Word similarity: Practical implementation

Benjamin Roth

Centrum für Informations- und Sprachverarbeitung
Ludwig-Maximilian-Universität München
beroth@cis.uni-muenchen.de

Word similarity model

demo

Word similarity model: steps

- What steps are neccessary to implement a word similarity model?
- (That is, to return based on a corpus the most similar words to a query word)

Word similarity model: steps

- preprocessing (tokenization, ...)
- define vocabulary, assignment of word $\Rightarrow$ numbering
- counting co-occurrences of words
- create co-occurrences matrix X. Line: (target) word, column: (context) word. Value: frequency of common occurrence.
- weighting of the matrix.
  positive pointwise mutual information (PPMI): Measures how much coincidence deviates from statistically expected frequency.

Word similarity model: steps

- singular value decomposition (SVD):
  - The matrix is decomposed into a product of three matrices:

$$X = U\Sigma V^T$$

  - Σ is a diagonal matrix, with descending sorted, non negative values. Size of a value $\Leftrightarrow$ share/importance in the reconstruction of X.
  - U: Matrix. Line: word, column: context representation. The columns contain the context information, compressed and sorted by importance!
  - V: Matrix. Equally optimized representation of the context.
- calculate similarity
  - Vector for request word: corresponding line in $U\Sigma$.
  - Calculate cosine similarity with vectors for all other words (i.e., with all words in $U\Sigma$).
  - **Why not just use U?**

Singular value decomposition: repetition

Singular value decomposition in Python: “naive” approach

- idea of SVD:
  - Consider only the n (e.g., 50) most important singular vectors.
  - $\Rightarrow$ Statistical “noise” is removed by ignoring other dimensions.
  - $\Rightarrow$ better similarity comparison, less outliers.

```
import numpy as np
U, sigma, V = np.linalg.svd(X)
U_trunc = U[:,:n]
sigma_trunc = sigma[:n]
V_trunc = V[:,:n]
```

$\Rightarrow$ Problem of the above approach?

Singular value decomposition in Python: "naive" approach

```
import numpy as np
U, sigma, V = np.linalg.svd(X)
U_trunc = U[:,:n]
sigma_trunc = sigma[:n]
V_trunc = V[:,:n]
```

$\Rightarrow$ Problem of the above approach?
The co-occurrence matrix is **sparse** *(99% or more of the entries are 0). The matrices U and V are* **dense***, so their calculation will require a factor of 100 (or more) more space. The truncated matrices are again very small because only a fraction of the columns are retained; the calculation of the intermediate step is often not possible due to space limitations.*

Singular value decomposition in Python: efficient approach

- There are special methods that compute only the n largest singular values and the associated vectors. (Or directly the truncated matrix $U\Sigma$, which is of interest.)
- These methods are also optimized for the calculation of sparse matrices.
- For our purposes: `sklearn.decomposition.TruncatedSVD`

```python
from sklearn.decomposition import TruncatedSVD

svd = TruncatedSVD(n_components=n)
U_sigma_trunc = svd.fit_transform(X)
```

Most similar words: “Naive” approach

1. Lookup vector for request word: Line in `X` or `U_sigma_trunc`, depending on whether the co-occurrence information is to be used with or without SVD.
2. Iterate over all words w in the vocabulary.
    1. Looking up vector for w.
    2. Calculate cosine similarity and append to list along with w.
3. Sort list by similarity.

Most similar words: “Naive” approach

```
class DenseSimilarityMatrix:
    def __init__(self, U_sigma_matrix, word_to_id):
        self.word_matrix = U_sigma_matrix
        self.word_to_id = word_to_id
        self.id_to_word = {w:i for i,w in self.word_to_id.items()}

    def most_similar_words(self, query, n):
        q_row = self.word_to_id[query]
        q_vec = self.word_matrix[q_row,:]
        dot_q_q = q_vec.dot(q_vec.T)
        sims_words = []
        for w in self.word_to_id:
            w_row = self.word_to_id[w]
            w_vec = self.word_matrix[w_row,:]
            dot_w_w = w_vec.dot(w_vec.T)
            dot_q_w = q_vec.dot(w_vec.T)
            sim = dot_q_w / math.sqrt(dot_q_q * dot_w_w)
            sims_words.append((sim, w))
        return [w for s,w in sorted(sims_words, reverse=True)[:n]]
```

Most similar words: “Naive” approach

Problem with above approach?

- From a theoretical point of view, not much to object.
- Nevertheless extremely inefficient in practice (factor $>$ 10-100):
  - Create single objects for each vector.
  - Expand the list.
  - Cosine calculation separately for each vector.
- $\Rightarrow$ Matrix multiplication is one of the most optimized operations in mathematical program libraries.
- $\Rightarrow$ Whenever possible, you should multiply matrices as a whole!

Most similar words: Efficient approach

- Avoid lookup of vectors for single words in vocabulary.
- For cosine calculation we need:
    - Dot product vectors of all words with request vector:
      $\Rightarrow$ Efficient Operation: multiplication of matrix with vector!
    - Dot product inquiry vector (one time operation).
    - Dot product of all vectors in vocabulary.
      $\Rightarrow$ efficient operation:
        - Componentwise multiplication of the matrix with itself.
        - Sum of the result.
        - Example: Whiteboard.
    - component-wise application of root and fractional calculation.

Most similar words: Efficient approach (for numpy dense arrays)

```
def most_similar_words(self, word, topn):
    row = self.word_to_id[word]
    vec = self.word_matrix[row,:]
    m = self.word_matrix
    dot_m_v = m.dot(vec.T) # vector
    dot_m_m = np.sum(m * m, axis=1) # vector
    dot_v_v = vec.dot(vec.T) # float
    sims = dot_m_v / (math.sqrt(dot_v_v) * np.sqrt(dot_m_m))
    return [self.id_to_word[id] for id in (-sims).argsort()[:topn]]
```

- For scipy sparse matrices, the syntax is a bit different.
- Hint: `vec` is a row vector, `vec.T` gives the corresponding column vector. (Dot product is defined for row and column vectors)

Most similar words: Efficient approach

- `v.argsort()` returns the indices of the sorted entries of a vector:

```
>>> v=np.array([5,1,1,4])
>>> v.argsort()
array([1, 2, 3, 0])
```

Summary

- Singular value decomposition (SVD)
    - Repetition
    - Application on co-occurrence matrices
    - Efficient calculation of the truncated SVD
- Similarity calculation with matrix multiplication