Anagramme, Komposita und Levenshtein-Abstand

In dieser Übung geht es um die Extraktion von Anagrammen und Komposita aus einem Textkorpus und um die Berechnung des Levenshtein-Abstandes zwischen Zeichenketten.

Laden Sie zunächst das Textkorpus von der URL `http://www.cis.uni-muenchen.de/~schmid/lehre/data/zeit-10M-tagged.gz` herunter und dekomprimieren Sie die Datei.

Anagramme extrahieren

Schreiben Sie ein Programm, welches alle Wörter in der ersten Spalte der Datei einliest, die nur aus Buchstaben bestehen, und dann alle Anagramme (z.B. Lampe, Palme, Ampel) ausgibt.

Aufruf:

```
python anagramme.py zeit.tagged
```

Die Ausgabe erfolgt auf den Bildschirm und sollte so aussehen:

```
Ampel Lampe Palme
Ernst Stern
Stuben buntes
...
```

Die Liste der Wörter in jeder Zeile soll absteigend nach Korpushäufigkeit sortiert sein. Wörter mit einer Häufigkeit von unter 10 werden ignoriert. Listen mit nur einem Element sollen nicht ausgegeben werden. Wenn ein Wort in zwei Schreibungen auftritt (bspw. “Klein” und “klein”), geben Sie nur die häufigere aus.

Vorüberlegungen

- Welche Eigenschaft ist allen Wörtern eines Anagrammes gemeinsam?
- Wie kann man die Menge aller Anagramme schnell (in linearer Zeit) finden?
- Welche Datenstrukturen sind sinnvoll?

Implementieren Sie das Programm.

Komposita aufspalten

Hier sollen Sie die Komposita-Zerlegungsstrategie von Philipp Koehn implementieren (`https://aclanthology.org/E03-1076.pdf`). Die Grundidee von Koehns Methode besteht darin, ein gegebenes Kompositum auf alle möglichen Arten so zu zerlegen, dass jedes

Element der Zerlegung mindestens drei Buchstaben umfasst und in einer Wortliste (unter Ignorierung der Groß-/Kleinschreibung) enthalten ist. Jede mögliche Zerlegung eines Kompositums (inkl. des nicht zerlegten Wortes) wird dann mit dem geometrischen Mittel der Worthäufigkeiten seiner Elemente bewertet.

Ausnahme: Das Fugenelement 's' darf zwischen zwei Elementen auftauchen und wird bei der Scoreberechnung nicht berücksichtigt und auch nicht als Teil der Zerlegung ausgegeben.

Sie sollen ein Programm schreiben, welches aus der ersten Spalte des Zeit-Korpus alle Nomen (Wortart "NN"), die nur aus einem Großbuchstaben und einer Folge von mindestens 2 Kleinbuchstaben bestehen, mit ihren Häufigkeiten extrahiert und diese dann mit der beschriebenen Methode in Teilwörter zerlegt. Auch die Teilwörter sollen Nomen aus dem Zeitkorpus sein.

Verwenden Sie für die Zerlegung eine rekursive Funktion, welche die Ergebnisse mit dem Python-Befehl *yield* zurückgibt.

Aufruf:

```
python split-compounds.py zeit.tagged
```

Die möglichen Zerlegungen jedes Wortes werden nach abnehmender Bewertung sortiert und im folgenden Format auf den Bildschirm ausgegeben:

```
Arbeitsamt 33.9 Arbeit Amt
Arbeitsamt 19 Arbeitsamt
Arbeitsamt 1.7 Arbeit Samt
Abteilungen 112 Abteilungen
Abteilungen 20.1 Abtei Lungen
...
```

Achtung: Dies sind nicht die echten Zerlegungen und Bewertungen, die sie für das obige Korpus erhalten!

Vorüberlegungen

- Welche Schritte muss Ihr Programm ausführen?
- Welche Datenstrukturen verwenden Sie am besten?
- Wie werden die Zerlegungen berechnet und bewertet?
- Wie behandeln Sie am besten das Fugen-s (Beispiel: *Arbeit-s-amt)*?

Levenshtein-Abstand

Der Levenshtein-Abstand (String-Edit-Distanz) berechnet die minimale Anzahl von Löschungs-, Einfügungs- und Ersetzungsoperationen, die notwendig sind, um einen String

a in einen String b zu überführen. Beispielsweise kann *aufgetaucht* mit zwei Löschoperationen, einer Ersetzungsoperation und einer Einfügeoperation in *auftauchen* transformiert werden.

Den Levenshtein-Abstand von $a = a_1...a_n$ und $b = b_1...b_m$ berechnen Sie mit Hilfe einer Matrix d mit den Dimensionen $(n+1) \times (m+1)$, deren Einträge $d[i,k]$ den minimalen Levenshtein-Abstand zwischen dem Präfix $a_1...a_i$ und dem Präfix $b_1...b_k$ angeben. $d[i,k]$ wird mit dynamischer Programmierung wie folgt berechnet:

$$d[i,k] := min \begin{pmatrix} d[i-1,k]+1, \\ d[i-1,k-1]+[a_i \neq b_k], \\ d[i,k-1]+1 \end{pmatrix}$$

Der Klammeroperator $[P]$ bildet den Wahrheitswert P auf 1 ab, falls P wahr ist, und sonst auf 0. $d[n,m]$ liefert den Levenshtein-Abstand von a und b. $d[0,0]$ wird mit 0 initialisiert.

Schreiben Sie ein Programm, welches eine Datei mit einem Wortpaar pro Zeile einliest (mit Tabulator als Trennzeichen) und jeweils den Levenshtein-Abstand berechnet und auf den Bildschirm ausgibt. Halten Sie sich bei der Berechnung des Levenshtein-Abstandes an die obige Formel.

Aufruf: `python levenshtein.py wordpairs.txt`

Erweitern Sie anschließend das Programm so, dass es zusätzlich alle optimalen Alignierungen der Buchstaben berechnet und ausgibt. Die optimalen Alignierungen des Stringpaares *aa a* sind:

a:a a:
a: a:a

Vorüberlegungen

- Welcher Operation entspricht jeder Term in der Minimum-Operation oben jeweils?
- Rechnen Sie das Beispiel *abgibt-abgegeben* durch.
- Die Felder in der ersten Zeile und der ersten Spalte der Matrix sind Spezialfälle. Überlegen Sie, was hier anders ist.
- Wie erweitern Sie den Code, um die optimalen Alignierungen der Buchstabenfolgen ausgeben zu können?

Achtung: Bei der Programmierung dürfen nur die Standard-Bibliotheken von Python verwendet werden.

Bitte schicken Sie die drei Programme bis zum angegebenen Abgabetermin an `schmid@cis.lmu.de`.