# MT Übung 2

Thema: Preprocessing und Postprocessing

Abgabetermin: Montag, 26. März 2018, 18 Uhr

**Hinweise zur Abgabe:**

- **Abgabeformat: zip**
- Bitte benennt eure Datei nach folgendem Schema: olatbenutzername_mt_uebungxx.zip, z.B. `mmuster_mt_uebung02.zip`
- Für diese Übung werdet ihr ein Python-Skript, eine Reihe von Dateien mit Preprocessing, Übersetzung und Postprocessing sowie einen kurzen Bericht zu dieser Übung abgeben. Gebt alles in derselben ZIP-Datei ab.
- Die Abgabe erfolgt über den Übungsbaustein auf OLAT. Bitte gebt pünktlich ab, denn der Baustein ist nur bis Montag, 18 Uhr offen.

# 1 Byte Pair Encoding (BPE)

In der Vorlesung habt ihr Byte Pair Encoding, kurz BPE, kennengelernt. Damit ihr euch noch besser vorstellen könnt, wie das funktioniert, werdet ihr in dieser Aufgabe ein Programm schreiben, das ein BPE-Modell lernt(`learn_bpe.py`). Folgt dazu den folgenden Schritten **(1.-4. je 1 Punkt, 5. 0.5 Punkte)**:

1. Als ersten Schritt gehen wir davon aus, dass unser Vokabular nur aus einzelnen Zeichen besteht. Schreibt eine Funktion, die eine Datei einliest, auf der ihr euer BPE-Modell lernen wollt. Manipuliert den Text dann so, dass ihr eine Liste von Token bekommt, wobei jedes Token als eine Liste von Zeichen repräsentiert wird (ihr könnt davon ausgehen, dass der Input schon tokenisiert ist). Das letzte Zeichen jedes Tokens soll noch mit einer speziellen Wortgrenzmarkierung ergänzt werden (</w>). Gebt dann diese Liste aus der Funktion zurück. Hier ein Beispiel:

   ```
   ein Satz .  # Inhalt einer Datei

   [["e","i","n</w>"],["S","a","t","z</w>"],[".</w>"]]  # Rückgabe der Funktion
   ```

2. Als Nächstes müssen wir herausfinden, welches Bigramm von Symbolen aus unserem Vokabular am häufigsten vorkommt. Schreibt dafür eine Funktion, die eine Liste im Format oben als Argument bekommt und zuerst Bigramme von allen Symbolen erstellt. Achtet euch darauf, dass ihr keine Bigramme beachtet, welche über die Tokengrenze hinausgehen. Dann sollt ihr für alle Bigramme berechnen, wie oft sie vorkommen. Dazu könnt ihr die Klasse `Counter` benutzen, welche euch automatisch die Häufigkeit der Bigramme berechnet. Ein Beispiel für die Benutzung von `Counter` findet ihr unten. Gebt den `Counter` aus der Funktion zurück.

3. Jetzt wissen wir, welches das häufigste Bigramm ist und müssen alle Symbole, aus denen das Bigramm zusammengesetzt ist, in unserem Text zusammenfügen. Schreibt dazu eine Funktion, die den Text im Format aus Funktion 1 und das häufigste Bigramm als Argumente bekommt und dann jedes Mal, wenn das Bigramm im Text vorkommt, die beiden Symbole zusammenklebt, sodass es nur noch ein Symbol ist. Die Funktion soll die neue Liste mit den zusammengefügten Symbolen zurückgeben. Hier ein Beispiel:

   ```
   ("e","i")  # häufigstes Bigram

   [["ei","n</w>"],["S","a","t","z</w>"],[".</w>"]]  # Rückgabe der Funktion
   ```

4. Schreibt noch eine zusätzliche Funktion, die den Text im Format aus Funktion 1 und die Anzahl gewünschter zusätzlicher Symbole bekommt. Die Funktion soll dann Funktionen 2 und 3 so lange nacheinander aufrufen, bis ihr entweder alle Symbole zusammengefügt habt (sprich ihr habt wieder die originalen Token) oder bis ihr die gewünschte Anzahl zusätzlicher Symbole dem Vokabular hinzugefügt habt. Speichert alle Symbole die ihr hinzufügt in einer Liste und gebt diese anschliessend zurück.

5. Schreibt zum Schluss noch die `main` Funktion, welche zuerst Funktion 1 und dann Funktion 4 aufruft. Anschliessend soll das Vokabular in einer Datei gespeichert werden, damit ihr euer gelerntes Modell später auf einen Text anwenden könnt. Damit euer Modell vom offiziellen `apply_bpe.py` Skript richtig gelesen werden kann, müsst ihr auf der ersten Zeile "#version: 0.2" einfügen und danach jedes Bigramm (die beiden Symbole getrennt mit einem Leerzeichen) auf eine neue Zeile schreiben. Hier ein Beispiel wie das Ausgabeformat aussehen soll:

```
#version: 0.2
e i
ei n
S a
...
```

Um euer Modell jetzt auf einen Text anzuwenden, könnt ihr das offizielle Skript von Rico Sennrich benutzen, welches sich im Aufgabenordner befindet (`apply_bpe.py`).

**Achtung:** Die oben beschriebene Implementation ist nicht besonders effizient. Das heisst, wenn ihr ein BPE Modell auf einem grossen Text lernen möchtet und viele Symbole hinzufügt, wird das Programm sehr lange laufen. Zugunsten der Verständlichkeit haben wir auf Optimierungen verzichtet. Wenn ihr möchtet, dürft ihr aber gerne versuchen, das Programm effizienter zu machen.

**Abgabe:** Ein Python-Skript, in welchem die oben beschriebenen Funktionen implementiert sind. Ihr könnt eure Implementation testen, indem ihr ein BPE Modell mit 150 Symbolen auf der Datei "test_bpe.txt" lernt. Euer Modell sollte dann so aussehen wie die Datei "test_bpe.codes". Wenn ihr euer Modell mit dem `apply_bpe.py` Skript auf die "test_bpe.txt" Datei anwendet, sollte dasselbe herauskommen wie die Datei "test_bpe.bpe":

```
python apply_bpe.py -i test_bpe.txt -o test_bpe_your.bpe -c test_bpe_your.codes
```

wobei "test_bpe_your.codes"der Output eures Skripts ist.

Ein Beispiel für den Gebrauch von `Counter`:

```
1 from collections import Counter
2
3 bigrams = [("e","i"),("i","n</w>"),("S","a"),("a","t"),("t","z</w>")]
4 freq_counter = Counter(bigrams)
5 print(freq_counter.most_common(1)) # gibt das haeufigste Bigramm aus
```

## 2 Installation und Vorbereitung

In dieser Übung werdet ihr bereits das erste Mal ein neuronales MÜ-System zum Übersetzen benutzen. Dafür müsst ihr sicherstellen, dass ihr alle notwendigen Voraussetzungen habt. Im Detail sollt ihr überprüfen, ob ihr folgende Einrichtungen bei euch installiert habt:

1. **Python**

   Das Framework, welches wir in dieser Übung benutzen, erfordert Python 2.7. Mit folgendem Befehl könnt ihr überprüfen, welche Default-Version von Python ihr bei euch installiert habt:

   ```
   python --version
   ```

   Falls Python 2.7 nicht eure Default-Version ist, könnt ihr ganz einfach eine virtuelle Python 2.7 Umgebung erstellen. Dafür könnt ihr `virtualenv`[1] benutzen. Folgt dazu den Anweisungen in Punkt 2.

   Auch wenn Python 2.7 eure Default-Version ist, ist es trotzdem empfehlenswert, eine virtuelle Umgebung für diesen Kurs zu erstellen, da wir auch in späteren Übungen damit arbeiten werden.

   Falls ihr gar keine Version von Python 2.7 auf eurem Computer habt, könnt ihr sie hier installieren: `https://www.python.org/downloads/`.

   Wenn ihr mit Windows arbeitet oder Probleme bei den weiteren Schritten habt, die ihr nicht lösen könnt, gibt es auch die Möglichkeit auf dem Studentenserver des Instituts (r2d2) zu arbeiten. Dort sind einige dieser Einrichtungen schon installiert. Ihr könnt euch einfach per Email[2] bei mir melden, wenn ihr Zugang zum Server benötigt.

2. **Virtualenv**

   Falls Python 2.7 nicht eure Default-Version von Python ist und ihr Anaconda nicht installiert habt, verwendet virtualenv, um eine virtuelle Python 2.7 Umgebung zu erstellen. Führt in eurer Kommandozeile die folgenden Schritte aus:

   - `pip install virtualenv`
     - um virtualenv zu installieren
   - `virtualenv --python=python2.7 mt_env`
     - um eine neue virtuelle Umgebung namens “mt_env” zu kreieren
   - `source activate mt_env`
     - um in die neue Umgebung “mt_env” zu wechseln

[1] `https://virtualenv.pypa.io/en/stable/`
[2] chantal.amrhein@uzh.ch

Wenn ihr wieder aus der virtuellen Umgebung raus möchtet, ruft diesen Befehl auf:

```
source deactivate
```

3. **Packages**

   Installiert von eurer neuen virtuellen Umgebung aus die foldenden Packages. Diese braucht ihr für das Framework, welches wir zum Übersetzen benutzen. Führt dazu den folgenden Befehl aus (in der virtuellen Umgebung):

   ```
   pip install numpy numexpr cython tables theano bottle bottle-log tornado
   ```

4. **Frameworks**

   Das erste Framework, welches wir für diese Übung benutzen heisst Nematus. Damit werdet ihr ein trainiertes Modell zum Übersetzen benutzen können. Ihr sollt euch eine lokale Kopie von Nematus auf euren Computer holen. Benutzt dafür den folgenden Befehl:

   ```
   git clone https://github.com/EdinburghNLP/nematus
   ```

   Weiterhin benötigen wir ein Framework für das Pre- und Postprocessing vor und nach dem Übersetzen. Das beinhaltet viele der Schritte, über die ihr in der Vorlesung gesprochen habt, z.B. Normalisierung von Zeichen, Casing und Tokenisierung. Holt euch das benötigte Framework, die Codebase von Moses, mit den folgenden Befehlen:

   - `git clone https://github.com/moses-smt/mosesdecoder`
     - für generelles Pre- / Postprocessing

5. **NMT System**

   Zum Schluss benötigen wir noch ein trainiertes Modell, um übersetzen zu können. Dazu könnt ihr ein bereits trainiertes System hier herunterladen:

   `http://data.statmt.org/rsennrich/wmt16_systems/de-en/model.npz`

   Speichert das heruntergeladene Modell im "model_de_en" Ordner (welcher sich in diesem Übungsordner befindet). Alle sonstigen Dateien, die fürs Übersetzen notwendig sind, findet ihr auch in diesem Ordner: ein deutsches Vokabular, ein englisches Vokabular, das benutzte BPE-Modell und das benutzte Truecase-Modell.

**Achtung: Falls ihr bei einem dieser Schritte Probleme habt, schaut, ob Google euch weiterhelfen kann, meldet euch im Forum oder kommt im Tutorat vorbei. Wie in Teilschritt 1 beschrieben, gibt es bei auch die Möglichkeit, dass ihr auf dem Studentenserver arbeitet. Meldet euch per Email bei mir für einen Zugang.**

# 3 Preprocessing

In der Vorlesung habt ihr gehört, dass das Preprossessing von Daten für maschinelle Übersetzung einige Schritte beinhaltet. In dieser Übung sollt ihr den Zeit-Text "text.de" der letzten Übung übersetzen. Dafür müsst ihr zuerst folgende Schritte des Preprossessing ausführen:

1. **Normalisieren**

   Als Erstes normalisieren wir Sonderzeichen im Text. Das heisst z.B., dass gerichtete Anführungs- und Schlusszeichen vereinheitlicht werden. Führt dazu den folgenden Befehl aus:

   ```
   perl mosesdecoder/scripts/tokenizer/normalize-punctuation.perl \
    < text.de > text.norm.de
   ```

2. **Tokenisieren**

   Als Nächstes tokenisieren wir den Text. Führt dazu den folgenden Befehl aus:

   ```
   perl mosesdecoder/scripts/tokenizer/tokenizer.perl -l de -q \
    < text.norm.de > text.tok.de
   ```

3. **Truecasing**

   Nun truecasen wir den Text. Das bedeutet, dass wir ein gelerntes Modell benutzen, um zu entscheiden, ob wir z.B. am Satzanfang die Grossschreibung eines Tokens beibehalten oder nicht. Führt dazu den folgenden Befehl aus (das bereits trainierte Truecase-Modell befindet sich im "model_de_en" Ordner):

   ```
   perl mosesdecoder/scripts/recaser/truecase.perl \
   --model model_de_en/truecase-model.de < text.tok.de > text.tc.de
   ```

4. **BPE**

   Jetzt wenden wir BPE auf den Text an. Leider könnt ihr nicht ein eigenes gelerntes Modell verwenden. Damit wir dieselben Symbole haben wie im Trainingsmaterial, benutzt das bereits gelernte Modell von Rico Sennrich, welches sich auch im "model_de_en" Ordner befindet:

   ```
   python apply_bpe.py -i text.tc.de -o text.bpe.de -c model_de_en/deen.bpe
   ```

# 4 Übersetzen

Hier werdet ihr zum ersten Mal einen Text übersetzen. Wir verwenden dafür das Nematus-Modell im "model_de_en" Ordner. Führt die folgenden Befehle aus:

1. `export THEANO_FLAGS=mode=FAST_RUN,floatX=float32,device=cpu,on_unused_input=warn`
   - um festzulegen, dass auf der CPU dekodiert wird
2. `cd model_de_en`
   - um in den Ordner mit dem Nematus-Modell zu wechseln
3. ```
   python ../nematus/nematus/translate.py  \
   -m model.npz -k 12 -p 1 --suppress-unk < ../text.bpe.de > ../text.out.en
   ```
   - um den Text zu übersetzen (kann eine Weile dauern)

# 5 Postprocessing

Nach dem Übersetzen muss man noch ein paar Schritte als Postprocessing durchführen:

1. `sed "s/\@\@ //g" < text.out.en > text.nobpe.en`
   - um BPE wieder zusammenzusetzen
2. ```
   perl mosesdecoder/scripts/recaser/detruecase.perl \
   < text.nobpe.en > text.ntc.en
   ```
   - um das Truecasing wieder rückgängig zu machen
3. ```
   perl mosesdecoder/scripts/tokenizer/detokenizer.perl -l en -q \
   < text.ntc.en > text.detok.en
   ```
   - um den Text zu detokenisieren
4. ```
   perl mosesdecoder/scripts/tokenizer/normalize-punctuation.perl \
    < text.detok.en > text.norm.en
   ```
   - um den Text nochmals zu normalisieren

**Abgabe für Aufgaben 3, 4 und 5:** Alle Dateien, die ihr beim Preprocessing, Übersetzen und Postprocessing erstellt **(0.5 Punkte)** habt plus ein kurzer Text, was ihr in dieser Übung gelernt habt **(1 Punkt)**. Ihr sollt auch das BLEU-Skript aus Übung 1 verwenden, um den BLEU Wert des Nematus-Modells mit DeepL, Google und Bing zu vergleichen und das Ergebnis ebenfalls in eurem Bericht diskutieren.

Bei Problemen oder Unklarheiten postet bitte einen Beitrag ins OLAT-Forum.
Viel Erfolg!