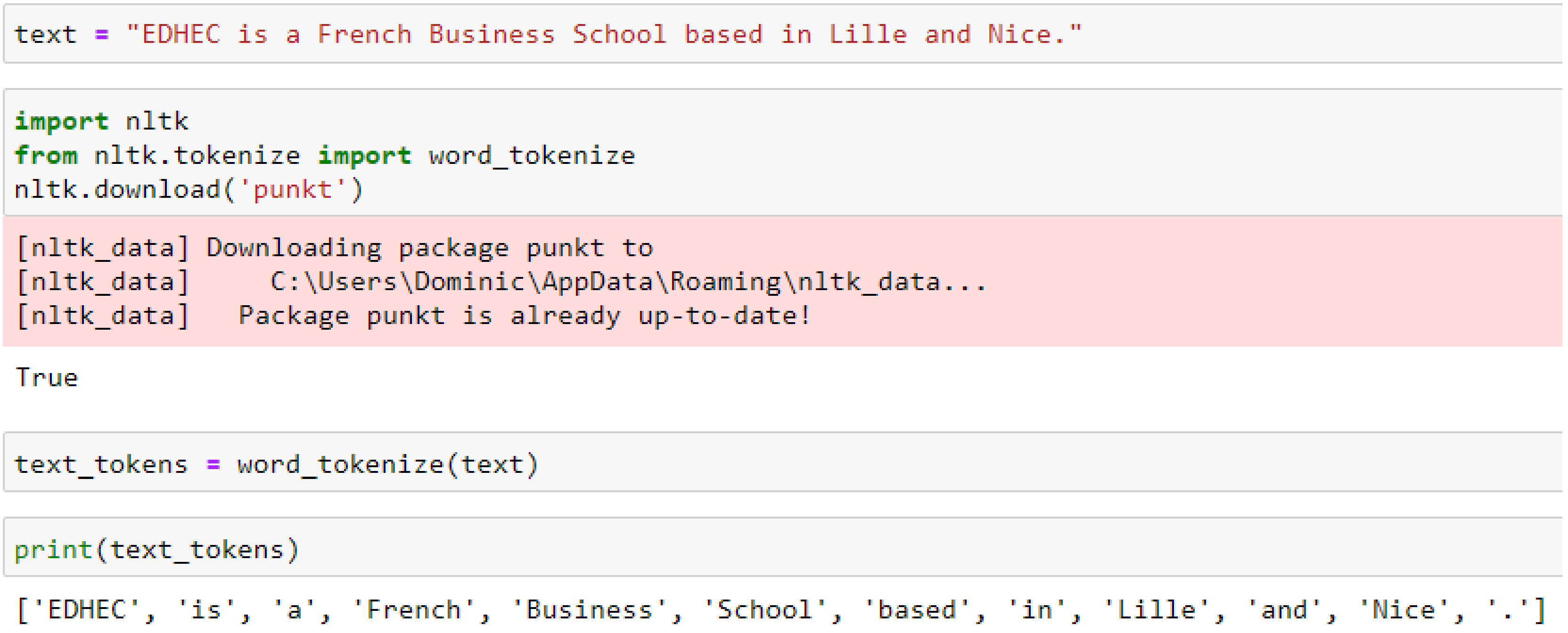
Version: January 2022

NLTK vs spaCy

- There are two very popular ML NLP libraries
- NLTK and SpaCy
- Researchers tend to use NLTK as it is easier to extend
- Practitioners tend to use SpaCy
- SpaCy tries to use the one best algorithm for each NLP task
- NLTK is string based – outputs are strings
- SpaCy is more object oriented
- I will use NLTK

Word Tokenization

- Tokenization is splitting text into segments called tokens
- A good implementation is in the NLTK library – we use the basic method called **word_tokenize** (requires **punkt** to be installed)



```
text = "EDHEC is a French Business School based in Lille and Nice."
```

```
import nltk
from nltk.tokenize import word_tokenize
nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\Dominic\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!

True
```

```
text_tokens = word_tokenize(text)
```

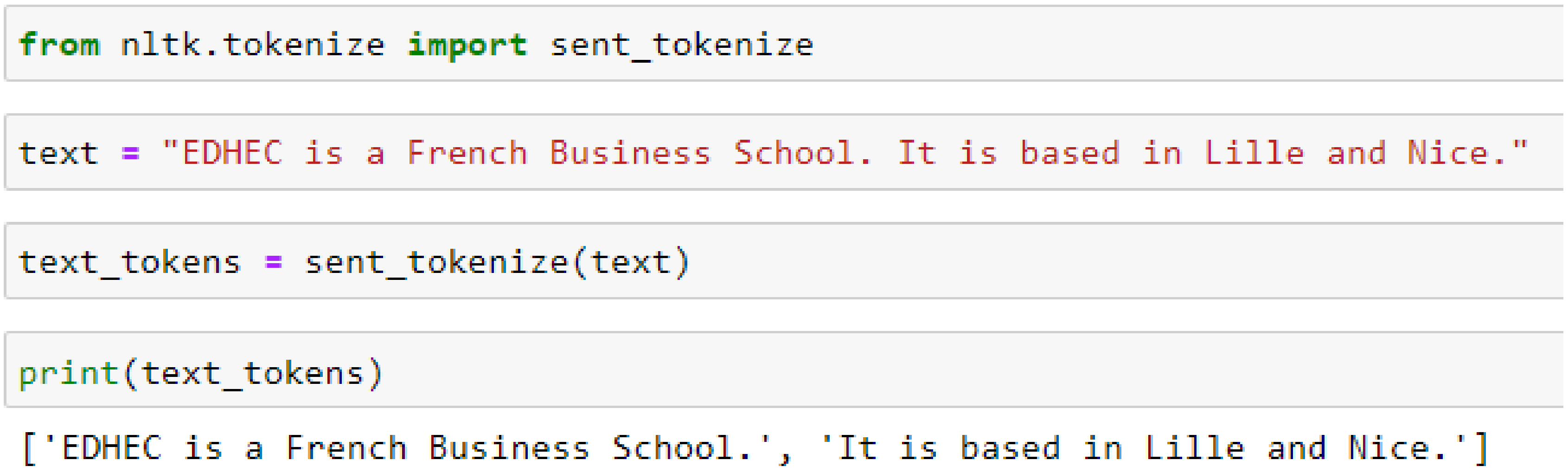
```
print(text_tokens)
```

```
['EDHEC', 'is', 'a', 'French', 'Business', 'School', 'based', 'in', 'Lille', 'and', 'Nice', '.']
```

- Other variations operate at sentence levels

Sentence Tokenization

- We can also tokenize at a sentence level
- We use a different method called **sent_tokenize**



```
from nltk.tokenize import sent_tokenize
```

```
text = "EDHEC is a French Business School. It is based in Lille and Nice."
```

```
text_tokens = sent_tokenize(text)
```

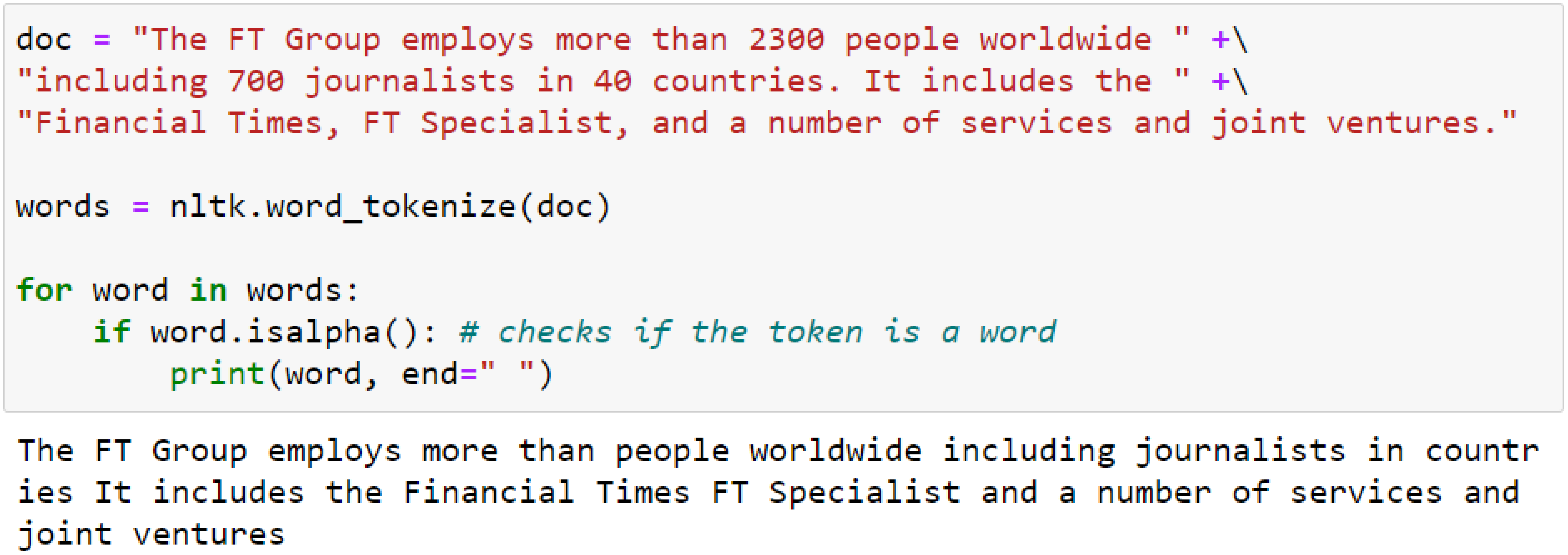
```
print(text_tokens)
```

```
['EDHEC is a French Business School.', 'It is based in Lille and Nice.']
```

- We can then do word tokenization on individual sentences
- We retain information about sentences which can be important for understanding meaning and context

Normalizing

- Normalizing cleans our text of numbers and non-words.



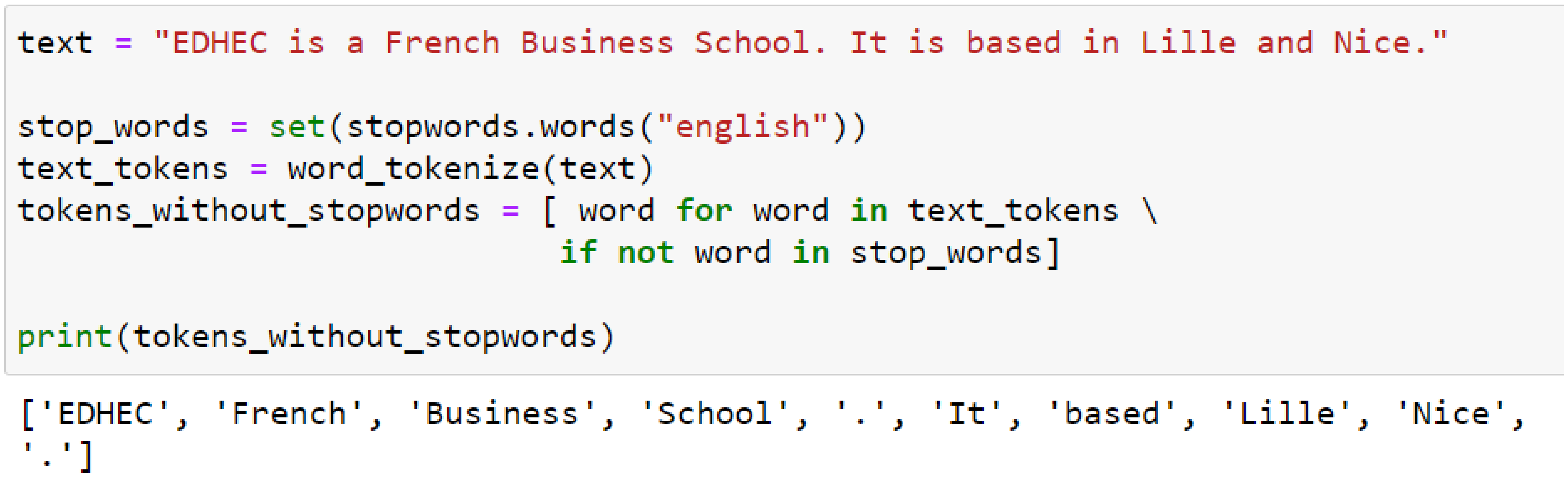
```
doc = "The FT Group employs more than 2300 people worldwide " +\
"including 700 journalists in 40 countries. It includes the " +\
"Financial Times, FT Specialist, and a number of services and joint ventures."

words = nltk.word_tokenize(doc)

for word in words:
    if word.isalpha(): # checks if the token is a word
        print(word, end=" ")
```

```
The FT Group employs more than people worldwide including journalists in countr
ies It includes the Financial Times FT Specialist and a number of services and
joint ventures
```

- It also includes converting text to lower case

Stop Words

- Stop words have little information and should be removed
- If not they can slow down the NLP analysis



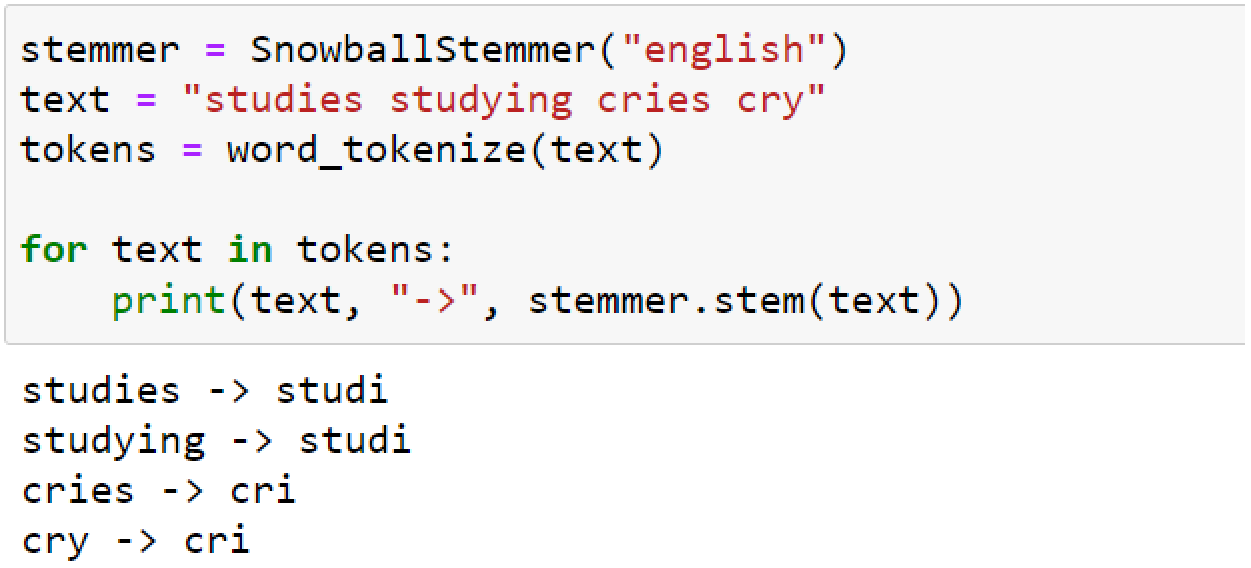
```
text = "EDHEC is a French Business School. It is based in Lille and Nice."

stop_words = set(stopwords.words("english"))
text_tokens = word_tokenize(text)
tokens_without_stopwords = [ word for word in text_tokens \
                            if not word in stop_words]

print(tokens_without_stopwords)
```

```
['EDHEC', 'French', 'Business', 'School', '.', 'It', 'based', 'Lille', 'Nice',
'.']
```

- In this example all it did was remove “is” and “a”
- But it can do more than very simple words
- It can also work for other languages

Stemming

- Some words are simply variations on the same word
- They differ in the suffixes - stemming chops off these suffixes
- NLTK can do this for different languages including French



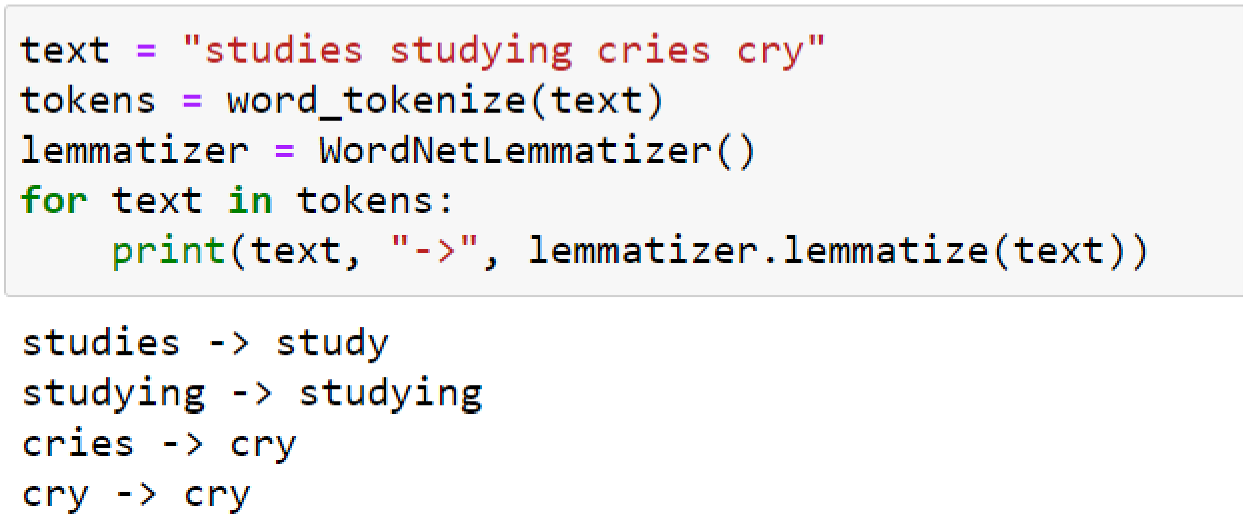
```
stemmer = SnowballStemmer("english")
text = "studies studying cries cry"
tokens = word_tokenize(text)

for text in tokens:
    print(text, "->", stemmer.stem(text))
```

```
studies -> studi
studying -> studi
cries -> cri
cry -> cri
```

- It does not always result in a proper word – “studi” and “cri” !
- But maybe that’s ok as the root may be the same

Lemmatization

- Stemming can create words which do not exist.
- Lemmatization always produces actual words.



```python
text = "studies studying cries cry"
tokens = word_tokenize(text)
lemmatizer = WordNetLemmatizer()
for text in tokens:
    print(text, "->", lemmatizer.lemmatize(text))
```

```
studies -> study
studying -> studying
cries -> cry
cry -> cry
```

- In this example it always finds real words
- However, it did not convert "studying" to "study"
- Requires more computational effort (needs a dictionary)

Parts of Speech

- It may be useful to know the way a word is being used
- NLTK provides a parts of speech (POS) tagging function

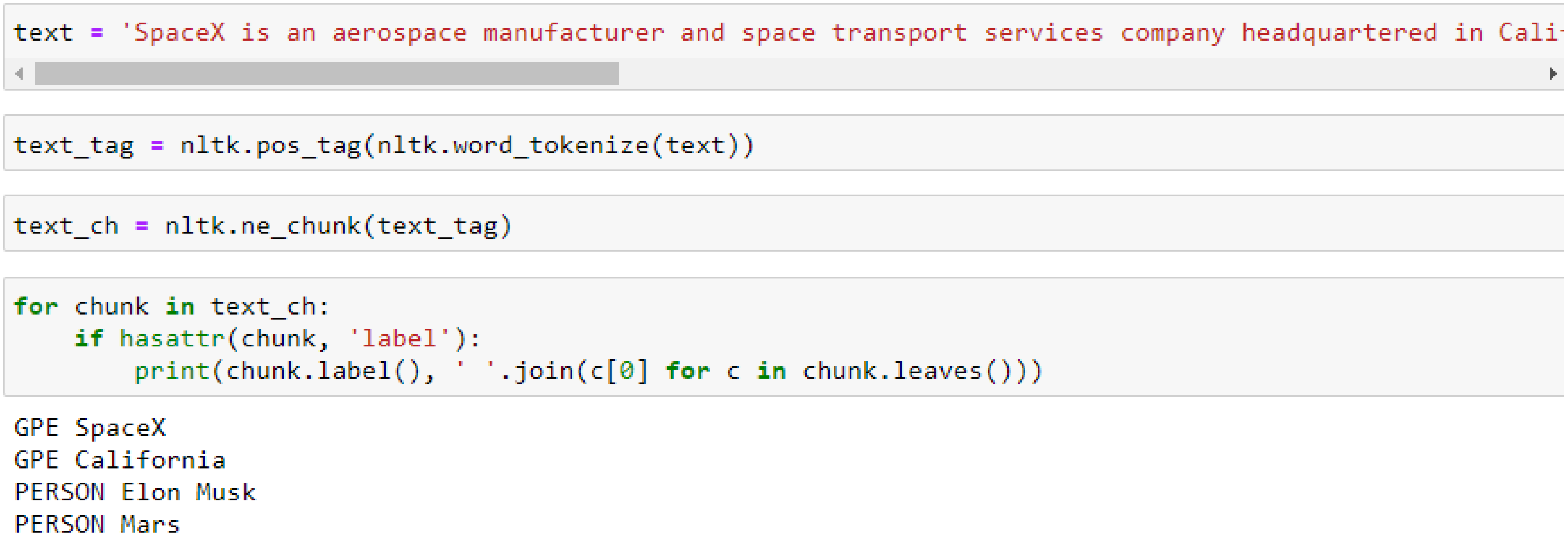
```
text = "EDHEC is a French Business School. It is based in Lille and Nice."
tokens = nltk.word_tokenize(text)
print(nltk.pos_tag(tokens))
```

```
[('EDHEC', 'NNP'), ('is', 'VBZ'), ('a', 'DT'), ('French', 'JJ'), ('Business',
'NNP'), ('School', 'NNP'), ('.', '.'), ('It', 'PRP'), ('is', 'VBZ'), ('based',
'VBN'), ('in', 'IN'), ('Lille', 'NNP'), ('and', 'CC'), ('Nice', 'NNP'), ('.',
'.')]
```

- We can ask NLTK what these tags mean
  - NNP = Noun, proper, singular
  - VBZ = Verb, present tense, third person singular
  - JJ = Adjective or numeral, ordinal
  - CC = Conjunction …

Named Entity Recognition

- Identifying names entities may add useful context
- This is a chunking strategy for dealing with named entities
  - People, Location, Organisations, Geographical, Political Events
- We tag, tokenize and then we chunk the text to find the named entities – in the process it also finds the type of named entity



```
text = 'SpaceX is an aerospace manufacturer and space transport services company headquartered in Cali
```

```
text_tag = nltk.pos_tag(nltk.word_tokenize(text))
```

```
text_ch = nltk.ne_chunk(text_tag)
```

```
for chunk in text_ch:
    if hasattr(chunk, 'label'):
        print(chunk.label(), ' '.join(c[0] for c in chunk.leaves()))
```

```
GPE SpaceX
GPE California
PERSON Elon Musk
PERSON Mars
```

Summary

- Before we begin a corpus analysis, we prepare the documents
- A typical NLP pipeline is as follows
  1. Tokenize
  2. Remove punctuation
  3. Lower case
  4. Remove stop words
  5. Lemmatize words
  6. Removed Named Entities
  7. Tag the parts of speech

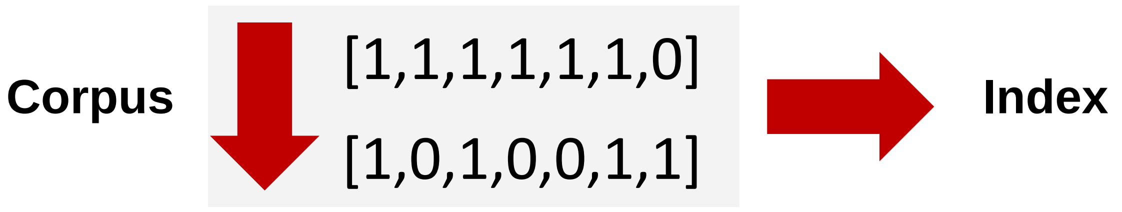
Comparing Documents: Bag of Words

Bag of Words Reminder

- In the last course I introduced Bag of Words for NLP
- Bag of Words (BoW) models have no context information
- Remember we make a list of all the unique words in the corpus
- We make a vector with a length equal to of all unique words
- Each document is a vector of word counts for each unique word
- Each word is a vector with a 1 at the corresponding element
- Bag of Words is an OK approach for identifying a few specific words with the same meaning – e.g., SPAM detection

Example of the Bag of Words Approach (Reminder)

- Corpus of two documents
  - Document 1 - "The car is red and green"
  - Document 2 - "The tree is green"
- The **vocabulary** is as follows where we index each word
  - 0 - the, 1 - car, 2 - is, 3 - red, 4- and 5 - green, 6 - tree
- Each word is a feature, and each document is represented as a vector of **frequency of occurrence of the word in the document**
- In this case, the two documents become the **bag of words**



[1,1,1,1,1,1,0]

[1,0,1,0,0,1,1]

- In this corpus no words occur more than once in each document
- We can see that they match at 'the' and 'is' and 'green'

Word Counts with CountVectorizer

- A search of all unique words across a corpus, plus a word count on each and building count vectors is **computationally expensive**
- Imagine you have 100 documents, each with 10,000 words
- That means you have 1M words (many of them many times)
- You have to do a unique search over all of the words
- You then have to count the number of each of these words in each document
- Scikit learn has a number of functions that do the work for us
- This is faster than if we write it ourselves
- Tensor Flow also has functions to help us but we will use the ones in Scikit

Word Counts with CountVectorizer

- We use the **fit** method to pass in all of the documents in the corpus and this generates the entire **vocabulary** of words

```
from sklearn.feature_extraction.text import CountVectorizer
count = CountVectorizer(min_df=1)
count.fit(X_train['text'].values)
```

- We then use the **transform** method to convert each document into a bag of words matrix that can be used for machine learning

```
bag = count.transform(X_train['text'].values)
```

- The min_df is the minimum number of times a word must occur to be included in the vocabulary
- If min_df is 10 and the word only occurs 8 times, it is not included in the vocabulary – saves time and memory

Example: Building a Bag of Words (Reminder)

```
docs = np.array(['EDHEC is the best business school in the world',
'Machine learning is cool', 'EDHEC teaches Machine Learning’])
# Find the vocabulary in the corpus
count.fit (docs)
# Print the list of words and their corresponding index
print(count.vocabulary_)
{'edhec': 3, 'is': 5, 'the': 10, 'best': 0, 'business': 1, 'school': 8, 'in':
4, 'world': 11, 'machine': 7, 'learning': 6, 'cool': 2, 'teaches': 9}
# Generate bag of words vectors for each document
bag = count.transform(docs)
print(bag.toarray())
[[1 1 0 1 1 1 0 0 1 0 2 1]
[0 0 1 0 0 1 1 1 0 0 0 0]
[0 0 0 1 0 0 1 1 0 1 0 0]]
```

Note that the ordering of the actual words in the sentences has been lost

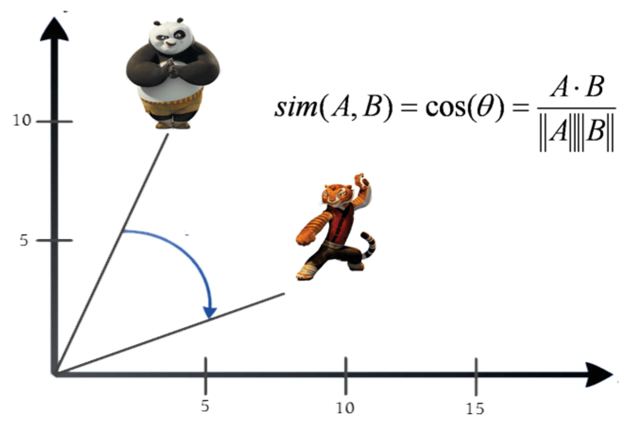
Cosine Similarity

- If we calculate the dot product of two document Bag of Words vectors a and b we get a number
- Geometrically this number can be seen as a cosine similarity

$$A.B = |a||b| \cos \theta$$

- The similarity is given by $sim(A,B) = cos\theta = \frac{A.B}{|A||B|}$

Cosine Similarity



- If sim(A,B) is 1 then the documents are essentially identical
- If sim(A,B) is 0 then the documents have nothing in common

Cosine Similarity Calculation (BoW representation)

- Assume that our objective is to calculate the cosine similarity of the two documents below:
  - d1 = 'the best deep learning course'
  - d2 = 'deep learning is easy'

	best	course	deep	easy	is	learning	the
Doc-1	1	1	1	0	0	1	1
Doc-2	0	0	1	1	1	1	0

- The documents can be represented (BoW) by the following vectors:

  $d_1 = [1,1,1,0,0,1,1]$ and $d_2 = [0,0,1,1,1,1,0]$

- $d_1 . d_2 = 2;\ \ |\mathrm{d}_1| = \sqrt{5};\ \ |d_2| = 2$

- $sim(d_1, d_2) = \frac{2}{2\sqrt{5}} = \frac{1}{\sqrt{5}}$

Application Exercise

- Implement the previous example in a Jupyter notebook using the scikit-learn library

TF-IDF

Purpose of TF-IDF

- Suppose we have a corpus of documents, and we want to find documents which are most relevant to some search term t
- A search engine like Google uses TF-IDF scores when it ranks search results
- If t occurs once in all the documents, then it is not very exclusive - we should return all documents but they each have low relevant information content
- If t occurs 100 times in 10 documents and once in the other documents, then we should return the 10 documents - each of those documents has high information content
- We need a way to decide which documents are the most relevant

Calculating the TF-IDF

- Mathematically we write

$$TF - IDF(d,t) = TF(d,t) \times IDF(t)$$
$$= TF(d,t) \times \left[1 + \log\left(\frac{1+N}{1+DF(t)}\right)\right]$$

- N is the number of documents in the corpus
- TF(d,t) counts the number of times term t is in document d
- DF(t) counts the number of documents in the corpus which contain term t

Cosine similarity calculation (TF-IDF representation)

- Our objective is to calculate for each word $TF - IDF(d,t)$ that appears in the two documents below:
  - d1 = 'the best deep learning course'
  - d2 = 'deep learning is easy'

		best	course	deep	easy	is	learning	the
Term frequency TF(d1,t)	**d1**	1	1	1	0	0	1	1
Term frequency TF(d2,t)	**d2**	0	0	1	1	1	1	0

		best	course	deep	easy	is	learning	the
Inverse document frequency		1.41	1.41	1.00	1.41	1.41	1.00	1.41
$IDF(t) = 1 + \ln\left(\frac{1+N}{1+DF(t)}\right)$								

		best	course	deep	easy	is	learning	the
TF-IDF(d1,t) = TF(d1,t)*IDF(t)	**d1**	1.4055	1.4055	1.0000	0.0000	0.0000	1.0000	1.4055
TF-IDF(d2,t) = TF(d2,t)*IDF(t)	**d2**	0.0000	0.0000	1.0000	1.4055	1.4055	1.0000	0.0000

- $d_1 . d_2 = 2;\ \ |\mathrm{d}_1| = 2.815;\ \ |d_2| = 2.439$
- $sim(d_1, d_2) = 0.2912$

Application Exercise

- Implement the previous example in a Jupyter notebook using the scikit-learn library

BoW and TF-IDF comparison

		best	course	deep	easy	is	learning	the
Term frequency TF(d1,t)	**d1**	1	1	1	0	0	1	1
Term frequency TF(d2,t)	**d2**	0	0	1	1	1	1	0

		best	course	deep	easy	is	learning	the
Inverse document frequency		1.41	1.41	1.00	1.41	1.41	1.00	1.41

$$IDF(t) = 1 + \ln\left(\frac{1+N}{1+DF(t)}\right)$$

		best	course	deep	easy	is	learning	the
TF-IDF(d1,t) = TF(d1,t)*IDF(t)	**d1**	1.4055	1.4055	1.0000	0.0000	0.0000	1.0000	1.4055
TF-IDF(d2,t) = TF(d2,t)*IDF(t)	**d2**	0.0000	0.0000	1.0000	1.4055	1.4055	1.0000	0.0000

- **BoW**: it creates a set of vectors containing the count of word occurrences in the document
- **TF-IDF**: unlike BoW, it gives larger values for less frequent words. TF(t,d) is high when the word t is frequent in document d but rare in all the corpus of documents
- Bag of Words and TF-IDF know about content, but not about context, i.e., they do not know the ordering of the words
- How can we capture the context?

Advanced NLP: Word Embeddings

Bag of Words to Word Embeddings

- Bag of Words has problems for more complex NLP tasks:
  - Word order is ignored
  - Words are treated separately from each other
  - All context is lost
- For example:
  - "There was a good ambience at the football match"
  - "There was a good atmosphere at the football match"
- The word atmosphere and ambience are used in the same context, which suggests that they have a similar meaning
- How can we train a ML model to recognize this sort of semantic similarity ?

Word Vectors

- We allow each word to have a word-vector
- This means that we represent a word by a high dimensional numerical array or real numbers, e.g.,

  "atmosphere" = [0.34, 7.48, 0.93, 2.83, …, 5.28, 19.22, 2.32]

- With a numerical representation we can do machine learning !
- The hope is that we can find a word representation such that two words that have a similar meaning will have a similar word-vector
- By similar word vector, I mean that we want similar words "atmosphere" and "ambience" to have a high cosine similarity
- How can we do this in practice ? Can we teach this to a NN ?

Word Embeddings

- The idea is old – stated in 1957 by the British linguist J.R. Firth:

  "You shall know a word by the company it keeps"

- Context is a guide to understanding meaning – context is about what words appear together and how close together

  1. *I laughed at his joke*
  2. *His joke made me laugh*

- The word "joke" associates with laugh and laughed – this will associate laugh and laughed
- A word embedding is a word vector that knows about context
- Words "laugh" and "laughed" should have a similar embedding

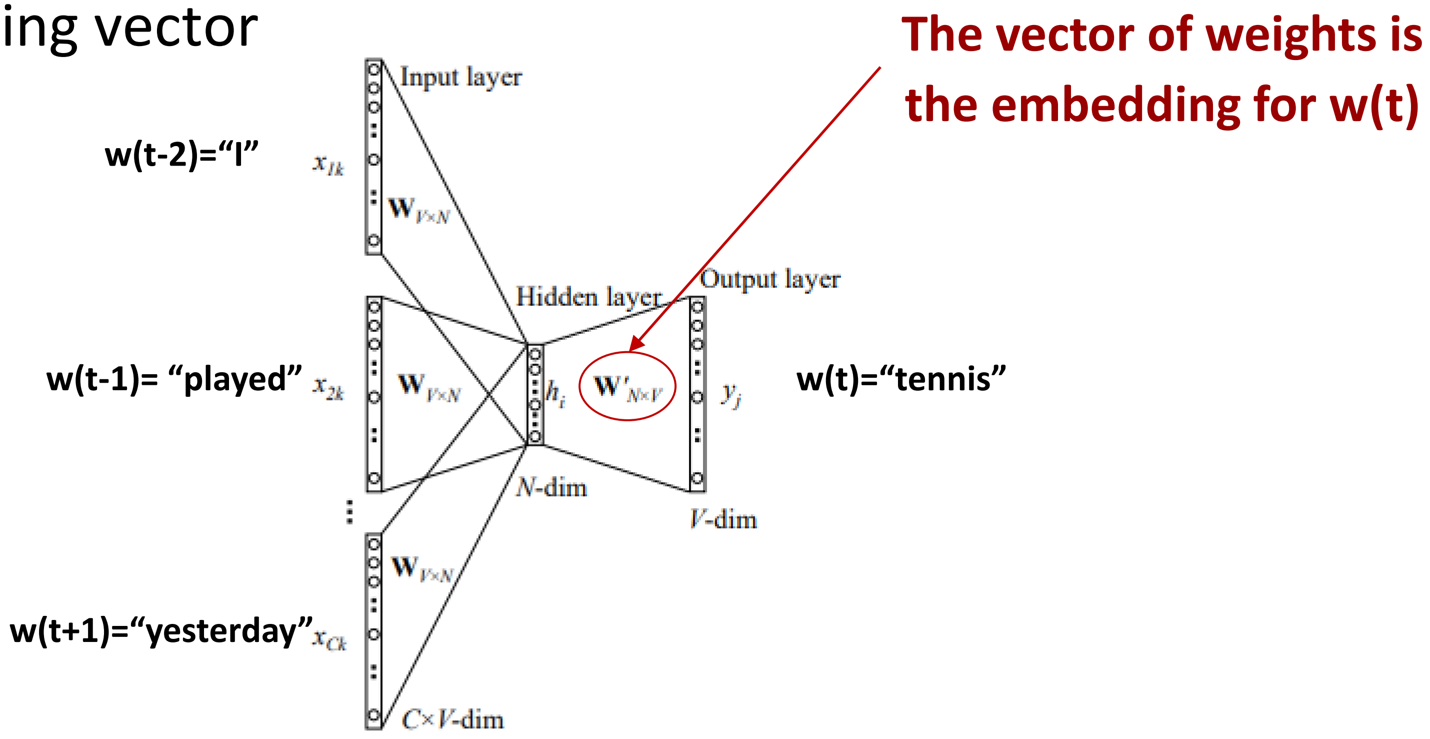
Word2Vec is a ML Algorithm for Word Embeddings

- Word2Vec is a Word Embedding algorithm
- It designed by Thomas Mikolov at Google in 2013
- This algorithm learns word associations from a corpus of words
- Each word is represented by a word vector of numbers
- **cosine similarity** indicates the semantic similarity between words
- The architecture to do this it uses a 2-layer neural network
  - Input layer
  - Projection layer
  - Output layer
- There are two different approaches in Word2Vec
  - Continuous Bag of Words (CBOW)
  - Skip Gram

Word2Vec: Continuous Bag Of Words

- CBOW predicts current word based on context words within a given window
- Inputs = Context Words – words in sentence around current word and Output = Current Word
- The hidden layer has the number of dimensions we want for the embedding vector

<u>Rationale:</u> Try to predict w(t) given w(t-2), w(t-1) and w(t+1)

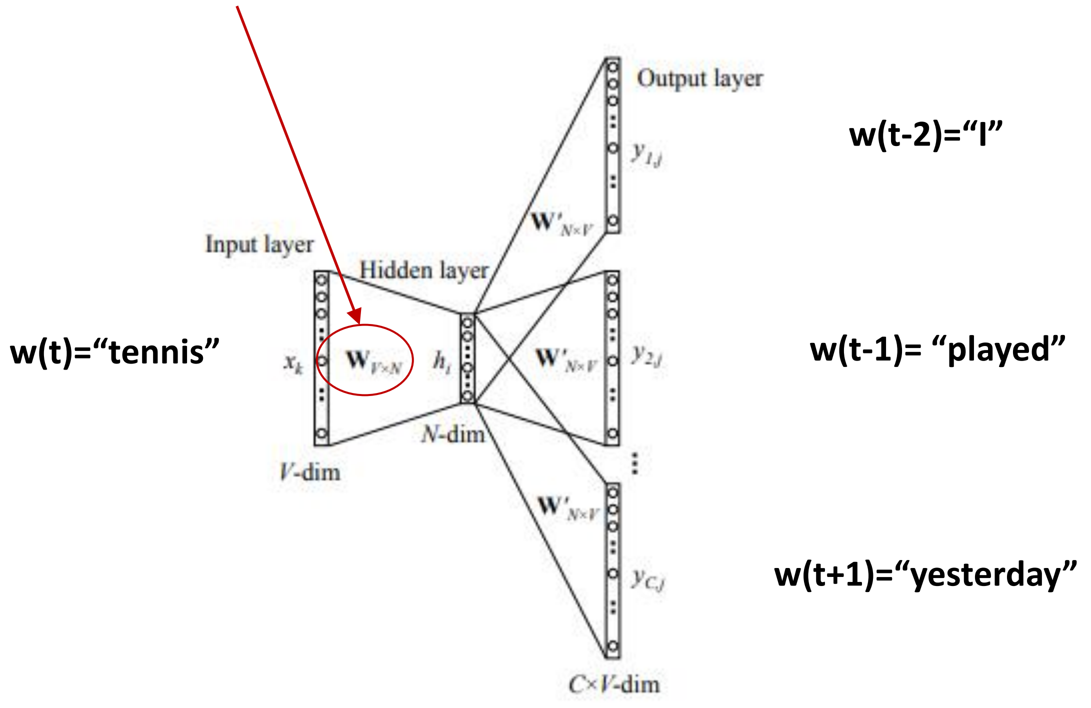


Word2Vec: Skip Gram

- Predicts the surrounding words given current word
- The hidden layer has the number of dimensions to be used to represent the word in the embedding

The vector of weights is the embedding for w(t)

Rationale: Try to predict w(t-2), w(t-1) and w(t+1) given w(t)



How Skip-Gram Works

We will only consider Skip-Gram's algorithm – we do not have the time to do a full description - this is a high-level description

1. We loop over all the text in our corpus
2. Take a word w(t), call it a centre word, and set a radius m
3. Word w(t-m) to w(t+m) are words in the radius around w(t)
4. Output the conditional probability of a context word p(w(t-k)| w(t)) – a context word is a word within the window around w(t)
5. It has a Softmax output layer, a cross entropy cost function that we wish to train the neural network to minimise
6. Parameters are vector representations of the words
7. Main hyperparameters are the length of each word vector and the radius of the window

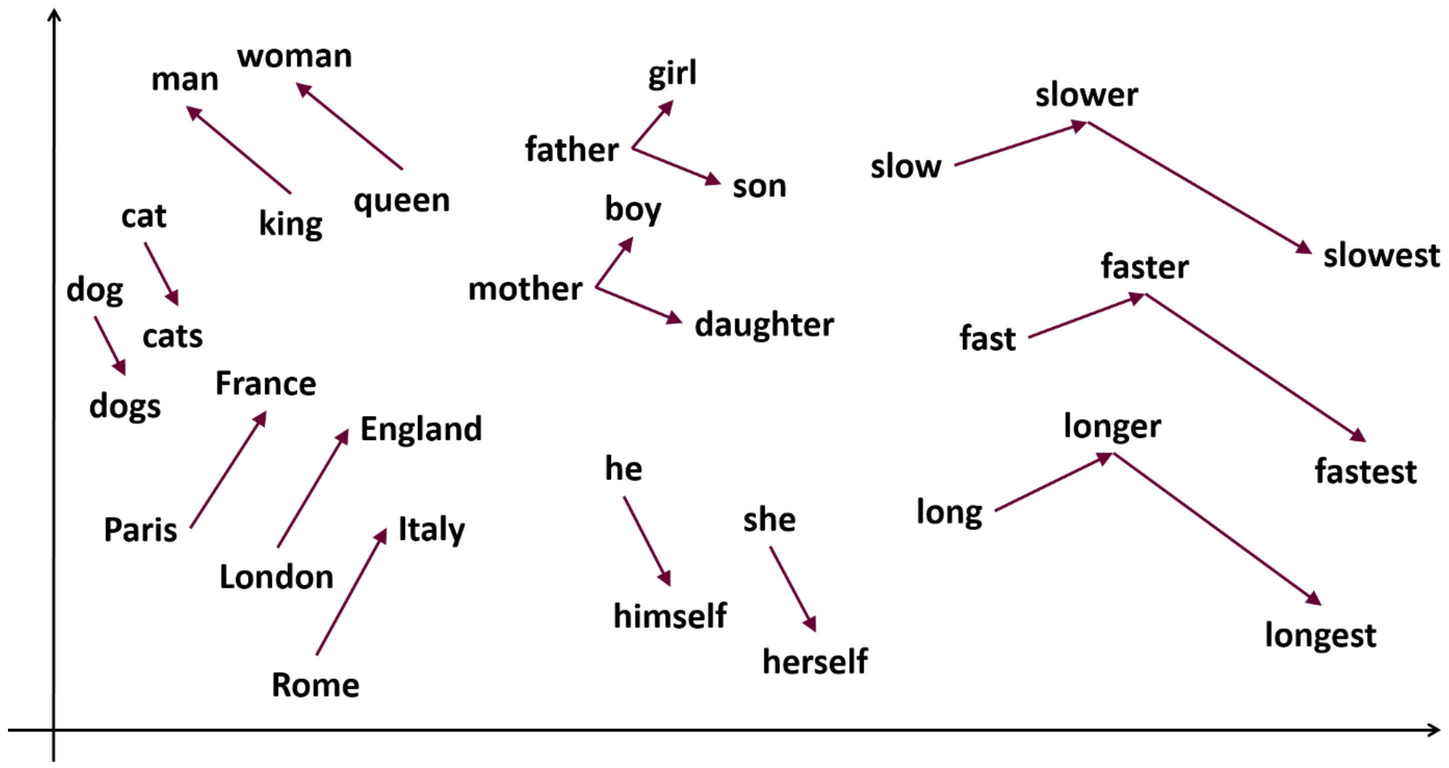
How Skip-Gram Works

- The corpus size and model parameters greatly affect the performance of a word2vec model
- In large corpora, Skip-Gram appears to work best – highest accuracy – but it is more computationally expensive than CBOW
- If you can predict many words from one word, you are doing well

Word2Vec can find Semantic Relationships ?

- Word2vec can learn about semantic differences in word meaning
- Remarkably we find that word embeddings can do things like:

Queen = King + (Woman – Man)



Word Embeddings - Disadvantages

- There is just one vector representation for each word (a word can have different meanings depending on the context)
- Depend a lot on the corpus used to train them
- You need to use the same tokenizer
- You need the same number of dimensions
- You may find a word that was not in the training set

Advanced NLP: Transformers

Word2Vec versus Transformer

- Word2Vec models: **context-independent embeddings**. There is just one vector representation for each word. Different senses of the word (if any) are combined into one single vector.

- A Transformer model generates **context-dependent** embeddings that allows to have multiple vector (numeric) representations for the same word, based on the context in which the word is used.

- Example: “$\overbrace{\text{We went to the river}}^{\text{Context}}$ bank” and “$\overbrace{\text{I made a deposit to}}^{\text{Context}}$ the bank”

- Word2Vec embeddings do not take into account the word position while a Transformer model explicitly takes as input the position (index) of each word in the sentence before calculating its embedding.

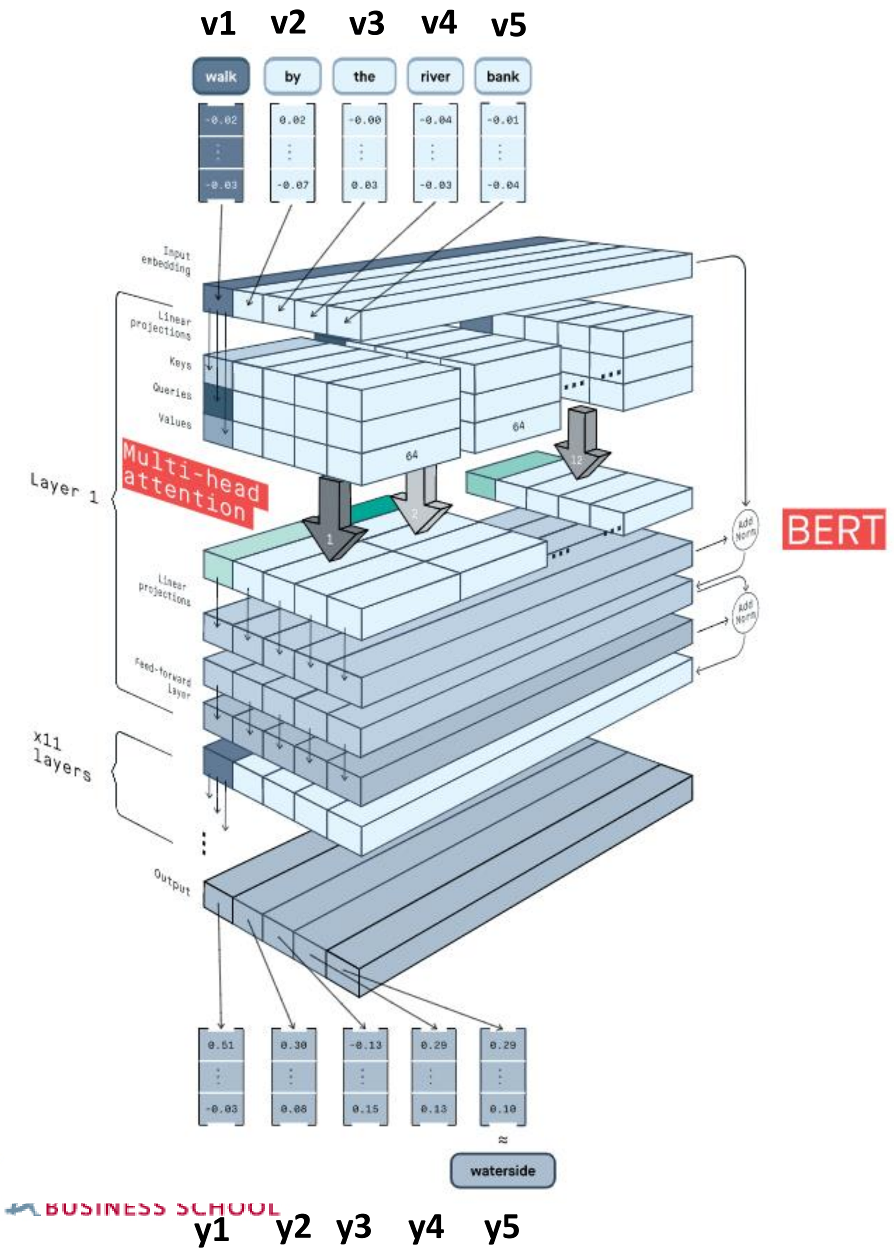
Transformer

- A **transformer** is a deep learning model that adopts the mechanism of **self-attention**, differentially weighting the significance of each part of the input data

- Like RNNs, transformers are designed to handle sequential input data, such as natural language, for tasks such as translation or text summarization. However, unlike RNNs, transformers do not necessarily process the data in order.

- Transformers were introduced in 2017 by a team at Google Brain and led to the development of pretrained systems such as BERT (Bidirectional Encoder Representations from Transformers)

BERT Transformer and Self-Attention Mechanism



All the words are encoded in vector embedding v_i

All the words are encoded in contextualized embedding vectors that take into account the context y_i

Transformer and Self-Attention Mechanism

- Two have more insight about the self-attention mechanism and the key, query and value projections, watch the two 10 minutes videos:
  - https://peltarion.com/blog/data-science/self-attention-video
  - https://www.youtube.com/watch?v=-9vVhYEXeyQ

Case Study