Computer-Linguistische Anwendungen

CLA | B.Sc. | LMU

Wort-Ähnlichkeitsmodell: Schritte

- Welche Schritte müssen gemacht werden, um ein Wort-Ähnlichkeitsmodell zu implementieren? (D.h. um auf Basis eines Korpus die ähnlichsten Wörter zu einem Anfrage-Wort zurückzuliefern)

    - Vorverarbeitung (Tokenisierung, ...)
    - Vokabular definieren, Zuweisung Wort) → Numerierung
    - Cooccurrence (Zusammenauftreten) von Wörtern zählen
    - Cooccurrence-Matrix C erstellen.
        Zeile: (Ziel-)Wort,
        Spalte: (Kontext-)Wort.
        Wert: Häufigkeit des gemeinsamen Vorkommens.

    - Gewichtung der Matrix.
      Positive Pointwise Mutual Information (PPMI): Misst wie sehr Zusammenauftreten von statistisch erwarteter Häufigkeit abweicht.

Wort-Ähnlichkeitsmodell: Schritte

Singulärwertzerlegung (SVD)

Die Matrix wird in ein Produkt aus drei Matrizen zerlegt:

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

- **Σ**: ist eine Diagonalmatrix, mit absteigend sortierten, nicht-negativen Werten. Größe eines Wertes , Anteil/Wichtigkeit bei der Rekonstruktion von X.
- **U**: Matrix. Zeile: Wort, Spalte: Kontext-Repräsentation. Die Spalten enthalten die Kontext-Information, komprimiert und nach Wichtigkeit sortiert!
- **V** : Matrix. Ebenso optimierte Repräsentation des Kontextes.

Ähnlichkeit berechnen:

- Vektor für Anfragewort: Entsprechende Zeile U Σ
- Kosinus-Ähnlichkeit mit Vektoren für alle anderen Wörter berechnen (d.h. Mit allen Wörtern in UΣ)
- **Warum nicht nur U verwenden?**

Singulärwertzerlegung in Python: “Naiver” Ansatz

Idee bei SVD:

- Betrachte nur die ***n*** (z.B. 50) wichtigsten Singulärvektoren
  → statistisches “Rauschen” wird entfernt, indem anderen Dimensionen ignoriert werden.
  → besserer Ähnlichkeitsvergleich, weniger Ausreißer

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

Warum naiv?

Singulärwertzerlegung in Python: “Naiver” Ansatz

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

→ Problem des obigen Ansatzes?
*Die Cooccurrence-Matrix ist **sparse** (99% oder mehr der Einträge sind 0). Die Matrizen U und V sind **dense**, bei ihrer Berechnung wird also ein um den Faktor 100 (oder mehr) größerer Speicherplatz benötigt. Die trunkierten (abgeschnittenen) Matrizen sind zwar wieder sehr klein, weil nur ein Bruchteil der Spalten beibehalten wird; die Berechnung des Zwischenschrittes ist oft aber aus Speicherplatzgründen so nicht möglich.*

Singulärwertzerlegung in Python: **Effizienter** Ansatz

- Es gibt spezielle Methoden, die nur die ***n*** größten Singulärwerte und die dazugehörigen Vektoren berechnen. (Bzw. direkt die trunkierte Matrix UΣ, die von Interesse ist.)
- Diese Methoden sind auch für die Berechnung von Sparse-Matrizen optimiert.
- Für unsere Zwecke: `sklearn.decomposition.TruncatedSVD`

```
from sklearn.decomposition import TruncatedSVD
svd = TruncatedSVD(n_components=n)
U_sigma_trunc = svd.fit_transform(X)
```

Ähnlichste Wörter: “Naiver” Ansatz

1. Nachschlagen von Vektor für Anfragewort: Zeile in `X` oder `U_sigma_trunc`, je nach dem ob die Cooccurence-Information mit oder ohne SVD verwendet werden soll.

2. Iterieren über alle Wörter w im Vokabular.
   a. Nachschlagen von Vektor für w.
   b. Kosinus-Ähnlichkeit berechnen, und zusammen mit w an Liste anhängen.

3. Liste nach Ähnlichkeit sortieren.

Ähnlichste Wörter: “Naiver” Ansatz

```
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]]
```

Ähnlichste Wörter: “Naiver” Ansatz

Problem mit obigem Ansatz?

- Aus theoretischer Sicht nicht viel einzuwenden.
- Trotzdem in der Praxis extrem ineffizient (Faktor > 10-100):
  - Erstellen von Einzelobjekten für jeden Vektor.
  - Erweitern der Liste.
  - Kosinus-Berechnung separat für jeden Vektor

→ Matrix-Multiplikation ist eine der am meisten optimierten Operationen in mathematischen Programm-Bibliotheken.
→ Wann immer möglich, sollte man Matrizen als Ganzes multiplizieren!

Ähnlichste Wörter: **Effizienter** Ansatz

- Vermeide Nachschlagen (Lookup) der Vektoren für Einzelwörter im Vokabular.
- Für Kosinus-Berechnung brauchen wir:
  - Dot-Product Vektoren aller Wörter mit Anfrage-Vektor:
    → Effiziente Operation: Multiplikation von Matrix mit Vektor!
  - Dot-Product Anfrage-Vektor (Einmalige Operation).
  - Dot-Produkt aller Vektoren im Vokabular.
    → Effiziente Operation:
    - Komponentenweise Multiplikation der Matrix mit sich selbst.
    - Summierung des Ergebnisses.
  - komponentenweise Anwendung von Wurzel und Bruchrechnung

Ähnlichste Wörter: **Effizienter** Ansatz

```
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]]
```

Für Scipy Sparse-Matrizen ist die Syntax etwas verschieden.
Hinweis: `vec` ist ein Zeilenvektor, `vec.T` ergibt den dazugehörigen Spaltenvektor. (Dot-Produkt ist definiert für Zeilen- mit Spaltenvektor)

Ähnlichste Wörter: **Effizienter** Ansatz

`v.argsort()` liefert die Indizes der sortierten Einträge eines Vektors:

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

Zusammenfassung

- Singulärwertzerlegung (SVD)
  - Wiederholung
  - Anwendung auf Cooccurrenz-Matrizen
  - Effiziente Berechnung der trunkierten SVD

- Ähnlichkeitsberechnung mit Matrix-Multiplikation