An introduction to word embeddings

(follows: https://github.com/nlptown/nlp-notebooks/blob/master/An%20Introduction%20to%20Word%20Emb

Training word embeddings

We use Gensim. We use the abstracts of all arXiv papers in the category cs.CL (CL: Computation and Language) published before mid-April 2021 (c. 25_000 documents). We tokenize the abstracts with spaCy.

::: {.cell 0='e' 1='x' 2='p' 3='o' 4='r' 5='t' execution_count=4}

```
import sys
import os
import csv
import spacy

class Corpus(object):
    def __init__(self, filename):
        self.filename = filename
        self.nlp = spacy.blank("en")

    def __iter__(self):
        with open(self.filename, "r") as i:
            reader = csv.reader(i, delimiter=',')
            for _, abstract in reader:
                tokens = [t.text.lower() for t in self.nlp(abstract)]
                yield tokens

documents = Corpus("data/arxiv.csv")
```

:::

Using Gensim we can set a number of parameters for training:

- min_count: the minimum frequency of words in our corpus
- window: number of words to the left and right to make up the context that word2vec will take into account
- vector_size: the dimensionality of the word vectors; usually between 100 and 1_000
- sg: One can choose fro 2 algorithms to train word2vec: Skip-gram (sg) tries to predict the context on the basis of the target word; CBOW tries to find the target on the basis of the context. Default is sg=0, hence: default is CBOW.

::: {.cell 0='e' 1='x' 2='p' 3='o' 4='r' 5='t' execution_count=5}

```python
import gensim

model = gensim.models.Word2Vec(documents, min_count=100, window=5, vector_size=100)
```

:::

Using word embeddings

With the model trained, we can access the word embedding via the **wv attribute** on model using the token as a key. For example the embedding for "nlp" is:

::: {.cell 0='e' 1='x' 2='p' 3='o' 4='r' 5='t' execution_count=6}

```python
model.wv["nlp"]
```

```
array([ 0.886086  , -1.3507926 ,  1.9594816 ,  0.15830246,  0.7997838 ,
       -0.05095133, -0.69671786,  0.3897952 , -0.2648519 ,  1.7748885 ,
        0.91234416, -2.0415502 ,  0.45631358,  0.93007153,  0.12164909,
        1.0863382 , -1.0784775 ,  1.6326956 ,  0.74276346,  1.7070123 ,
        1.4887909 ,  0.14379857, -1.6810449 ,  0.7892891 , -1.4270338 ,
       -2.2591631 ,  1.3879418 ,  1.2997457 ,  1.8315829 , -1.2399971 ,
        2.4112606 , -0.3999709 ,  0.7315848 , -0.536275  ,  0.87647456,
        0.17804186, -1.1123791 , -4.0686617 ,  0.5918724 ,  1.9567369 ,
       -2.0902705 , -1.8516064 ,  3.256888  ,  1.0831386 , -2.84713   ,
       -1.2378649 ,  2.6290998 ,  0.55701196,  0.00663733, -1.5813098 ,
        1.3789881 ,  2.1580083 , -0.12077241, -1.1474407 ,  0.55166465,
        0.8421333 , -0.689064  ,  0.7303354 ,  4.248617  , -0.18814568,
       -0.0495442 ,  2.0214965 , -1.0620869 ,  0.82814294, -0.66422015,
       -0.5761963 , -1.1183583 , -3.0037358 ,  1.2614313 ,  0.13696168,
       -2.0045218 , -1.8689885 ,  0.17104591,  0.81668717,  0.7780453 ,
        0.96487904,  2.3210366 , -1.1554203 , -1.2181611 ,  1.641921  ,
        0.88061166,  2.481447  ,  0.5429929 ,  0.5873016 , -1.5432439 ,
        1.7715456 ,  0.56789696, -0.7806051 , -1.3643101 ,  0.0492861 ,
       -0.10300807,  1.4124674 , -1.5692657 ,  1.321583  , -1.2433962 ,
       -0.45330024,  0.32593846, -0.17630157,  0.27859974, -1.371039  ],
      dtype=float32)
```

:::

Find the similarity between two words. We use the cosine between two word embeddings, so we use a ranges between -1 and +1. The higher the cosine, the more similar two words are.

::: {.cell 0='e' 1='x' 2='p' 3='o' 4='r' 5='t' execution_count=7}

```
print(model.wv.similarity("nmt", "smt"))
print(model.wv.similarity("nmt", "ner"))
```

```
0.65849614
0.38778433
```

:::

Find words that are most similar to target words we line up words via the embeddings: semantically related, other types of pre-tained models, related general models, and generally related words:

::: {.cell 0='e' 1='x' 2='p' 3='o' 4='r' 5='t' execution_count=8}

```
model.wv.similar_by_word("bert", topn=10)
```

```
[('transformer', 0.7813543081283569),
 ('roberta', 0.7682604193687439),
 ('elmo', 0.7174107432365417),
 ('transformers', 0.7109030485153198),
 ('pretrained', 0.6987662315368652),
 ('mbert', 0.6643920540809631),
 ('xlnet', 0.6560472846031189),
 ('xlm', 0.6532952189445496),
 ('lstm', 0.6324996948242188),
 ('gpt-2', 0.6174245476722717)]
```

:::

Look for words that are similar to something, but dissimilar to something else with this we can look for a kind of **analogies**:

::: {.cell 0='e' 1='x' 2='p' 3='o' 4='r' 5='t' execution_count=9}

```
model.wv.most_similar(positive=["transformer", "lstm"], negative=["bert"], topn=1)
```

```
[('rnn', 0.802582323551178)]
```

:::

So a related transformer to lstm is rnn, just like bert is a particular type of transformer; really powerful.

We can also zoom in on **one of the meanings of ambiguous words**. In NLP **tree** has a very specific meaning, is nearest neighbours being: constituency, parse, dependency, and syntax:

::: {.cell 0='e' 1='x' 2='p' 3='o' 4='r' 5='t' execution_count=10}

```
model.wv.most_similar(positive=["tree"], topn=10)
```

```
[('trees', 0.7930562496185303),
 ('constituency', 0.7241601347923279),
 ('parse', 0.7082517147064209),
 ('recursive', 0.706534206867218),
 ('dependency', 0.6458475589752197),
 ('syntax', 0.6273614764213562),
 ('constituent', 0.6265847086906433),
 ('transition', 0.6249228715896606),
 ('parser', 0.6115063428878784),
 ('path', 0.609127402305603)]
```

:::

If we add **syntax** as a negative input to the query, we see that the ordinary meaning of tree kicks in: Now forest is one of the nearest neighbours.

```
#! export
model.wv.most_similar(positive=["tree"], negative=["syntax"], topn=10)
```

```
[('forest', 0.542504072189331),
 ('crf', 0.47240906953811646),
 ('random', 0.45181718468666077),
 ('greedy', 0.4484288692474365),
 ('feed', 0.44518253207206726),
 ('bayes', 0.4260954260826111),
 ('logistic', 0.4220973253250122),
 ('binary', 0.4216032922267914),
 ('top', 0.4214417338371277),
 ('modified', 0.40552183985710144)]
```

Throw a list of words at the model and filter out the odd one (here svm is the only non-neural model):

::: {.cell 0='e' 1='x' 2='p' 3='o' 4='r' 5='t' execution_count=12}

```
print(model.wv.doesnt_match("lst cnn gru transformer svm".split()))
```

```
svm
```

:::

Plotting embeddings

About visualizing embeddings. We need to reduce our 100-dimensions space to 2-dimensions. We can use t-SNE method: map similar data to nearby points and dissimilar data to faraway points in low dimensional space.

t-SNE is present in Scikit-learn. One has to specify two parameters: **n_components** (number of dimensions) and **metric** (similarity metric, here: cosine).

In order NOT to overcrowd the image we use a subset of embeddings of 200 most similar words based on a **target word**.

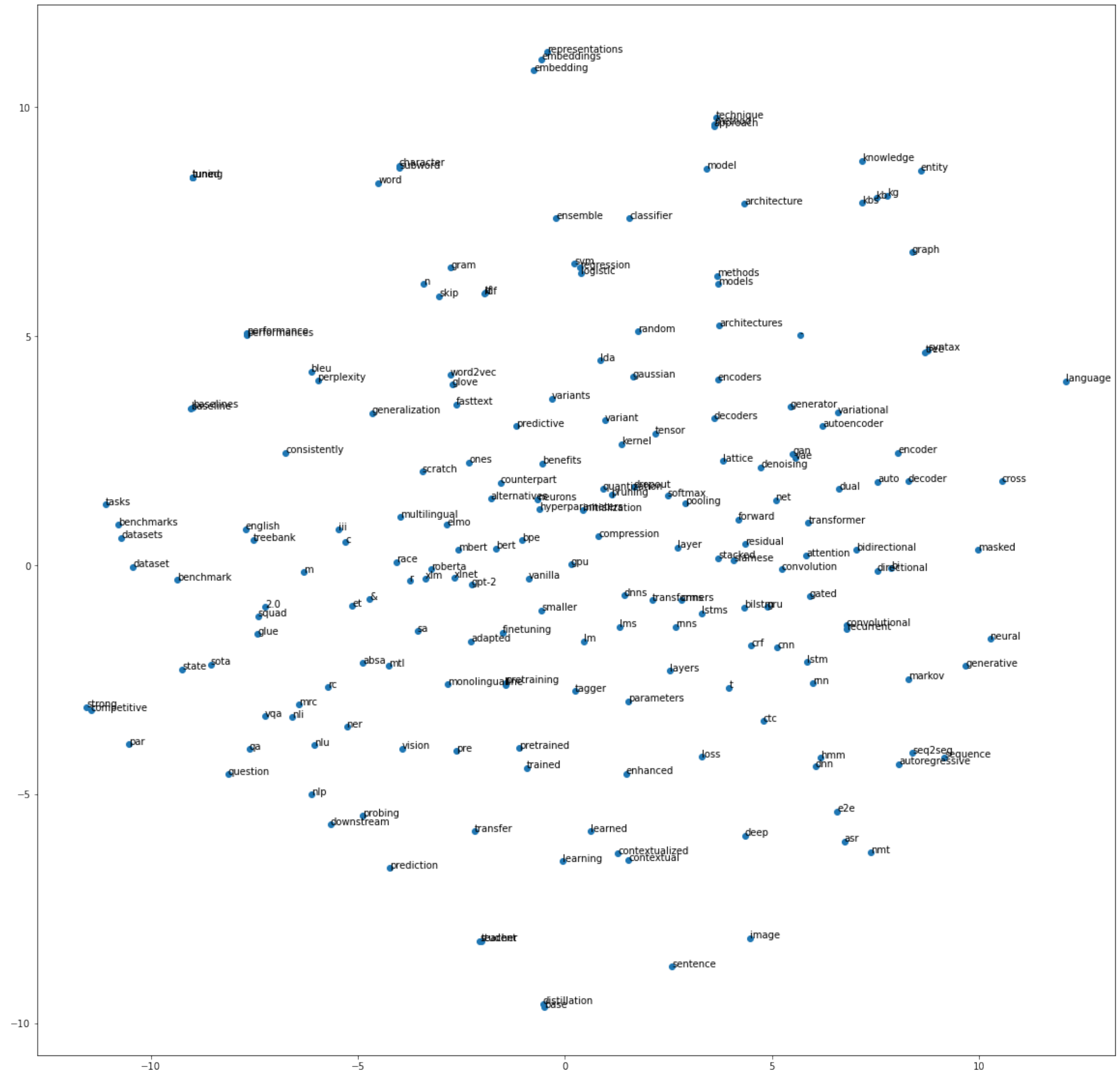
::: {.cell 0='e' 1='x' 2='p' 3='o' 4='r' 5='t' execution_count=13}

```
#%matplotlib inline

import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import sklearn
from sklearn.manifold import TSNE

target_word = "bert"
selected_words = [w[0] for w in model.wv.most_similar(positive=[target_word], topn=200)] +
embeddings = [model.wv[w] for w in selected_words] + model.wv["bert"]

mapped_embeddings = TSNE(n_components=2, metric='cosine', init='pca').fit_transform(embedd
/home/peter/anaconda3/lib/python3.9/site-packages/sklearn/manifold/_t_sne.py:691: FutureWarn
  warnings.warn(
```

:::

::: {.cell 0='e' 1='x' 2='p' 3='o' 4='r' 5='t' execution_count=14}

```
plt.figure(figsize=(20,20))
x = mapped_embeddings[:,0]
y = mapped_embeddings[:,1]
plt.scatter(x, y)

for i, txt in enumerate(selected_words):
    plt.annotate(txt, (x[i], y[i]))
```



:::

Exploring hyperparameters

What is the quality of the embeddings? Should embeddings capture syntax or semantical relations. Semantic similarity or topical relations?

One way of monitoring the quality is to check nearest neighbours: Are they two nouns, two verbs?

::: {.cell 0='e' 1='x' 2='p' 3='o' 4='r' 5='t' execution_count=16}

```
import spacy

nlp = spacy.load('en_core_web_sm')

word2pos = {}
for word in model.wv.key_to_index: # model call can be
    word2pos[word] = nlp(word)[0].pos_

word2pos["translation"]
```

```
'NOUN'
```

:::

::: {.cell 0='e' 1='x' 2='p' 3='o' 4='r' 5='t' execution_count=17}

```
import numpy as np

def evaluate(model, word2pos):
    same = 0
    for word in model.wv.key_to_index:
        most_similar = model.wv.similar_by_word(word, topn=1)[0][0]
        if word2pos[most_similar] == word2pos[word]:
            same = same + 1
    return same/len(model.wv.key_to_index)

evaluate(model, word2pos)
```

```
0.6392384640206519
```

:::

Now we want to change some of the settings we used above:

- embedding size (dimensions of the trained embeddings): 100, 200, 300
- context window: 2, 5, 10

We will use a Pandas dataframe to keep track of the different scores (but this will take time: We train 9 models!!!):

::: {.cell 0='e' 1='x' 2='p' 3='o' 4='r' 5='t' execution_count=18}

```
sizes = [100, 200, 300]
windows = [2,5,10]

df = pd.DataFrame(index=windows, columns=sizes)

for size in sizes:
    for window in windows:
        print("Size:", size, "Window:", window)
        model = gensim.models.Word2Vec(documents, min_count=100, window=window, vector_siz
        acc = evaluate(model, word2pos)
        df[size][window] = acc

df
```

```
Size: 100 Window: 2
Size: 100 Window: 5
Size: 100 Window: 10
Size: 200 Window: 2
Size: 200 Window: 5
Size: 200 Window: 10
Size: 300 Window: 2
Size: 300 Window: 5
Size: 300 Window: 10
```

	100	200	300
2	0.682156	0.679251	0.680865
5	0.656663	0.64666	0.649242
10	0.625363	0.620845	0.610197

:::

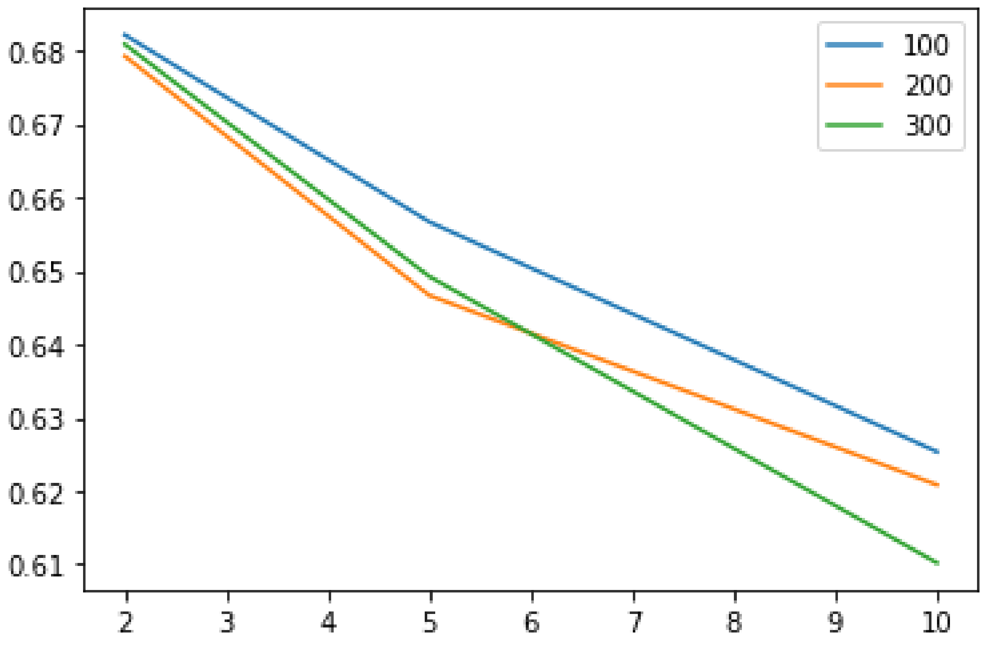
Results are close:

1. Smaller contexts seem to yield better results. Which makes sense because we work with the syntax - nearer words often produce more information.
2. Higher dimension word embeddings not always work better than lower dimension. Here we have a relatively small corpus, not enough data for such higher dimensions.

Let's visualize our findings:

```
df.plot()
```

```
<AxesSubplot:>
```



Conclusions

Word embeddings allow us to model the usage and meaning of a word, and discover words that behave in a similar way.

We move from raw strings -> vector space: word embeddings which allows us to work with words that have a similar meaning and discover new patterns.