# Count Based Distributional Semantics

In this notebook we will implement a simple count based model. We will not try to optimize the model in order to get the best results possible. Instead we will try to keep things as simple as possible. Thus the main principles can become clear.

We will use a mid-sized corpus as a compromise between fast processing and usefull results.

In this notebook we will use German data. However, not much knowledge of German is required. And learning some new German words on the fly is not too bad.

**Caution**

Some of the cells require quite long computation time!

## Data

We use a corpus of 300k sentences from wikipedia collected by the university of Leipzig. You can download the required data from http://wortschatz.uni-leipzig.de/en/download/ . The file used here is **deu_wikipedia_2016_300K-sentences**

### Reading the data

The texts in the corpus are already split into sentences. We read thes sentence and word-tokenize each sentence. To save time we use the infected words and do not do anly lemmatization or stemming.

In [ ]:
```
import codecs
import nltk

sentences = []

#Caution: change the path to this file!
source = codecs.open('/mnt/Farjad_Ahmed/Masters/NLP/Tutorials/Tutorial_5/deu_wikipedia_2016_300K/deu_wikipedia_2016_300K-senten
for line in source:
    nr,sent = line.split('\t')
    sentences.append(nltk.word_tokenize(sent.strip(),language='german'))
```

Let us check whether the senteces are in the list as we expect them to be:

In [ ]:
```
sentences[447]
```

Out[ ]:
```
['1819',
 'wurde',
 'daher',
 'ein',
 'neues',
 '„',
 'Vereidigungsbuch',
 '“',
 'angelegt',
 '.']
```

## Vectors of Co-occurrence Values

We write a function that computes vectors with co-occurrence numbers for a number of words with a given list of 'context' words. As context words we take all words that exceed a specified minimum frequency. KCo-occurrence with rare words will hardly contribute to the comparison between context vectors. Another parameter that needs to be set is the maximum distance between two words to count as co-competition, known as the window_size size. We proceed sentence by sentence here. This means that the last word of a sentence and the first word of the next sentence are never counted as co-occurring. Note, however, that sentence bondaries are often not taken into account in such procedures. It is not clear whether this has a seriosu impact. Another detail to note is that we first calculate the window size for the competition, and then determine the relevant words in the window. Alternatively, you can first remove all irrelevant words and then determine the words within the window.

In general, a larger window can compensate for a too small corpus. In addition, a smaller window captures more syntactical properties of a word, while a larger window takes into account broader semantic relationships.

When all co-occurrence values are calculated, we normalize the length of the vectors.

In [ ]:
```
```

In [ ]:
```
from scipy import sparse
from collections import Counter
import numpy as np

def mag(x):
    return np.sqrt(x.dot(x))
```

```
def makeCV(words,sentences, window = 2, minfreq = 10):
    #first count all words
    freq = Counter()
    for s in sentences:
        freq.update(s)

    #determine which words have to be used as features or context words
    context_words = [w for w,f in freq.items() if f > minfreq]

    #we add all words to the context words, if they are not already in that list.
    #Just for convenience to have all words in one list.
    for w in words:
        if w not in context_words:
            context_words.append(w)
    dim = len(context_words)

    #Give each context word a unique number and build dictionaries to switch between numbers and words
    cw2nr = {}
    for i in range(dim):
        cw = context_words[i]
        cw2nr[cw] = i

    w2nr = {}
    for i in range(len(words)):
        w = words[i]
        w2nr[w] = i


    #initialize a matrix
    #We use  a sparse matrix class from scipy
    matrix = sparse.lil_matrix((len(words),dim))

    #Now we start the real work. We iterate through all sentences and count the co-occurrences!
    for s in sentences:
        i_s = [cw2nr.get(w,-1) for w in s]
        for i in range(len(s)):
            w = s[i]
            if w in words:
                i_w = w2nr[w]
                for j in range(max(0,i-window),min(i+window+1,len(s))):
                    if i != j: # a word is not in its own context!
                        i_cw = i_s[j]
                        if i_cw > 0:
                            matrix[i_w,i_cw] += 1

    #finally make a dictionary with vectors for each word
```

```
    wordvectors = {}
    for w,i_w in w2nr.items():
        v = matrix[i_w].toarray()[0]
        wordvectors[w] = v/mag(v)
    return wordvectors
```

Let us test the function for a short list of words:

In [ ]:
```
vectors_count =  makeCV(['Kloster','Kirche','Garten','Haus','Hof','Schweden','Deutschland','betrachten','anschauen','beobachten
```

Since the vectors have the same length, we can use the inner product, which is identical to the cosine, for comparison:

In [ ]:
```
print(vectors_count['Schweden'].dot(vectors_count['Deutschland']))
print(vectors_count['Schweden'].dot(vectors_count['Hof']))
```

```
0.8824327413213885
0.7023764205372964
```

Next we want to know, what words are most similar to a given word. To do so, we need to compare a word with each other word in the list. We use an ordered list to store the results. Since these list always sort ascending, we need to consider always the last elements of this list. Finally, we return the results in inverse order.

In [ ]:
```
import bisect

def most_similar(word,vectors,n):
    best = []
    vec_w = vectors[word]
    for z in vectors:
        if z == word:
            continue
        sim = vec_w.dot(vectors[z])

        #we have to add this result only, if we do not yet have n results, or if the similarity is larger than the similarity w
        if len(best) < n or sim > best[0][0]:
            bisect.insort(best,(sim,z))
            best = best[-n:]

    return best[::-1] #present the list in descending order
```

In [ ]:
```
most_similar('Garten',vectors_count,3)
```

Out[ ]:
```
[(0.8131085029186811, 'Hof'),
 (0.6601100880142219, 'Haus'),
 (0.6391321442093744, 'Schweden')]
```

It is more interesing to find the most similar word if we have more words to choose from. Let us collect some mid-frequency words to do so.

In [ ]:
```
wortfrequenz = Counter()

for satz in sentences:
    wortfrequenz.update(satz)

vocabulary  = [w for w,f in wortfrequenz.items() if 30 < f < 3000]
vocabulary_size = len(vocabulary)
print(vocabulary_size)
```

```
12177
```

Now it takes some time to compute all vectors.

In [ ]:
```
vectors_count = makeCV(vocabulary,sentences,window=2)
```

In [ ]:
```
most_similar('Garten',vectors_count,10)
```

Out[ ]:
```
[(0.9303729732770968, 'Park'),
 (0.8713290722481782, 'Saal'),
 (0.8638230056577244, 'Tempel'),
 (0.8631281311361956, 'Bezirk'),
 (0.8618980713054947, 'Sturm'),
 (0.8599988689679838, 'Keller'),
 (0.8571963797713257, 'Raum'),
 (0.854935433012446, 'Wald'),
 (0.8531439466922369, 'Kreis'),
 (0.8458564830673464, 'Sinn')]
```

In [ ]:
```
most_similar('betrachten',vectors_count,10)
```

Out[ ]:
```
[(0.9585602463943912, 'verstehen'),
 (0.9577321881251438, 'ermitteln'),
 (0.9575302761329436, 'schaffen'),
 (0.9566353415343976, 'bauen'),
 (0.9549942222425168, 'beobachten'),
 (0.9537790612894577, 'verwenden'),
 (0.9525533613296538, 'bringen'),
 (0.9523625237497214, 'erhöhen'),
 (0.9484461037256293, 'kämpfen'),
 (0.9478641686123344, 'retten')]
```

In [ ]:
```
most_similar('Schweden',vectors_count,10)
```

Out[ ]:
```
[(0.9466501415959565, 'Ungarn'),
 (0.9385311272990015, 'Polen'),
 (0.9344181199344914, 'Frankreich'),
 (0.930491672233815, 'Australien'),
 (0.9287830698511249, 'Spanien'),
 (0.9231790664595388, 'Russland'),
 (0.9228546620449983, 'Italien'),
 (0.9221253661112269, 'England'),
 (0.9194553370244143, 'Kanada'),
 (0.9179109485171332, 'Brasilien')]
```

In [ ]:
```
most_similar('Direktor',vectors_count,10)
```

Out[ ]:
```
[(0.9483059267409419, 'Leiter'),
 (0.941059344345683, 'Chef'),
 (0.9373318244835647, 'Kommandeur'),
 (0.9343354455869818, 'Vorsitzender'),
 (0.9231897847174313, 'Präsident'),
 (0.9108755997814568, 'Mitglied'),
 (0.8921792527539255, 'Vizepräsident'),
 (0.8718734003027769, 'Vorstandsmitglied'),
 (0.8696699002069143, 'Sekretär'),
 (0.8601018930922202, 'Kommandant')]
```

# Evaluation

The examples look nice, but how good are our vectors? There are a number of tests that can be done to check the quality of the vectors. One type of test is to compare the calculated similarity between two words to the similarity that subjects have given for word pairs. We can simply use the correlation to check the similarity.

A dataset with similarity assessments for German word pairs is the so-called Gur350 dataset, which was developed by the working group of Prof. Dr. Iryna Gurevych at the TU Darmstadt. Further information and a link for the download can be found here: https://www.informatik.tu-darmstadt.de/ukp/research_6/data/semantic_relatedness/german_relatedness_datasets/index.en.jsp

The data does not give a similarity between the words but a relationship, which is not exactly the same. For the evaluation, we only use the word pairs for which both words are contained in our vocabulary.

In [ ]:
```
import math
testfile = codecs.open('/mnt/Farjad_Ahmed/Masters/NLP/Tutorials/Tutorial_5/datasets/wortpaare350.gold.pos.txt','r','utf8')
```

```
testfile.readline()

testdata = []
missing = set()
for line in testfile:
    w1,w2,sim,p1,p2 = line.split(':')
    if w1 in vocabulary and w2 in vocabulary:
        testdata.append((w1,w2,float(sim)))

def evaluate(data,vectors):
    gold = []
    predicted = []
    for v,w,sim in data:
        pred = vectors[v].dot(vectors[w])

        gold.append(sim)
        predicted.append(pred)
        #print(v,w,pred,sim,sep = '\t')

    av_p = sum(predicted)/len(predicted)
    av_g = sum(gold)/len(gold)

    cov = 0
    var_g = 0
    var_p = 0
    for s,t in zip(gold,predicted):
        cov += (s-av_g) * (t-av_p)
        var_g += (s-av_g) * (s-av_g)
        var_p += (t-av_p) * (t-av_p)

    return cov / math.sqrt(var_g*var_p)
```

In [ ]:
```
len(testdata)
```

Out[ ]:
```
119
```

Note, that we are using only a very small part of the test data from the Gur350 dataset for testing. Thus we cannot compare our results to official results for these data! We can of course include more words in our vocabulary of mid-frequency words, but many words simpy are not in our corpus.

In [ ]:
```
evaluate(testdata,vectors_count)
```

Out[ ]:
```
0.17255411250703448
```

Despite the good looking lists that we generated above, we see that the calculated similarities correlate only very weakly with the similarity judgments.

## Somewhat more advanced

Our first attempt still offers various possibilities for optimization. First of all we could not use the simple co-occurrence frequencies but the *Positive Pointwise Mutual Information* (PPMI). The *Pointwise Mutual Information* between two words a and b is in principle the ratio between the actual probability that they will occur together and the expected probability that they will occur if their occurrences were independent. We define $pmi(a,b) = log(\frac{p(ab)}{p(a)p(b)})$. The ppmi(a, b) is now the pmi(a, b) if this value is positive, and otherwise 0. The use of the ppmi is based on the assumption that only the fact that words occur together more often than expected is interesting, while lower probabilities are more likely to be based on coincidence or in any case do not say anything interesting about word pairs.

Another problem with our naive approach was that the vectors are extremely long. The vectors not only need a lot of storage space. Many of the dimensions also contain little or only redundant information. This problem can be solved by using a common dimension reduction process. Below we will use Singular Value Decomposition for this and then use the most important 100 dimensions.

In [ ]:
```
from sklearn.decomposition import TruncatedSVD
import math
import numpy as np

def makeCV_SVD(words,sentences, window = 2, minfreq = 10, size =256):

    freq = Counter()
    for s in sentences:
        freq.update(s)

    context_words = [w for w,f in freq.items() if f > minfreq]
    for w in words:
        if w not in context_words:
            context_words.append(w)
    dim = len(context_words)
    cw2nr = {}
    for i in range(dim):
        cw = context_words[i]
        cw2nr[cw] = i

    w2nr = {}
    for i in range(len(words)):
        w = words[i]
        w2nr[w] = i
```

```
matrix = sparse.lil_matrix((len(words),dim))

n = 0
for s in sentences:
    n+=1
    i_s = [cw2nr.get(w,-1) for w in s]
    for i in range(len(s)):
        w = s[i]
        if w in words:
            i_w = w2nr[w]
            for j in range(max(0,i-window),min(i+window+1,len(s))):
                if i != j:
                    i_cw = i_s[j]
                    if i_cw > 0:
                        matrix[i_w,i_cw] += 1

#up to here nothing new
N = matrix.sum()

#Let us  get the probabilities for each word:
freq_w = matrix.sum(axis = 1)
freq_w = np.array(freq_w.T)[0]
prob_w = np.array(freq_w) / N

#Let us  get the probabilities for each context word:
freq_cw = matrix.sum(axis = 0)
freq_cw = np.array(freq_cw)[0]
prob_cw = np.array(freq_cw) / N

(rows,cols) = matrix.nonzero() #Returns a tuple of arrays (row,col) containing the indices of the non-zero elements of the
for i_w,i_cw in zip(rows,cols):
    p = matrix[i_w,i_cw]/N
    p_w = prob_w[i_w]
    p_cw = prob_cw[i_cw]
    ppmi = max(0,math.log(p/(p_w * p_cw) ))
    matrix[i_w,i_cw] = ppmi


#We cannot be completely sure, that for every word we found at least some positive pmi-values.
#The frequent neighbors of a word eventaully are not in the set of context words
#A row with only zeros, will cause a problem for the SVD
for i in range(len(words)):
    if np.sum(matrix[i]) == 0:
        print("Empty row for:",words[i])
        print("Please remove this word from the wordlist.")
```

```
    svd = TruncatedSVD(n_components=size)
    svd.fit(matrix)
    matrix = svd.transform(matrix)
    wordvectors = {}
    for w,i_w in w2nr.items():
        v = matrix[i_w]
        wordvectors[w] = v/mag(v)
    return wordvectors
```

In [ ]:
```
vectors_SVD = makeCV_SVD(vocabulary,sentences,window=2, size=100)
```

In [ ]:
```
most_similar('Garten',vectors_SVD,10)
```

Out[ ]:
```
[(0.7656588064751748, 'Saal'),
 (0.7207829276642457, 'Wohnhaus'),
 (0.7207142783164258, 'Grundstück'),
 (0.7152888403695544, 'Schlosspark'),
 (0.7135164388009677, 'Friedhof'),
 (0.7116116278337536, 'Brunnen'),
 (0.7098123982917997, 'Haus'),
 (0.7097884871703362, 'Ensemble'),
 (0.7015669408292721, 'Pavillon'),
 (0.6983357459655675, 'Gebäudekomplex')]
```

In [ ]:
```
most_similar('betrachten',vectors_SVD,10)
```

Out[ ]:
```
[(0.800376297961646, 'denken'),
 (0.7565373133084523, 'wenden'),
 (0.7562864699024485, 'erklären'),
 (0.7491746524505113, 'untersuchen'),
 (0.7422653877970952, 'tun'),
 (0.7410471936873776, 'erinnern'),
 (0.7382061553935914, 'bezeichnen'),
 (0.7373671888102061, 'sprechen'),
 (0.7343977368084303, 'verstehen'),
 (0.7339052023461071, 'handeln')]
```

In [ ]:
```
most_similar('Schweden',vectors_SVD,10)
```

Out[ ]:
```
[(0.9162071217833722, 'Ungarn'),
 (0.8816609385956848, 'Frankreich'),
 (0.8716921487018372, 'Norwegen'),
 (0.8706114254949588, 'Finnland'),
 (0.8667800502642821, 'Polen'),
 (0.8657185236417015, 'Brasilien'),
 (0.8651073124270825, 'Italien'),
 (0.8644199260561896, 'Dänemark'),
 (0.8629187503645185, 'Rumänien'),
 (0.8627403799360186, 'Spanien')]
```

In [ ]:
```
most_similar('Park',vectors_SVD,10)
```

Out[ ]:
```
[(0.7858793030972667, 'Forest'),
 (0.7723566700190202, 'Valley'),
 (0.7615404010928544, 'Bay'),
 (0.760159925800872, 'Airport'),
 (0.75384902217407, 'Avenue'),
 (0.752376341608628, 'Castle'),
 (0.734140734019549, 'River'),
 (0.7339341247914842, 'Manhattan'),
 (0.7338995440774598, 'Central'),
 (0.7338595472828965, 'Bridge')]
```

We notice that park is not interpreted as a synonym for garden, but rather is perceived as part of English place names. The context 'English words' has apparently become more important.

In [ ]:
```
evaluate(testdata,vectors_SVD)
```

Out[ ]:
```
0.5621115517023889
```

Take the last model from the notebook DistrSem1.ipynb and download the small set of synonyms and non-synonym word pairs STW German Synonyms from http://textmining. wp.hs-hannover.de/datasets.html. Rank the words form this dataset according to their predicted similarity.

1. Evaluate the ranking using AUC
2. Investigate whether and how the result depends on the following parameters. Test at least 2 parameters, better 3 or all 4. a) Window size b) Corpus size (i.e. use only a part of the corpus) c) Lower and upper frequency bound of words to be included as context words. d) Number of dimensions You do not have to test all combinations. Just test each of the parameters independently while fixing the other parameters to a reasonable value. Note that you have to change frequency ranges for word inclusion if you reduce the corpus size! Finally, if you are learning German, just for fun, you could try to rank the word pairs yourself (not looking at the labels) and compute the AUC of your personal ranking!

The optimization was apparently successful: the correlation has become significantly larger. Bullinaria and Levy (2007), Mohammad and Hirst (2012),

Bullinaria and Levy (2012) and Kiela and Clark (2014) provide overviews of the combinations of parameters, training quantities, etc. that lead to optimal results.

In [ ]:
```
import numpy as np
from sklearn import metrics

stw = codecs.open('STW-Syn.txt','r','utf8')


testdata_stw = []
for line in stw:
    w1,w2,sim = line.strip().split('\t')
    if w1 in vocabulary and w2 in vocabulary:
        testdata_stw.append((w1,w2,sim))

def evaluate_bin(data,vectors):
    labels = []
    predictions = []
    for v,w,sim in data:
        pred = vectors[v].dot(vectors[w])
        labels.append(sim)
        predictions.append(pred)
    fpr, tpr, thresholds = metrics.roc_curve(labels, predictions, pos_label='+')
    return metrics.auc(fpr, tpr)
```

In [ ]:
```
evaluate_bin(testdata_stw,vectors_count)
```

Out[ ]:
```
0.5695592286501377
```

In [ ]:
```
from sklearn.decomposition import TruncatedSVD
import math

def makeCV_SVD(words,sentences, window = 2, minfreq = 10, size =256):
    N = 0
    freq = Counter()
    for s in sentences:
        N += len(s)
        freq.update(s)

    context_words = [w for w,f in freq.items() if f > minfreq]
    for w in words:
        if w not in context_words:
            context_words.append(w)
    dim = len(context_words)
```

```
    cw2nr = {}
    for i in range(dim):
        cw = context_words[i]
        cw2nr[cw] = i

    w2nr = {}
    for i in range(len(words)):
        w = words[i]
        w2nr[w] = i


    matrix = sparse.lil_matrix((len(words),dim))
    n  =  0
    for s in sentences :
        n += 1
        i_s = [cw2nr . get (w ,-1) for w in s]
        for  i  in  range (len(s)):
            w  =  s[i]
            if w in words :
                i_w = w2nr[w]
                for j in range(max(0,i-window),min(i+window+1,len(s))):
                    if i != j:
                        i_cw = i_s[j]
                        if i_cw > 0:
                            matrix[i_w,i_cw] += 1


    probabilities = []
    for cw in context_words:
        probabilities.append(freq[cw]/N)

    N2 = matrix.sum()/2
    (rows,cols) = matrix.nonzero()
    for i_w,i_cw in zip(rows,cols):
        p_w = probabilities[i_w]
        p_cw = probabilities[i_cw]
        p = matrix[i_w,i_cw]/N2
        ppmi = max(0,math.log(p/(p_w * p_cw * 2 * window) ))
        matrix[i_w,i_cw] = ppmi

    svd = TruncatedSVD(n_components=size)
    svd.fit(matrix)
    matrix = svd.transform(matrix)
    wordvectors = {}
    for w,i_w in w2nr.items():
        v = matrix[i_w]
```

```
        wordvectors[w] = v/mag(v)
    return wordvectors
```

In [ ]:
```
vectors_SVD = makeCV_SVD(vocabulary,sentences,window=3,size=50)
```

```
/tmp/ipykernel_28439/3406290792.py:63: RuntimeWarning: invalid value encountered in true_divide
  wordvectors[w] = v/mag(v)
```

In [ ]:
```
wordlist = ['Guten', 'Schweden', 'Park']
out = []
for w in wordlist:
    out.append(most_similar(w,vectors_SVD,10))
print(out)
```

```
[[(0.8995127802671076, 'Mensch'), (0.8968436141268652, 'Menschheit'), (0.8922717188782779, 'Wirklichkeit'), (0.886289398182873, 'Seele'), (0.884582083665708, 'Alles'), (0.8799719982434838, 'Hoffnung'), (0.8798842901305821, 'Freiheit'), (0.8780358138034768, 'Herzen'), (0.8722102724041599, 'wirklich'), (0.8718914424311429, 'Inhalt')], [(0.9587682422089914, 'Brasilien'), (0.949968023470248, 'Finnland'), (0.9437857649002306, 'Portugal'), (0.9422672286566536, 'Norwegen'), (0.940137910673187, 'Südafrika'), (0.9393354308197659, 'Belgien'), (0.9369897591788723, 'Neuseeland'), (0.9332148559484487, 'Russland'), (0.9324117070340221, 'Rumänien'), (0.9322421387382575, 'Südkorea')], [(0.9016053402777532, 'Street'), (0.8954645259592968, 'Castle'), (0.8815114307156195, 'Station'), (0.8812594734162735, 'Point'), (0.8761696335065321, 'Town'), (0.874540305265088, 'Lake'), (0.872313829681639, 'Fort'), (0.8710674608574857, 'District'), (0.8704109238462485, 'River'), (0.8672713522034854, 'Beach')]]
```

In [ ]:
```
evaluate(testdata,vectors_SVD)
```

Out[ ]:
```
0.4554505431937267
```

In [ ]:
```
evaluate_bin(testdata_stw,vectors_SVD)
```

Out[ ]:
```
0.8849862258953168
```

In [ ]:
```
vectors_SVD = makeCV_SVD(vocabulary,sentences,window=4,size=50)
```

```
/tmp/ipykernel_28439/637546162.py:62: RuntimeWarning: invalid value encountered in true_divide
  wordvectors[w] = v/mag(v)
```

In [ ]:
```
wordlist = ['Guten', 'Schweden', 'Park']
out = []
for w in wordlist:
    out.append(most_similar(w,vectors_SVD,10))
print(out)
```