[Tutorial] Word Embedding

1. Principles

1.1. Limitations of classical NLP features

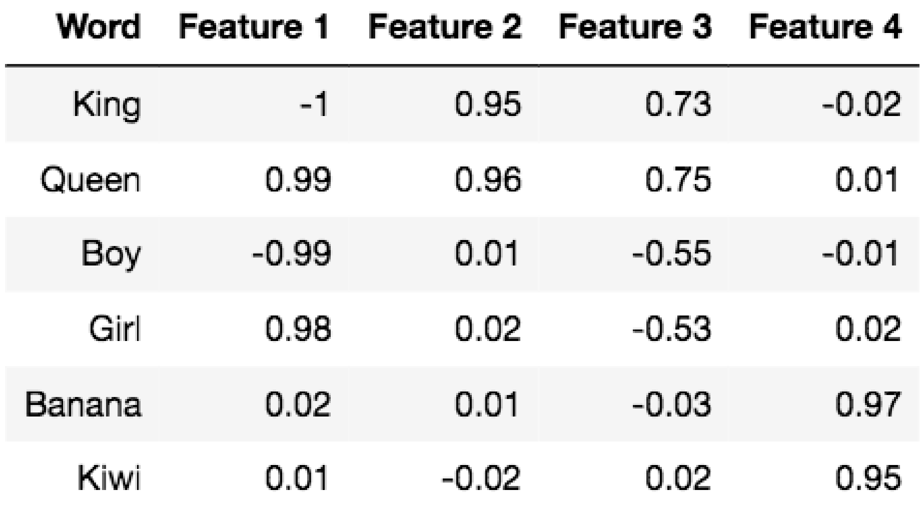
Basic NLP BOW or TF-IDF do not have relationship between words: we would like for example **to have a relationship between words** like man/woman, banana/apple, car/truck or even lunch/dinner, right?

Word embedding allows to have such a relationship between words, while keeping **a quite small input features size**.

1.2. What is a Word Embedding ?

A Word Embedding is a matrix representation, meaning **each word has a given number of features**. You can see the words as vectors and the features as their coordinates.

Below is an example of Word Embedding WW of 6 words and 4 features :



Word	Feature 1	Feature 2	Feature 3	Feature 4
King	-1	0.95	0.73	-0.02
Queen	0.99	0.96	0.75	0.01
Boy	-0.99	0.01	-0.55	-0.01
Girl	0.98	0.02	-0.53	0.02
Banana	0.02	0.01	-0.03	0.97
Kiwi	0.01	-0.02	0.02	0.95

We can see some properties from that simple example:

- $W_{King} - W_{Queen} \simeq W_{Boy} - W_{Girl}$WKing−WQueen≃WBoy−WGirl
- $W_{Banana} \simeq W_{Kiwi}$WBanana≃WKiwi

Those properties bring a whole new dimension to NLP: words are not just 0 or 1 anymore like in TF-IDF or BOW. Words have relationships between them:

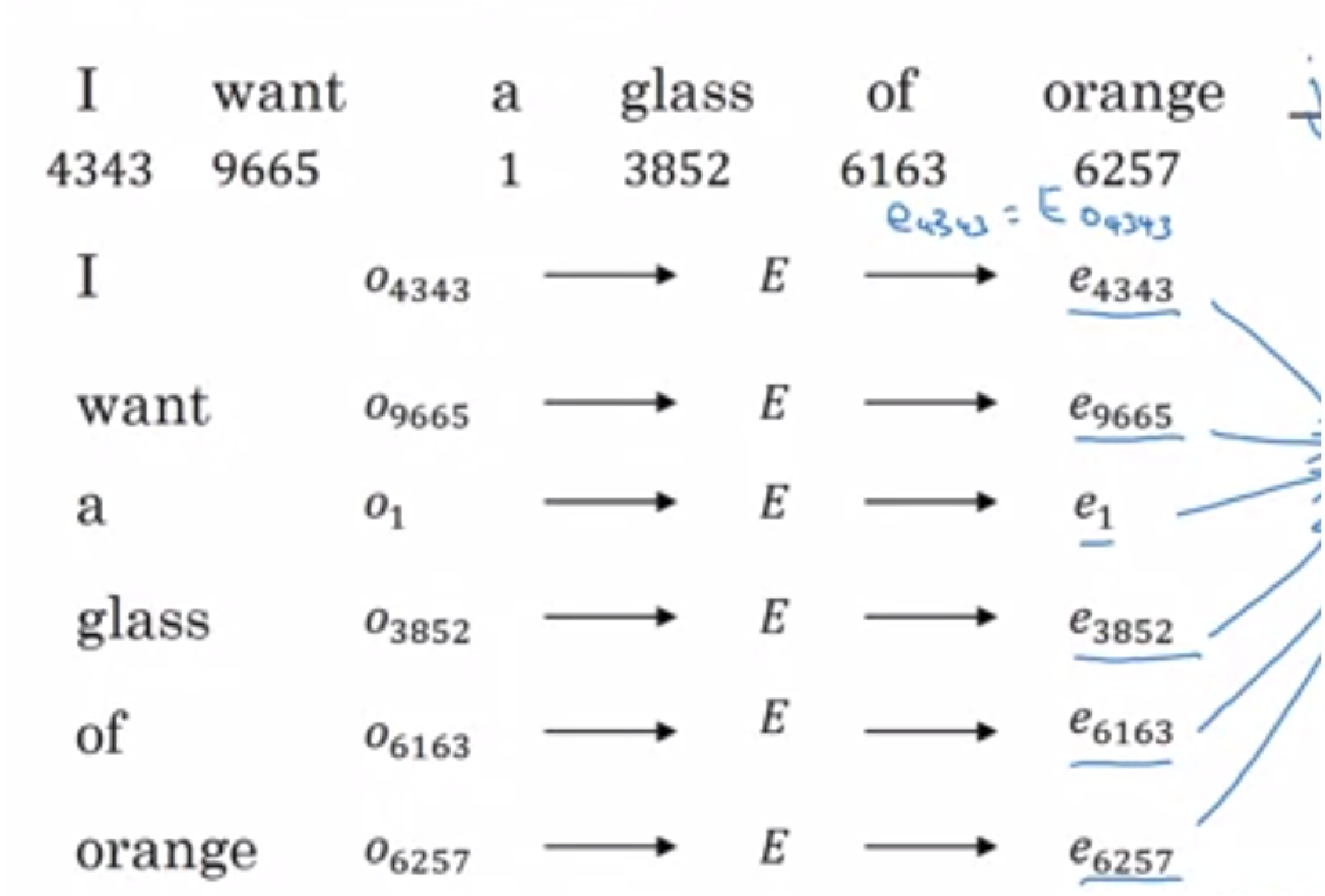
- The difference between a King and a Queen is basically the gender: the same goes between a Boy and a Girl
- A Banana and a Kiwi are both fruits: they are not so different

Even more information is encoded in that table:

- Feature 2 is highly correlated to King and Queen, and not to any other meaning : it can be a feature for Royalty
- Feature 3 is correlated to King and Queen positively, to Boy and Girl negatively: it can be a feature for the Age

1.3. How is a Word Embedding built ?

The main idea to build a Word Embedding is to **try to predict a next word** into a sentence or a sequence of words:



I want a glass of orange
4343 9665 1 3852 6163 6257
$e_{4343} = E\,o_{4343}$
I o_{4343} → E → e_{4343}
want o_{9665} → E → e_{9665}
a o_1 → E → e_1
glass o_{3852} → E → e_{3852}
of o_{6163} → E → e_{6163}
orange o_{6257} → E → e_{6257}

As you can see on the picture, it is a **multiclass classification** problem: you have the first words of a sentence `I want a glass of orange` as input, and you want to predict the next word: `juice`.

But before, instead of giving a Bag Of Words or a TF-IDF as input, you pass the words into a matrix `E` that will be learnt with the model. This matrix `E` **transforms each word into a new vector**, so each word is transformed into a new set of features.

More specifically, there are **two main models** to learn this embedding matrix `E`:

- Skip-Gram: the idea is to **predict the word giving just one random word of context** (e.g. predict `juice` with only `glass` as input)

- Continuous Bag Of Words: the idea is to **predict the context giving one random word** (e.g. predict `I want a glass of orange` with `juice` as input)

---

2. Most Used Word Embeddings

There are many different Word Embeddings. We will only mention **the three most relevant and frequently used**.

2.1. GloVe

GloVe (for Global Vector) word embedding was made by Stanford NLP group. This is a **relatively light** word embedding, but still very powerful. Several models exist, to encode words into vectors of 50, 100, 200 or even 300 dimensions.

More information about this word embedding can be found **here (https://nlp.stanford.edu/projects/glove/)** , as well as pretrained models, which can be really helpful.

We will show in the next section how to use it and what it does.

2.2. Word2Vec

Word2Vec is probably **the most used** Word Embedding currently. It is really **powerful and versatile**. It has been proven to get relationships between words that were unexpected.

Most of the time, this word embedding is **used with a pretrained model by Google**. More information about Word2Vec can be found **here (https://code.google.com/archive/p/word2vec/)** and the pretrained model can be found **here (https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=sharing)** .

This model is **particularly complex**, thus the pretrained model is saved in a **very large file** (about 4 GB). So if your computer does not have enough RAM, you won't be able to use it, that why we will show you how to use GloVe only, which is lighter.

However, for your information, below is a code snippet that would allow you to load and use a Word2Vec model using Gensim:

```
import gensim

# Load Google's pre-trained Word2Vec model, give the right path to the downloaded file
model = gensim.models.KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bin', binary
=True)

# Then you can get the embedding of a word as a numpy array this way
word_embedding = model.wv['hello']
```

2.3. FastText

FastText is developped by Facebook, and is probably **the most efficient** Word Embedding. It does not only uses words, but also **all possible n-grams** (at character level) of any word to compute its embedding.

Thus, FastText is sometimes superior to other word embeddings, but is **far more complex and heavy to use**.

You can find more information on how to use FastText on the dedicated webpage: [https://fasttext.cc/ (https://fasttext.cc/)](https://fasttext.cc/) .

---

3. Usage and Visualization

3.1. Load and use a Word Embedding

We will now see how to use a Word Embedding, with GloVe as an example. Even though the code might be different to load another Word Embedding, the usage would remain the same.

👉 First, we have to **define a function to read a pretrained model**.

```
import numpy as np
import pandas as pd

# Function that allows to read a pretrained model and returns words and a dictionary of word embeddi
ngs
def read_glove_vecs(glove_file):
    with open(glove_file, 'r') as f:
        words = []
        word_to_vec_map = {}

        for line in f:
            line = line.strip().split()
            curr_word = line[0]
            words.append(curr_word)
            word_to_vec_map[curr_word] = np.array(line[1:], dtype=np.float64)

    return words, word_to_vec_map
```

👉 Then we use this function to load the pretrained model provided by the Stanford NLP group.

```
words, word_to_vec_map = read_glove_vecs('../../../../../glove.6B.50d.txt')
len(words)
```

```
400000
```

```
pd.DataFrame(word_to_vec_map).head()
```

```
print('number of words:', len(words))
print('5 words examples:', words[1000:1010])

print('dimension of word embedding: (', len(word_to_vec_map),',',len(word_to_vec_map['hello']), ')')
print('example of word embedding:', word_to_vec_map['paris'])
```

```
number of words: 400000
5 words examples: ['themselves', 'firm', 'injured', 'itself', 'governor', 'movie', 'range', 'cross',
'track', 'programs']
dimension of word embedding: ( 400000 , 50 )
example of word embedding: [ 0.76989   1.181    -1.1299   -0.74725  -0.5969   -1.0518   -0.46552
  0.27009  -0.99243  -0.04864   0.28642  -0.75261  -1.0566   -0.19205
  0.572    -0.24391  -0.36054  -0.70876  -0.91951  -0.27024   1.5131
  1.0313   -0.55713   0.52952  -0.71494  -1.0949   -0.60565   0.31329
 -0.44488   0.55915   2.1429    0.43389  -0.5529   -0.24261  -0.43679
 -0.96014   0.25828   0.79385   0.37132   0.49623   0.84359  -0.25875
  1.5616   -1.1199    0.091676  0.076675 -0.45084  -0.86104   0.97599
 -0.35615 ]
```

To summarize, in this word embedding, we have:

- 400000 words
- For each word, an embedding into 50 dimensions

Indeed, we loaded the file `glove.6B.50d.txt`, `50d` standing for 50 dimensions. But GloVe comes also with files of 100, 200 and 300 dimensions if more information is needed.

👉 We can now compute some interesting properties, like the similarities between words:

```
from sklearn.metrics.pairwise import cosine_similarity

father = word_to_vec_map["father"].reshape(1, -1)
mother = word_to_vec_map["mother"].reshape(1, -1)
ball = word_to_vec_map["ball"].reshape(1, -1)
president = word_to_vec_map["president"].reshape(1, -1)
queen = word_to_vec_map["queen"].reshape(1, -1)
king = word_to_vec_map["king"].reshape(1, -1)
girl = word_to_vec_map["girl"].reshape(1, -1)
boy = word_to_vec_map["boy"].reshape(1, -1)

print("cosine_similarity(father, mother) = ", cosine_similarity(father, mother))
print("cosine_similarity(ball, crocodile) = ", cosine_similarity(ball, president))
print("cosine_similarity(queen - king, girl - boy) = ", cosine_similarity(queen - king, girl - boy))
```

```
cosine_similarity(father, mother) =  [[0.89090384]]
cosine_similarity(ball, crocodile) =  [[0.20942503]]
cosine_similarity(queen - king, girl - boy) =  [[0.63886288]]
```

👉 We could also find analogies between words using the same idea (ex: Paris is to France what Tokyo is to ?). You can try to build the function as an exercice!

3.2. Visualization

One of the sexiest thing about word embeddings is the fact that you can make **great visualizations** showing the relationships between words.

Let's choose some words, and then apply a t-SNE on their embeddings and display their relationships.

```
# We choose a bunch of words we want to visualize
words_to_viz = ['president', 'obama', 'trump', 'minister', 'power', 'goverment', 'law',
                'apple', 'banana', 'pasta', 'pizza', 'burger', 'food', 'snack', 'meal', 'dinner',
                'france', 'paris', 'japan', 'tokyo', 'rome', 'italy', 'country', 'city',
                'man', 'woman', 'girl', 'boy', 'daughter', 'son', 'father', 'mother',
                'school', 'college', 'university', 'homework', 'student', 'scholarship',
                'company', 'work', 'employee', 'boss', 'hire', 'salary', 'startup']

# We compute the word embedding of those words
embed_to_viz = []
for w in words_to_viz:
    embed_to_viz.append(word_to_vec_map[w])
```

```
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

# We compute the t-SNE of the word embeddings
tsne = TSNE(n_components=2, perplexity=10)
viz = tsne.fit_transform(embed_to_viz)

# We plot the results of the t-SNE
plt.figure(figsize=(15,15))
for i, w in enumerate(words_to_viz):
    plt.annotate(s=w, xy=(viz[i,0], viz[i, 1]), xytext=(viz[i,0]+0.5, viz[i, 1]+0.5))
plt.scatter(viz[:,0], viz[:,1])
plt.plot()
```

```
[]
```

As you can see, some relationships between words appear on the t-SNE visualization. This is one more reason why word embeddings got so popular: they can be understood easily!

---

4. Bias in Word Embeddings

⚠️ Word Embeddings area very powerful and very popular tool for NLP, but they are not devoid of **biases**.

➡️ **Word embeddings are only as neutral as the input data on which they were trained** : since they are largely trained on corpuses from the Internet (Wikipedia, notably), and those corpuses are full of biases, those biases will be reflected in the output of the embeddings.

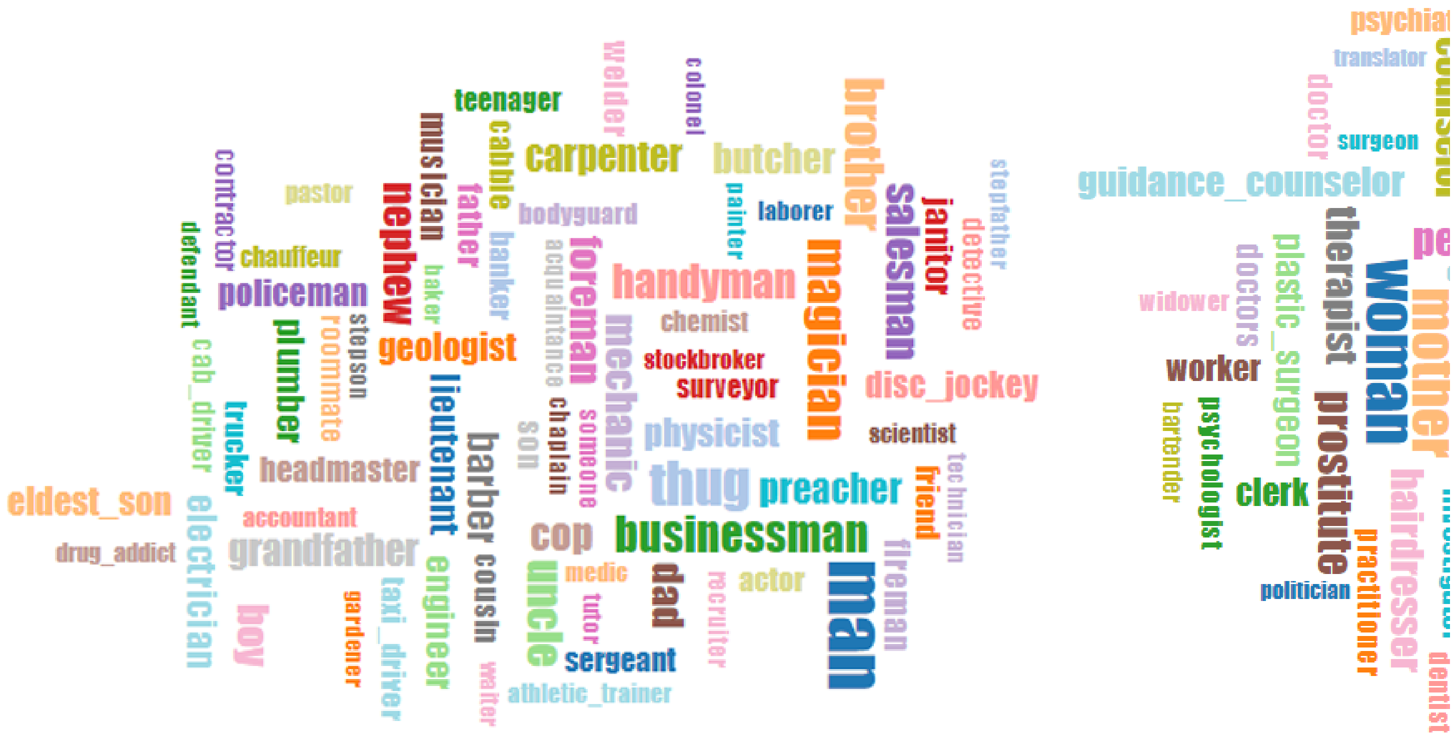
Gender bias, for instance, is a recurring problem. In most words embeddings, you will find that **"man" is to "doctor" what "woman" is to "nurse"**.

```
woman = word_to_vec_map["woman"].reshape(1, -1)
man = word_to_vec_map["man"].reshape(1, -1)
nurse = word_to_vec_map["nurse"].reshape(1, -1)
doctor = word_to_vec_map["doctor"].reshape(1, -1)
```

```
print("Similarity between woman and nurse : ", cosine_similarity(woman, nurse))
print("Similarity between man and nurse : ", cosine_similarity(man, nurse))
print("Conclusion : nurse is a very gendered occupation according to the GloVe embedding")
```

```
Similarity between woman and nurse :  [[0.71550204]]
Similarity between man and nurse :  [[0.57187035]]
Conclusion : nurse is a very gendered occupation according to the GloVe embedding
```

Another way to visualize this bias is through wordclouds : the following schema shows the closest words, in terms of occupation, to the words "man" and "woman".



- "Man" is close to "businessman", "physicist" and "headmaster".
- "Woman" is close to "nurse", "receptionist", "mother" and "prostitute".

> 📚 For more informations on the biases of word embeddings (and why that can be a problem when using those for NLP tasks), you can check out this article (https://towardsdatascience.com/gender-

> bias-word-embeddings-76d9806a0e17) and this academic paper (https://www.aclweb.org/anthology/N19-1064.pdf) .