Übung 4: Grosse Datenmenge und Dynamisches Programmieren

Programmiertechniken in der Computerlinguistik II, FS 2019

Abgabe: 30. April, 23:59

Allgemeine Hinweise

- Die Übungen dürfen zu zweit gelöst werden.
- Bitte gib jede Quellcode-Datei als *importierbares Modul* ab. Teste dies, indem du (im selben Verzeichnis) den Python-Befehl `import` *`Dateiname`* (ohne die Endung *.py*) ausführst.
- Bei einigen Aufgaben muss ein bestimmtes Interface erfüllt werden, das in Form einer Funktionssignatur definiert ist. Stelle sicher, dass du das Interface einhältst, indem
  - der Dateiname,
  - der Funktionsname und
  - die Anzahl, Reihenfolge und Typ der Argumente

  mit den Vorgaben übereinstimmen. Bei der Implementierung hast du hingegen freie Hand; z. B. ist es oft sinnvoll, die Funktionalität auf weitere Funktionen aufzuteilen.
- Aller Programm-Code ist grundsätzlich mit Docstrings und, wo nötig, mit Kommentaren zu versehen.
- **ACHTUNG: Eure Abgabe müsst ihr für diese Übung nicht auf OLAT laden, sondern in ein GitHub Repository. Folgt dazu der Anleitung im Repository, in dem auch dieses PDF liegt.**
- Die Daten, die ihr von uns zur Verfügung gestellt bekommt und die ihr nicht verändert, sondern nur einlest, müssen nicht abgegeben werden (z. B. Korpora). **Falls ihr Korpusdateien auf eurem Rechner speichert, legt diese im Verzeichnis `Korpusdaten` ab. Dadurch werden sie bei Commits und Pushes ignoriert.**
- Bitte schreib deinen Namen als Kommentar an den Anfang deiner Skripte. Falls ihr die Übung zu zweit löst, gebt beide Namen an.

Praktische Tipps

In dieser Übung wirst du mit grossen Datenmengen arbeiten. Dabei müssen ein paar Dinge beachtet werden:

- Plane genügend Zeit ein. Das Herunterladen, Entpacken und Verarbeiten kann je nach deiner Situation (Netzwerkverbindung, Prozessorgeschwindigkeit) lange dauern.
- Sei auf einen allfälligen Systemabsturz vorbereitet! Falls dein Code ein 'Speicherleck' hat, kann dies zu einem Speicherüberlauf führen, der dein System abstürzen lässt. Überwache während der Laufzeit deines Programmes deine Speicherauslastung mit einem Task-Manager.
- Das Korpus, das du für Aufgabe 1.1 herunterlädst, ist mit bz2 komprimiert. Mit dem Unix-Werkzeug `bzip2` kannst du es entpacken, falls du das für nützlich erachtest. Vergiss dabei die Flag `-k` nicht, damit das komprimierte File nicht verschwindet. Mit `bzless` kannst du das File inspizieren, ohne es entpacken zu müssen (auf Windows mit `bzcat file_in_question.bz2 | more` als Workaround).

1 Verarbeitung grosser Datenmengen

1.1 Deduplizierung

Dein Freund Timotheus will einen Chatbot bauen, der besonders lakonische Antworten gibt. Wie gewohnt ist er im Verzug, und weil du ihm bereits einmal geholfen hast, kommt er wieder auf dich zurück. Er verlangt von dir, ihm ein Korpus mit möglichst vielen lakonischen Repliken aufzubauen.

Ziel dieser Aufgabe ist es, aus einer grossen Sammlung von Reddit-Kommentaren (2 GB) geeignete Kommentare auszuwählen und diese in ein neues File zu schreiben. Das Korpus findest du hier[1]. Im rohen Korpus ist jeder Kommentar einzeln als JSON-Objekt abgelegt. In den jeweiligen Objekten hat es neben dem eigentlichen Kommentar auch noch viele Meta-Informationen dazu, so zum Beispiel den Nutzernamen des Kommentierenden oder einen Zeitstempel. Das meiste davon kannst du ignorieren. Im neuen Korpus soll jeder Kommentar nur einmal vorkommen, und auf einer Zeile soll jeweils genau ein Kommentar stehen.

Um eine gewisse Qualität und 'Lakonie' zu gewährleisten, sollst du die Kommentare filtern. Es soll drei Filterkriterien geben: Die Anzahl Upvotes (Likes), die unter dem Key `score` gegeben sind, eine Mindest- sowie eine Maximallänge. Die Länge bezieht sich auf die Anzahl Zeichen.

[1] Dieses Korpus (und viele andere im Zusammenhang mit Reddit) wird von Pushshift zur Verfügung gestellt, einem Projekt zur Analyse von Big-Data. Disclaimer: Reddit ist ein ungefilterter Spiegel einer Internet-Community. Betrachtet den Inhalt als Wissenschaftler*innen.

Interface: Schreibe eine Funktion `mk_meme_corpus`, welche als Argumente das zum Lesen im Binärmodus geöffnete Korpus, den Namen des Ausgabekorpus sowie die angegeben Filterparameter enthält:

```
def mk_meme_corpus(infile: BinaryIO,
                   outfile: str,
                   min_score: int=100,
                   min_len: int=1,
                   max_len: int=50):
```

Die Funktion soll keine Rückgabewert besitzen, aber ein mit `gzip` komprimiertes Korpus unter dem angegebenen Namen anlegen.

Abgabe: Abzugeben ist ein importierbares Modul `comment_picker.py`, welches die Funktion `mk_meme_corpus` implementiert. Es sollen *KEINE* Korpusdateien abgeben werden!

1.2 Randomisierung

Für viele Anwendungen im maschinellen Lernen muss man seine Daten in ein Trainings-, ein Development- und ein Testset aufteilen. Die Idee dahinter ist, dass man sein Modell mit Daten testet, die es während des Trainings noch nicht gesehen hat. So kann man ein Modell daraufhin trimmen, dass es nicht 'auswendiglernt', sondern gut verallgemeinern kann.

In dieser Aufgabe sollst du ein grosses Korpus im XML-Format in genau diese drei Sets aufteilen. Wenn man zu Beginn festlegt, wie gross das Test- und das Dev-Set sein sollen, eignet sich besonders Knuths 'Algorithm R' für diese Aufgabe. Hier findest du das Korpus (1.3 GB), das dein Skript aufteilen und in drei aufbereiteten Dateien wieder ausgeben soll.

Das Korpus beinhaltet die Abstracts von verschiedenen akademischen Texten. In den Ausgabedateien soll jeweils auf einer Zeile der Inhalt eines einzelnen Abstracts (XML-Tag `<document>`) stehen. Als Inhalt interessieren uns nur die Sätze (XML-Tag `<sentence>`), die Titel und Untertitel sollen nicht mit ausgegeben werden. Die einzelnen Sätze eines Abstracts sollen mit Leerzeichen verbunden sein.

Interface: Schreibe eine Funktion `split_corpus`, welche als Argumente das zum Lesen im Binärmodus geöffnete Korpus, das Ausgabeverzeichnis und die Grösse des Test- und Dev-Sets verlangt.

```
def split_corpus(infile: BinaryIO,
                 targetdir: str,
                 n=1000: int):
```

Die Funktion hat keinen Rückgabewert, soll aber im angegebenen Ausgabeverzeichnis drei komprimierte Dateien erzeugen:
abstracts.txt.training.gz
abstracts.txt.test.gz
abstracts.txt.development.gz

Die Abstracts, die für das Test- oder Dev-Set ausgewählt wurden, dürfen natürlich nicht auch im Trainings-Set stehen.

Am besten gehst du bei dieser Aufgabe wieder schrittweise vor. Stelle zuerst sicher, dass du gefahrlos durch das XML-File iterieren und die Ausgabesätze zusammenbauen kannst, ohne deinen Arbeitsspeicher zu überlasten. Erst sobald das funktioniert, solltest du dich an die Implementation des Algorithm R wagen. Aber Achtung: Auch hier musst du auf den Arbeitsspeicher aufpassen.

Abgabe: Abzugeben ist ein importierbares Modul `corpus_splitter.py`, das die Funktion `split_corpus` implementiert. Es sollen *KEINE* Korpusdateien abgegeben werden!

2 Dynamisches Programmieren

In der Vorlesung hast du erfahren, wie man das Problem der minimalen Editierdistanz mit dynamischen Programmieren lösen kann. Das dynamische Programmieren ist aber auch noch für viele andere Fälle eine nützliche Möglichkeit, ein Problem sehr effizient zu lösen. So können damit auch die **längsten gemeinsamen Substrings** zweier Strings bestimmt werden.

Das Vorgehen dazu sieht folgendermassen aus: Für jede mögliche Präfixkombination der beiden Strings wird die Länge der gemeinsamen Endung dieser Präfixe bestimmt. Hier siehst du die resultierende Tabelle für die Strings *Meisterklasse* und *Kleistermasse*:

j

i

	[]	M	E	I	S	T	E	R	K	L	A	S	S	E
[]	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0	**2**	0	0	0	0
E	0	0	**1**	0	0	0	**1**	0	0	0	0	0	0	**1**
I	0	0	0	**2**	0	0	0	0	0	0	0	0	0	0
S	0	0	0	0	**3**	0	0	0	0	0	0	**1**	**1**	0
T	0	0	0	0	0	**4**	0	0	0	0	0	0	0	0
E	0	0	**1**	0	0	0	**5**	0	0	0	0	0	0	**1**
R	0	0	0	0	0	0	0	**6**	0	0	0	0	0	0
M	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0
S	0	0	0	0	**1**	0	0	0	0	0	0	**2**	0	0
S	0	0	0	0	**1**	0	0	0	0	0	0	0	**3**	0
E	0	**1**	0	0	0	**1**	0	0	0	0	0	0	0	**4**

Im Feld für die Präfixe *Mei* und *Klei* steht eine 2, denn die Länge ihrer gemeinsamen Endung (*ei*) ist 2. Im Feld für die Präfixe *Meis* und *Klei* hingegen steht eine 0, denn sie haben keinen gemeinsamen Endung, sie enden mit einem unterschiedlichen Buchstaben. Aus der Tabelle kann man ablesen, dass der längste Substring die Länge 6 hat. Wenn man die Strings an dieser Stelle um 6 Zeichen zurückgeht, erhält man den längsten gemeinsamen Substring, nämlich *eister*.

Interface: Schreibe eine Funktion `longest_substrings`, welche als Argumente zwei verschiedene Strings annimmt und den/die längsten Substring/s in einer Iterable zurückgibt.

```
def longest_substrings(x: str, y: str) -> Iterable[str]:
```

Es soll nicht zwischen Gross- und Kleinschreibung unterschieden werden. Falls es kein einziges gemeinsames Zeichen gibt, soll die Funktion `None` zurückgeben.

Ein guter Ausgangspunkt für die Implementierung ist die oben aufgeführte Tabelle. Überlege dir, wie genau die Zahlen in den einzelnen Zelle zustande kommen.
Achtung: Für nicht-dynamische Lösungen werden Punkte abgezogen. Auch wenn es möglich ist, jede Zelle isoliert zu berechnen, ist das Ziel dieser Übung, auf bereits berechnete Werte zurückzugreifen und so Rechenzeit zu sparen.
Ein paar Beispiele zum Rückgabewert der Funktion:

```
>>> longest_substrings('Tod', 'Leben')
None
>>> longest_substrings('Haus', 'Maus')
['aus']
>>> longest_substrings('mozart','mozzarella')
```

```
['moz','zar']
>>> longest_substrings('keep the interface!', 'KeEp ThE iNtErFaCe!')
['keep the interface!']
```

Abgabe: Abzugeben ist ein importierbares Modul `fun_with_strings.py`, welches die Funktion `longest_substrings` implementiert.

Reflexion/Feedback

a) Fasse deine Erkenntnisse und Lernfortschritte in zwei Sätzen zusammen.

b) Wie viel Zeit hast du in diese Übungen investiert?