Introduction

Since I already programmed the IBM Model 1 for a course on Statistical Machine Translation, the submitted code is just a polished and commented version of it. I am also attaching a draft of the paper *Leveraging Neural Machine Translation for Word Alignment*, which you may find interesting. Please do not share it outside of your team as it is currently in a review process. It covers three topics: (1) using an NMT to get word alignments, (2) various hard alignment extractor methods and (3) combination of all in a small neural network. The extractor methods are in brackets. Hard alignment extractors are reused from this projects and this document describes them very superficially and without mathematical notation. The original version was based on Kohen's pseudocode, which he recently revised. For the graphics I also used Slow Align, a small tool for displaying word alignment (needs major reworking).

The submitted files:

```
img/
  align_{a3,a4,int}.svg  # illustrations for this document
src/
  aligner.py             # main aligner code
  run_eval.py            # runs a configuration and evaluates it
  fastalign.sh           # runs and evaluates fast_align configurations
  extractor.py           # hard alignment extractors, re-used from LeverageAlign
  intersector.py         # used for intersection alignments extrensically
  intersect_reverse.py   # computes forward and backward alignments and intersects them
Leverage_Align.pdf       # draft of Levaraging Neural Machine Translation for Word Alignment paper
rust/main.rs             # rust implementation of IBM1
```

Baseline

Baseline observed.

```
Precision = 0.243110
Recall    = 0.544379
AER       = 0.681684
```

Custom Models

IBM1 (A0)

Even though the code uses numpy whenever possible, it still takes considerable amount of time to run on all 100k examples (~10 minutes for 5 steps). This version simply takes the maximum aligned token. This makes a strong assumption that every token is aligned to exactly one other token.

Results for `IBM1 / A0` with 5 EM steps.

```
python3 ./src/aligner.py -n 100000 -e A0 -s 5 | python3 ./jhu-mt-hw/hw2/score-alignments --data jhu-mt-hw/
Precision = 0.615811
Recall    = 0.801775
AER       = 0.324835
```

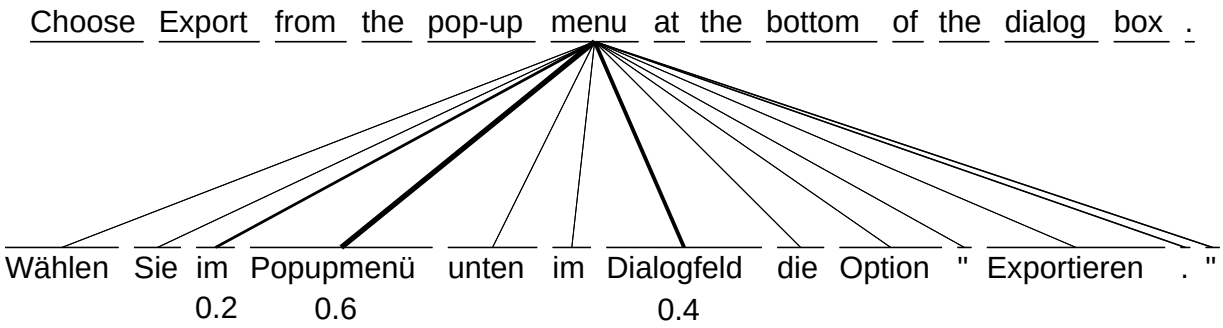
Threshold (A2)

Results for `A2(0.35)` with 7 EM steps.

```
python3 ./src/aligner.py -n 100000 -e A2 -s 7 | python3 ./jhu-mt-hw/hw2/score-alignments --data jhu-mt-hw/
Precision = 0.685567
Recall    = 0.778107
AER       = 0.280435
```

Proportion of best (A3)

This extraction method examines all target tokens and selects the best one. All other alignments which have scores of at least `alpha` times the maximum one are taken. Here `alpha` is a parameter. In the following example consider `alpha=0.4`, then *menu* would align to *Popupmenü* and *Dialogfeld* (mistakenly) and not to others.

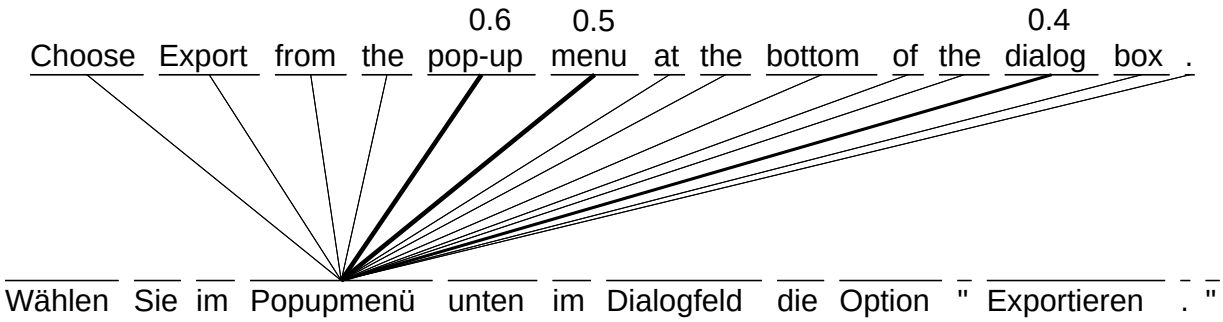


Results for `A3(0.98)` with 7 EM steps.

```
python3 ./src/aligner.py -n 100000 -e A3 -s 7 | python3 ./jhu-mt-hw/hw2/score-alignments --data jhu-mt-hw/
Precision = 0.537281
Recall    = 0.878698
AER       = 0.370400
```

Proportion of reverse best (A4)

The same approach can be done from the target token perspective. Surprisingly, this works a bit better than A3. In the example for `alpha=0.5`, *Popupmenü* would align to *pop-up*, *menu* and *dialog* (mistakenly).

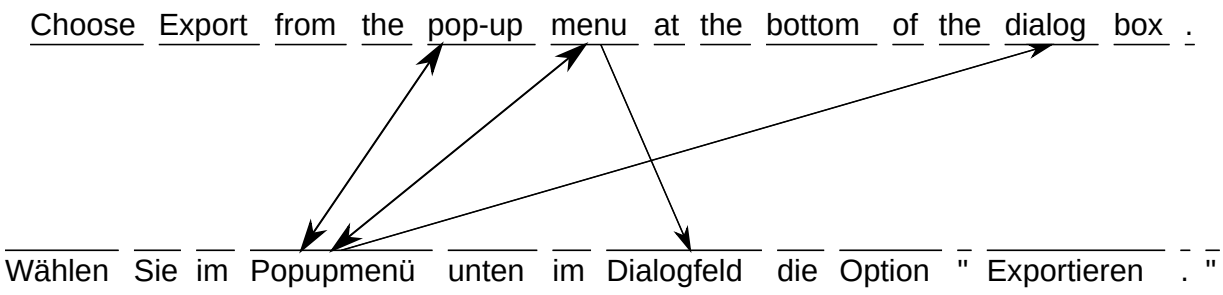


Results for `A4(0.98)` with 7 EM steps.

```
python3 ./src/aligner.py -n 100000 -e A4 -s 7 | python3 ./jhu-mt-hw/hw2/score-alignments --data jhu-mt-hw/
Precision = 0.590789
Recall    = 0.846154
AER       = 0.330601
```

Intersection (A2*A3*A4)

As described by Koehn, intersection in alignment directions increases precision. The intersection of different alignment methods also improves the performance. Intersecting the previous two examples leads to *Popupmenü* being aligned to *pop-up* and *menu*.



The main disadvantage of both A3 and A4 is that even with `alpha=1`, they insist that every token is aligned. Thresholding the alignments improves the performance. Results for `A2(0.25) \cap A3(0.98) \cap A4(0.98)` with 7 EM steps.

```
python3 ./src/aligner.py -n 100000 -e A2*A3*A4 -s 7 | python3 ./jhu-mt-hw/hw2/score-alignments --data jhu-
Precision = 0.758483
Recall    = 0.784024
AER       = 0.231228
```

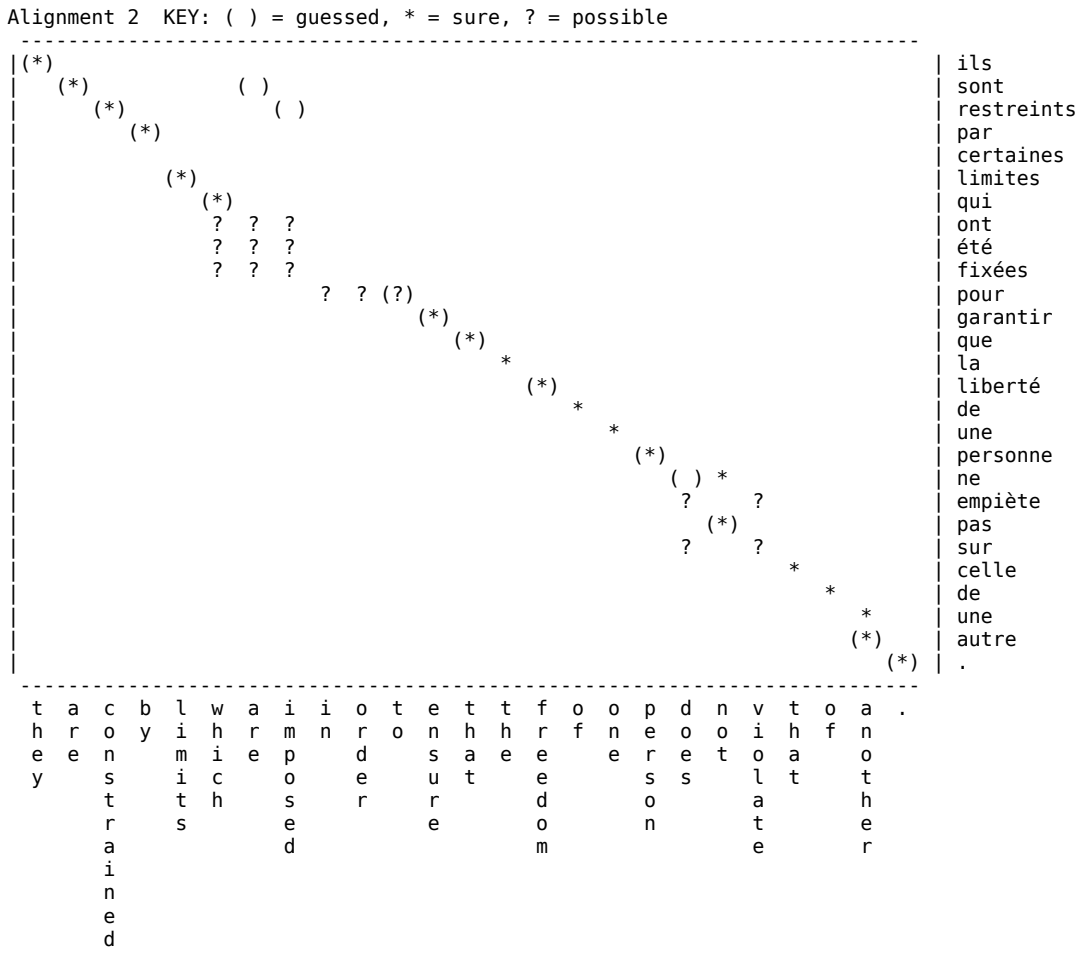
Intersection And Reverse (A2'*A3'*A4' * A2*A3*A4)

The final step combines the previous approach by computing the alignment from the other direction and intersecting it as well. In this version only alignments which are within 7 places of each other are considered. This promotes diagonal alignment. A more elegant solution would be to not create this hard boundary, but only increase scores to alignments on the diagonal. The decision process would then be delegated to the extractors.

Results for `A2(0.25) \cap A3(0.9) \cap A4(0.9) \cap A2'(0.25) \cap A3'(0.9) \cap A4'(0.9)` with 5 EM steps.

```
src/intersect_reverse.sh
Precision = 0.879452
Recall    = 0.718935
AER       = 0.197724
```

The third sentence of the dataset:

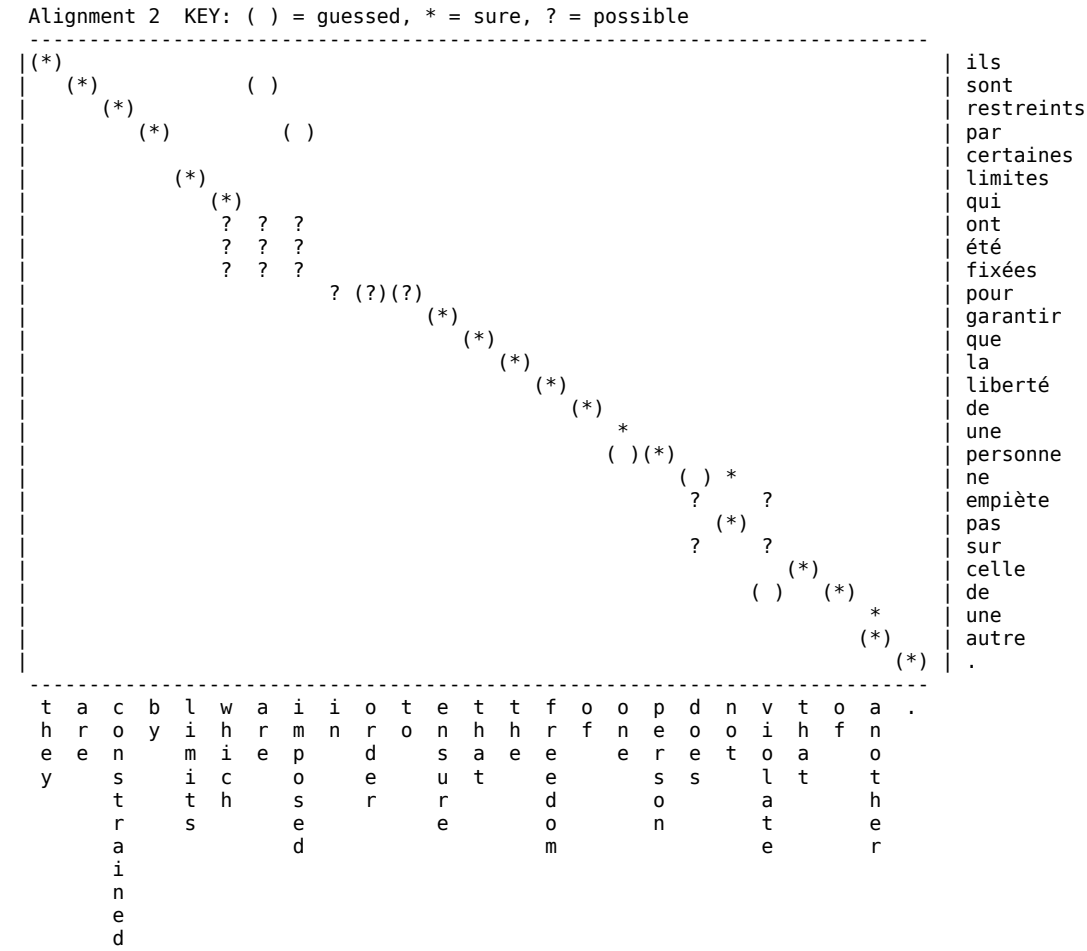


fast_align

The script `src/fastalign.sh` runs fast_align with best practice arguments (`-v` Dirichlet prior and `-do` close to diagonal). EM steps are default 5.

```
Precision = 0.788122
Recall    = 0.831361
AER       = 0.196670
```

Alignment of the third sentence in the data by fast_align. Neither the current implementation nor fast_align guessed the *ont été fixées* phrase correctly, but fast_align performed slightly better. This is true even for overall performance, where fast_align is better by `0.001`. I am however sure, that if one would choose better parameters, then both fast_align and the proposed aligner would produce even better results.



Without `-d` the results are:

```
Precision = 0.645161
Recall    = 0.766272
AER       = 0.313448
```

Performance

I was quite concerned with the implementation speed. So I wrote it in Rust (`rust/`). The time went from `~10min` to `~15s`. This is comparable to fast_align (`~14s`). The comparison is however not fair from both sides, since the Rust implentation uses only IBM1 EM computation, but on the other fast_align is multi-threaded.

Only the A0 extraction method was implemented. The results are almost the same as for the IBM1 / A0 model. The small differences are caused by unstable argmax.

```
Precision = 0.618585
Recall    = 0.798817
AER       = 0.323890
```