Dokumentacja projektowa wrappera do pakietu tłumaczeń statystycznych Phrasal

Łukasz Sieńko, email: l.sienko@stud.elka.pw.edu.pl
Instytut Informatyki, Wydział Elektroniki i Technik Informacyjnych PW,
wersja dokumentacji: 1.0, data: 09.05.2018

1. Podstawowe informacje o pakiecie Phrasal

Phrasal [1] jest narzędziem służącym do budowania systemów SMT (ang. Statistical Machine Translation) uczącym się na podstawie zbioru par zdań będących tłumaczeniami w dwóch językach, pomiędzy którymi ma odbywać się tłumaczenie budowanego systemu tłumaczeń. Posiada możliwości podobne do systemu Moses [12]. Powstał na Uniwersytecie Stanforda w USA w Stanford NLP Group, a jego głównymi twórcami są: Spence Green, Daniel Cer oraz Christopher D. Manning. Pakiet został stworzony z wykorzystaniem technologii Java.

2. Etapy tworzenia systemu tłumaczeń SMT (ang. Statistical Machine Translation)

System tłumaczeń potrzebuje do działania pewnych danych, które są tworzone w następujących etapach przygotowawczych:

Etap 1) Zgromadzenie tłumaczeń zdań w postaci korpusu równoległego

Oznacza to posiadanie dwóch plików, jeden zawierający tylko tekst wybranego języka a drugi zawierający tylko tekst innego języka. W obu plikach odpowiadające sobie numery wierszy (linie tekstu) zawierają to samo zdanie przetłumaczone na język, którego plik dotyczy. Na przykład, dla języka polskiego i angielskiego, linia numer 234 w pliku ze zdaniami w języku polskim będzie zawierała polskie tłumaczenie zdania angielskiego, umieszczonego w linii o numerze 234 w pliku ze zdaniami w języku angielskim. Istnieją gotowe korpusy równoległe dostępne do pobrania (np. [13]) lub można samemu wygenerować wyrównane korpusy na poziomie zdań na podstawie posiadanych dokumentów i ich tłumaczeń. Istnieją do tego dedykowane narzędzia np.: yalign [14], yasa [15] lub Microsoft Bilingual Sentence Aligner [16]. Istnieją różne algorytmy osiągające różną jakość wyrównania np. Church and Gale. Wytwarzanie korpusów równoległych jest

zagadnieniem samym w sobie i na tym etapie nie zajmowałem się analizą tego problemu. Korzystałem z gotowych korpusów – więcej informacji w punkcie nr 6.

Etap 2) Tokenizacja i zamiana wielkich liter na małe

To pierwszy etap wspierany przez wrapper'a. Oznacza to, że do wrapper'a należy wskazać jedynie ścieżkę do plików korpusu równoległego wyrównanego na poziomie zdań w odpowiadających sobie liniach (patrz poprzedni punkt). Ten etap wykonywany jest przez skrypty powłoki Linuxa, które wywoływane są z poziomu Javy w sposób transparentny z punktu widzenia użytkownika wrapper'a. Skrypty są dostępne pod adresem: [17].

Etap 3) Wyrównanie na poziomie wyrazów (ang. word alignment)

Etap ten można wyobrażać sobie jako stworzenie modelu, który jest w stanie wskazać, dla każdego zdania z korpusu równoległego, wyrazy będące tłumaczeniami wyrazów ze zdania po drugiej stronie korpusu. Na przykład mając w korpusie parę zdań: „Anna ma kota" oraz „Ann has a cat", model będzie w stanie wskazać następujące pary wyrazów: „Anna" – „Ann", „ma" – „has", „kota" – „cat". Etap ten stanowi wejście na kolejny, niższy poziom analizy statystycznej języka.

Phrasal obsługuje dwa rodzaje modeli wyrównania. Pierwszy z nich powstaje przez skorzystanie z narzędzia Berkeley Aligner. Alternatywą jest Giza++. We wrapper'rze wykorzystano Berkeley Aligner, ponieważ jest zalecany przez twórców Phrasal'a i jest napisany w języku Java, co umożliwia w stosunkowo łatwy sposób integrację w projekcie. W przypadku problemów z przetworzeniem korpusu można wykorzystać Gizę, która jest programem w postaci skompilowanej, co daje nadzieję na szybsze działanie i być może mniejsze zużycie pamięci. Twórcy narzędzia Berkeley Aligner przewidzieli możliwość konfiguracji procesu budowania modelu przy pomocy wielu różnych parametrów, które wypisane są w punkcie 10.

Etap 4) Budowanie modelu języka

Model języka jest estymowany na podstawie korpusu przy użyciu oprogramowania KenLM. Program ten korzysta z algorytmu „Kneser-Ney smoothing" w wersji bez wygładzania. Działa jako plik wykonywalny (`bin/lmplz`), którego parametry uruchomienia widoczne są na poniższym zrzucie ekranu z dokumentacji KenLM.

Usage

Command line options are documented by running with no argument:

```
bin/lmplz
```

The following arguments are particularly important:

`-o`
: Required. Order of the language model to estimate.

`-S`
: Recommended. Memory to use. This is a number followed by single-character suffix: % for percentage of physical memory (on platforms where this is measured), b for bytes, K for kilobytes, M for megabytes, and so on for G and T. If no suffix is given, kilobytes are assumed for compatability with GNU `sort`. The sort program is not used; the command line is simply designed to be compatible.

`-T`
: Recommended. Temporary file location.

The corpus is provided on stdin and the ARPA is written to stdout:

```
bin/lmplz -o 5 -S 80% -T /tmp <text >text.arpa
```

Mówiąc w skrócie, program do budowy modelu języka uruchamiamy przy pomocy powłoki Linuxa podając rząd modelu (czyli znane nam „n” z modelu typu n-gram: np. dla n=5 dostajemy model obejmujący sekwencje wyrazów: unigramy, bigramy, 3-gramy, 4-gram oraz 5-gramy) plik wejściowy oraz wyjściowy. Opcjonalnie możemy podać ilość pamięci do alokacji i przestrzeń tymczasową.

```
\data\
ngram 1=62390
ngram 2=1393647
ngram 3=5406883
ngram 4=9707861

\1-grams:
-6.1385984      <unk>   0
0       <s>     -1.9076369
-3.1838596      </s>    0
-3.2646976      action  -0.9371025
-3.6327689      taken   -0.65485597
-2.2983341      on      -0.9336027
-3.1544242      parliament      -0.6375023
-2.7733388      's      -0.6293409
-3.93564        resolutions     -0.640998
-2.5238473      :       -0.78280133
-3.8778586      see     -0.46901777
-4.2597737      minutes -0.521681
-3.6898775      documents       -0.6385263
-3.7348511      received        -0.4806644
-3.9276528      written -0.49557167
-3.7198813      statements      -0.71557534
-2.684515       (       -0.5337305
-3.685324       rule    -0.55351293
```

Powyższy zrzut ekranu przedstawia początek pliku modelu języka w formacie ARPA, wygenerowany przy pomocy programu `lmplz`. Linia pliku opisuje sekwencję n-gramu (np.

unigramu jak na powyższym fragmencie) oraz prawdopodobieństwo warunkowe wystąpienia n-gramu. Dla unigramu będzie to prawdopodobieństwo wystąpienia w całym korpusie. Dla 4-gramu np. „w1 w2 w3 w4" będzie to prawdopodobieństwo wystąpienia w4 pod warunkiem, że poprzedza je ciąg „w1 w2 w3". Prawdopodobieństwo jest zapisane w pliku w specjalnym formacie. Pierwsza liczba w wierszu to logarytm o podstawie 10 z prawdopodobieństwa ciągu wyrazów następującego po tej liczbie. Jeżeli po ciągu wyrazów następuje jeszcze dodatkowa, druga liczba (współczynnik), to aby otrzymać wynikowe prawdopodobieństwo należy przemnożyć tę liczbę przez wartość pierwszej liczby. Ta druga liczba zwana jest jako „backoff-weight" i również występuje w postaci logarytmu o podstawie 10 z właściwej wartości współczynnika. Współczynnik „backoff-weight" występuje, gdy dany n-gram pojawia się jako prefix innego n-gramu w modelu. Na początku pliku znajduje się informacja o liczbie unigramów, bigramów, … , n-gramów w modelu.

Plik testowy z modelem, wygenerowany przy pomocy programu `lmplz`, możemy przekształcić do postaci binarnej przy pomocy dołączonego programu `build_binary`. Dzięki temu otrzymujemy model o znacznie krótszym czasie ładowania do pamięci programu, który z niego korzysta. Przekształcenie do postaci binarnej wykonujemy za pomocą polecenia:

```
bin/build_binary trie nazwapliku.arpa nazwapliku.binary
```

Etap 5) Budowanie modelu tłumaczeń

Kolejnym etapem jest zbudowanie modelu, który wraz z modelem języka, posłuży do działania procesu tłumaczenia. Stąd wynika robocza nazwa „model tłumaczeń". Twórcy Phrasal'a nazywają ten etap „ekstrakcja reguł" lub „ekstrakcja tabeli fraz", odpowiednio: „rule extractum" i „extraction of a phrase table". Wynikiem tej fazy są dwa pliki na dysku.

Etap 6) Ładowanie systemu tłumaczeń

Wszystkie poprzednie etapy tworzą „pipeline", czyli potok przetwarzania danych, tzn. każdy etap pobiera pewne dane i wyrzuca na dysk wyniki swojej pracy w postaci zbioru plików. Etapy nie muszą być wykonywane w kolejności opisanej powyżej, ponieważ nie każdy etap wymaga danych z etapu opisanego wcześniej.
Ten etap jest ostatnim etapem i polega na uruchomieniu systemu tłumaczeń, tj. załadowaniu do pamięci modelu języka oraz modelu tłumaczeń. Po załadowaniu, system jest gotowy do tłumaczenia tekstu przychodzącego ze standardowego wejścia lub z innego strumienia np. danych pochodzących z pliku.

Podsumowując, poniższa tabela przedstawia zależność kolejnych etapów (wiersze) od wyników działania pozostałych etapów (kolumny). Zależność oznaczono znacznikiem „potrzebuje” na zielonym tle:

Output / Etap	korpus równoległy	word alignment	model języka	model tłumaczeń	ładowanie modelu tłumaczeń
korpus równoległy					
word alignment	potrzebuje ↑				
model języka	potrzebuje ↑				
model tłumaczeń	potrzebuje ↑	potrzebuje ↑			
ładowanie modelu tłumaczeń			potrzebuje ↑	potrzebuje ↑	

3. Opis projektu i użycia pakietu Phrasal (bez istnienia wrapper’a)

Projekt Phrasal napisany jest w całości w języku Java. Jako projekt open−source, udostępnia oprócz gotowego archiwum JAR z bajt-kodem, również kod źródłowy. W projekcie wykorzystano narzędzie Gradle do budowania oraz pobierania wielu bibliotek zależnych jako plików JAR. Do projektu załączono też kod źródłowy narzędzia KenLM. Oczywiście wszystko należy sobie zbudować lokalnie. Zbudowanie Phrasala narzędziem Gradle jest stosunkowo łatwe w porównaniu z nieco problematycznym procesie budowania narzędzia KenLM. Warto również wspomnieć, że katalogi pakietu Phrasal są bardzo bogate w różnego rodzaju skrypty bash’a, Python’a lub Perl’a o niejasnym przeznaczeniu, którego można się domyślać jedynie po nazwie pliku lub po żmudnej analizie kodu źródłowego. Projekt Phrasal nie posiada dokumentacji ani nawet Javadoc’u kodu źródłowego. Istnieje tylko jeden dokument sformatowany do postaci strony wiki, który daje bardzo ogólne rady jak krok po kroku zbudować system SMT. Moje liczne i usilne próby podążania za tym poradnikiem zakończyły się niepowodzeniem. Strona wiki dobre wyjaśnia jak zbudować narzędzie Phrasal z kodów źródłowych oraz narzędzia dodatkowe, które wymagane są na dalszych etapach. Jednak samo przejście pipeline’u przygotowującego system tłumaczeń do działania jest opisane bardzo skromnie i dopiero analiza kodu źródłowego Phrasal’a, linijka po linijce, umożliwiła mi zrozumienie działania narzędzia na poziomie wystarczającym do uruchomienia i wykorzystania go do eksperymentów. Aby uzupełnić tę krótką listę źródeł informacji o narzędziu Phrasal

należy wspomnieć o istnieniu dwóch artykułach naukowych [3] oraz [4], które co prawda nie pomagają w uruchomieniu i wykorzystaniu narzędzia, lecz informują o ogólnym zastosowaniu pakietu z licznymi odniesieniami do artykułów naukowych i algorytmów z zakresu statystycznego tłumaczenia maszynowego (SMT). Autorzy artykułów kładą szczególny nacisk na zaprezentowanie wyników tj. wydajności i jakości tłumaczenia w odniesieniu do narzędzia Moses, cieszącego się dużą popularnością i powszechną dobrą opinią. Nie wszystkie informacje zamieszczone w artykule okazały się prawdziwe, ponieważ pakiet Phrasal znacznie ewoluował już po wydaniu artykułów. Na przykład artykuł [4] wyjaśnia, że narzędzie umożliwia łatwe i szybkie stworzenie systemu SMT przy pomocy tylko jednego konstruktora klasy CreateModel, podając ścieżki do plików korpusu. Okazuje się, że klasa ta obecnie nie istnieje, stworzenie systemu tłumaczeń jest skomplikowane, a potrzeba istnienia tak prostego mechanizmu jest jednym z powodów stworzenia wrapper'a. W artykule [3] wspomniano również o istnieniu implementacji webserwisu, który ukrywa niepotrzebne z punktu widzenia użytkownika szczegóły implementacyjne pakietu, lecz próba uruchomienia go nie udała się, ponieważ ten kod nie kompiluje się. Istnieją odwołania do klas i metod właściwego kodu Phrasala, które nie istnieją. Najprawdopodobniej kod Phrasala ewoluował, a kod webserwisu nie został zaktualizowany. Problemy spowodowane są popełnieniem błędu projektowego, polegającego na nieistnieniu niezmiennych interfejsów, od których można uzależniać kod zewnętrzny (który nie zależałby od kodu implementacji wewnętrznej logiki działania). Naprawa kodu webserwisu jest trudna dla osoby niebędącej twórcą pakietu Phrasal lub nieznającej logiki wewnętrznej pakietu.

Z punktu widzenia użytkownika pakietu Phrasal, twórcy tego narzędzia do budowy całego systemu SMT przewidzieli istnienie jednego głównego skryptu bashowego `scripts/phrasal.sh`, który uruchamia maszynę wirtualną Javy z wieloma różnymi parametrami oraz opcjami programu, zapisanymi w specjalnie utworzonych do tego celu plikach konfiguracyjnych `example/example.vars` oraz `example/example.ini`. Przeznaczenie tego skryptu jest znane, w przeciwieństwie do innych skryptów w pakiecie, ponieważ wspomniana wyżej strona wiki próbuje wyjaśnić działanie właśnie tego skryptu. Stąd analiza działania tego skryptu, a w szczególności sposobu uruchomienia kodu Javy, posłużyła mi jako punkt wyjścia do zrozumienia Phrasal'a i stworzenia wrapper'a.

4. Motywacja i określenie celu projektu wrapper'a

Wykorzystanie pakietu Phrasal do budowy systemu SMT i badań dot. tłumaczenia maszynowego, w postaci dostarczonej przez twórców narzędzia, okazało się skomplikowane i niewygodne. Zakłada ono korzystanie ze skryptów powłoki Linuxa, które uruchamiają maszynę

wirtualną Javy, która przetwarza wczytane dane i zapisuje wynik do pliku. Następnie uruchamiamy maszynę wirtualną Javy z innymi parametrami, które wykonują inną operację, zdefiniowaną przez inne opcje konfiguracyjne, które odczytuje z pliku skrypt bash'owy. Taka forma pracy z oprogramowaniem jest charakterystyczna dla skompilowanych programów np. napisanych w C++, które wsadowo przetwarzają dane i które uruchamia się z poziomu powłoki Linux'a. Twórcy narzędzia w artykule [4] wspomnieli, że zdecydowali się na wykorzystanie technologii Java z powodu „dobrej równowagi pomiędzy wydajnością i produktywnością programisty". Wszystko wskazuje na to, że zapomnieli w tym kontekście o programiście, użytkowniku końcowym narzędzia. Podstawowym założeniem paradygmatu programowania obiektowego jest enkapsulacja i istnienie dobre zdefiniowanych, niezmiennych i udokumentowanych interfejsów programistycznych, które ukrywają nieistotną dla użytkownika kodu implementację. Dopiero wtedy korzystanie z takiego kodu może być łatwe i przyjemne, co znacznie wpływa na poziom tego, co zostało nazwane przez twórców Phrasal'a „produktywnością programisty".

Celem tego projektu jest po pierwsze zrozumienie logiki wewnętrznej Phrasal'a na takim poziomie by stworzyć kod Javy, który buduje system tłumaczeń. Następnie należy wystawić interfejs Javowy (fasadę/API), który umożliwia w łatwy sposób budowanie systemu SMT, przykrywając niepotrzebne kwestie implementacyjne Phrasal'a oraz użycie dodatkowego oprogramowania zewnętrznego (np. tokenizer, Berkeley Aligner, KenLM). Gotowy wrapper umożliwi wykonanie eksperymentów związanych z tłumaczeniem maszynowym.

5. Przygotowanie Phrasal'a do użycia we wrapperze

Pierwszym krokiem przygotowawczym było poprawienie błędu istniejącego w pakiecie Phrasal, w metodzie:

`edu.stanford.nlp.mt.train.FlatPhraseExtractor.addPhrasesToIndex(WordAlignment sent)`.

Poniższy zrzut ekranu przedstawia początek metody w wersji już poprawionej. W oryginale instrukcja `alGrid.init(sent)` znajduje się przed instrukcją `if`, przez co metoda nie odrzuca skutecznie błędnych danych wejściowych, a więc danych przekraczających maksymalny rozmiar, co prowadzi do rzucenia wyjątku i przerwania pracy programu.

```
protected boolean addPhrasesToIndex(WordAlignment sent) {

  int fsize = sent.f().size();
  int esize = sent.e().size();

  if (fsize > MAX_SENT_LEN || esize > MAX_SENT_LEN) {
    System.err.println("Warning: skipping too long sentence. Length: f="
            + fsize + " e=" + esize);
    return false;
  }

  alGrid.init(sent);
```

Następnie pakiet Phrasal został spakowany do jednego pliku JAR razem ze wszystkimi swoimi bibliotekami, z których korzysta. Ten plik JAR został następnie użyty jako biblioteka projektu wrapper'a.

6. Opis testowych zbiorów danych

Dane testowe, z których korzystałem podczas testowania działania wrapper'a pochodzą ze strony: [18].Wykorzystałem polsko-angielski korpus równoległy zawierający 632 565 par zdań w tych językach. Dodatkowe zbiory danych można pobrać stąd: [19]. Korpus równoległy został wyrównany na poziomie zdań przez dostawcę danych. Zawiera wypowiedzi polityków w Parlamencie Europejskim.

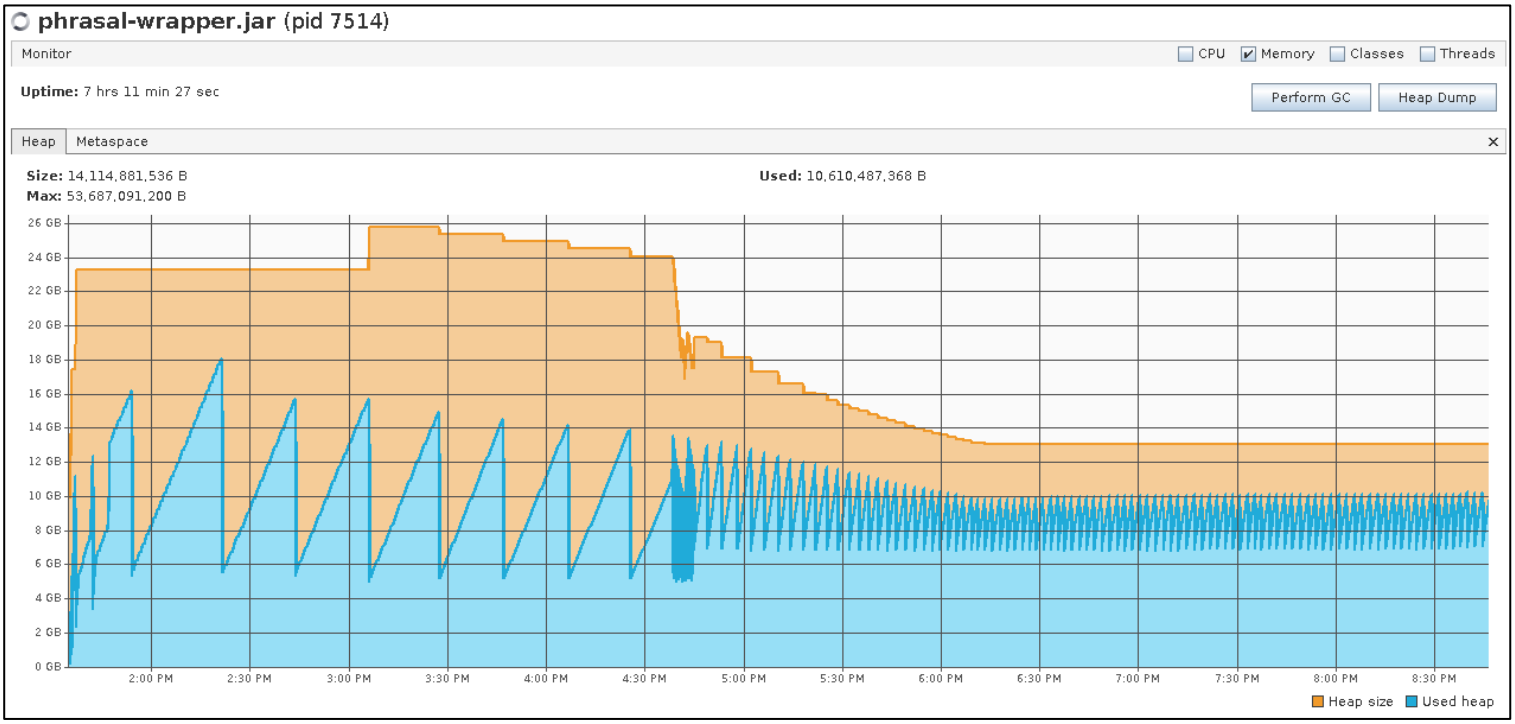
7. Czas trwania obliczeń i zużycie pamięci w poszczególnych etapach

Podczas wykonanych eksperymentów, czas trwania obliczeń różnił się dla tej samej wersji oprogramowania i tego samego zbioru danych testowych. Być może miało to związek z różnym obciążeniem serwera podczas kolejnych eksperymentów. Czas trwania podczas wykonanych trzech prób dla stabilnej, pozbawionej błędów wykonania, wersji wrapper'a prezentuje poniższa tabela.

Etap	Eksperyment nr 1	Eksperyment nr 2	Eksperyment nr 3
tokenizacja	1min	1min	1min
word alignment	2h 52 min	5h 25min	12h 21min
model języka	1,5 min	1,5 min	1,5 min
model tłumaczeń	33 min	51 min	1h 15min
ładowanie mod. tł.	6 min	6 min	7 min
SUMA	~3h	~6.5h	~13h 45min

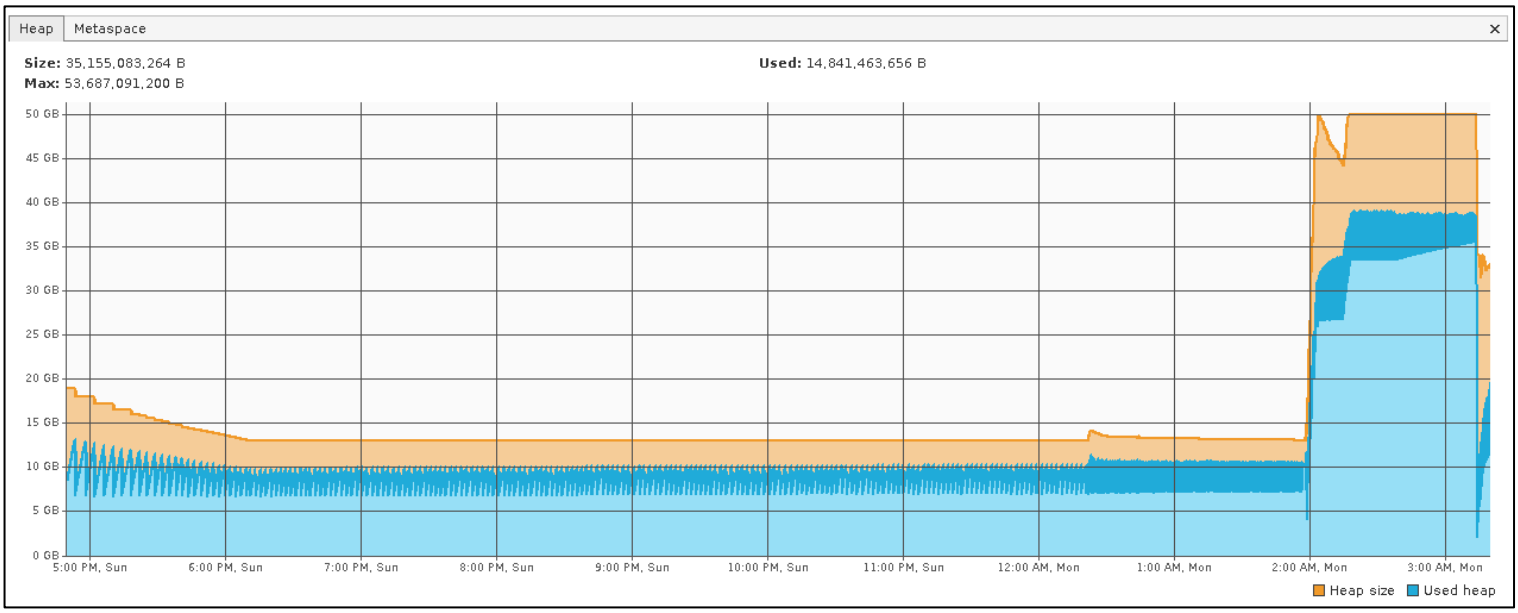
Wnioskiem z pomiarów czasu trwania jest to, że dość zaskakujące są aż tak duże różnice. Postaram się ustalić co wpływa na tak duże wahania. Widać wyraźnie, że wąskim gardłem jest tutaj etap wyrównywania zdań na poziomie wyrazów. Pozostałe etapy zajmują stosunkowo mało czasu. Tak naprawdę czas trwania obliczeń nie jest dużym problemem, ponieważ dla całego korpusu danych są wykonywane jednorazowo. Proces uczenia można uruchomić na noc i następnego dnia rano nowy system jest gotowy do pracy. Każde kolejne uruchomienie tak powstałego systemu będzie trwało tyle ile ładowanie modelu tłumaczeń, czyli 6 minut, ponieważ wszystkie etapy przygotowawcze wyniki swoich obliczeń zapisują na dysku twardym. W przypadku gdyby czas tworzenia nowego systemu SMT zaczął mieć znaczenie, należy wypróbować alternatywne narzędzie do wyrównywania zdań (*word-alignment'u*), czyli Giza++.

Zużycie pamięci nie zmieniało się znacznie pomiędzy eksperymentami 1 – 3. Jednak z punktu widzenia prowadzonych eksperymentów, jest to krytyczny czynnik, mogący istotnie ograniczyć rozmiary przetwarzanych danych. Serwer posiada bardzo dużą ilość pamięci RAM (około 60GB), lecz pojemność ta jest stała. Na wykonanie obliczeń można poczekać kilka czy nawet kilkanaście godzin więcej, ale tylko pod warunkiem, że dostępność pamięci dla obliczeń będzie wystarczająca by proces mógł się pomyślnie zakończyć. Poniższe wykresy przedstawiają zużycie pamięci dla eksperymentu nr 3.



Eksperyment nr 3 zakładał początkowy rozmiar sterty 10GB oraz ograniczenie na maksymalny rozmiar wynoszący 50 GB. Pierwszy etap, tokenizacja, trwa bardzo szybko i

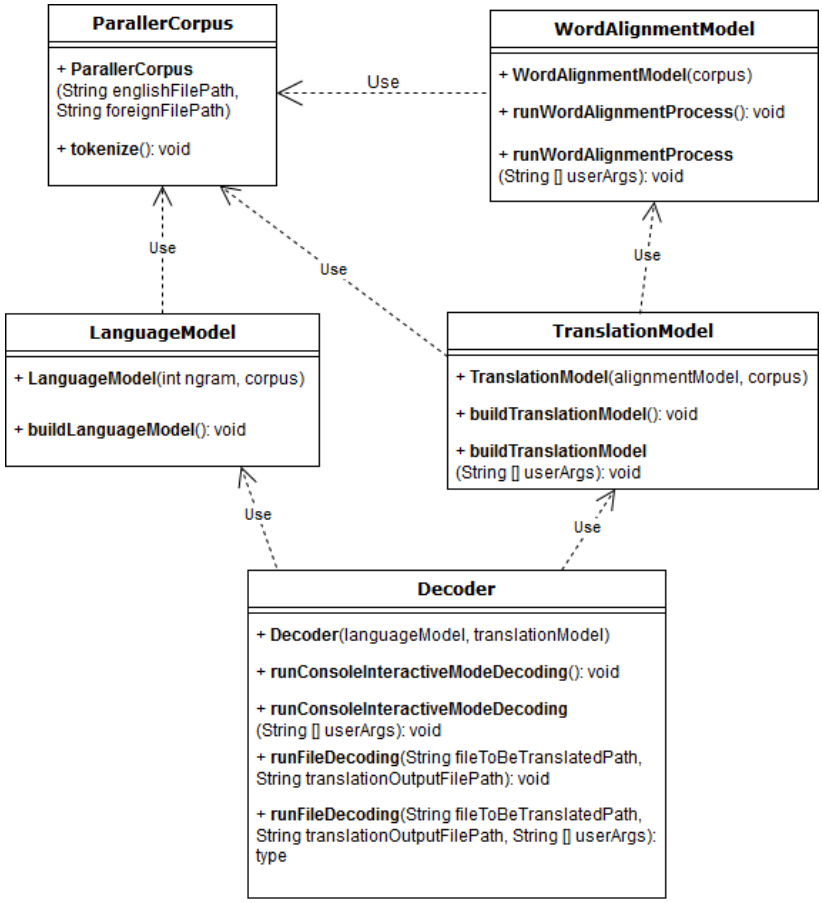
zużywa mało pamięci RAM. Kolejny, word-alignment, zajmuje cały powyższy wykres (i nie jest to koniec tego etapu obliczeń). Zużywa maksymalnie 18GB RAM oraz maksymalnie alokuje 26GB sterty. Dalszy ciąg obliczeń na poniższym wykresie.



Jak widać, w dalszym ciągu proces *word-alignment'u* stabilizuje się nie przekraczając 15 GB. Następnym etapem jest budowanie modelu tłumaczeń, w którym rozmiar sterty osiąga wartość maksymalną (50GB), natomiast realne zużycie sięga prawie 40GB. Ostatni etap to ładowanie modelu tłumaczeń do pamięci (uruchomienie systemu) gdzie rozmiar sterty nie przekroczył 35 GB, a realne zużycie osiągnęło 20 GB.

8. Opis użycia wrappera

W celu wygodnego użycia wrapper'a przygotowałem zestaw klas, które w elegancki sposób, zgodny z metodyką obiektową, opakowują nieistotne szczegóły implementacyjne. Poniżej zamieszczam diagram klas wrapper'a:



Maszynę wirtualną Javy uruchamiamy z parametrami określając m.in. pamięć sterty:

```
-ea -Xmx50g -Xms10g -XX:+UseParallelGC -XX:+UseParallelOldGC
```

Na początku tworzymy obiekt reprezentujący korpus równoległy podając w konstruktorze ścieżki do plików korpusu równoległego. Opcjonalnie możemy wykonać tokenizację i zmniejszenie liter korpusu na małe. Ta operacja nadpisze istniejące pliki.

```
ParallerCorpus corpus = new ParallerCorpus(englishFilePath, foreignFilePath);
corpus.tokenize();
```

Phrasal wymaga wyrównania na poziomie wyrazów. Operację wykonujemy jednokrotnie.

```
WordAlignmentModel alignmentModel = new WordAlignmentModel(corpus);
alignmentModel.runWordAlignmentProcess();
```

Warto dodać, że poprzednie polecenie utworzy podfolder `/models/aligner_output` i tam zapisze wyniki pracy. Podfolder `/models` jest stworzony w folderze, w którym znajduje się plik angielskiej strony korpusu równoległego. Teraz pora na zbudowanie modelu języka:

```
LanguageModel languageModel = new LanguageModel(5, corpus);
languageModel.buildLanguageModel();
```

Tutaj prosimy o model n-gramowy podając wartość 5 jako parametr. Wyniki zapisane są do podfolderu `/models/language_model`, podobnie jak wyniki *word-alignment'u*. Budowanie modelu tłumaczeń będzie wymagało podania referencji do modelu wyrównania oraz korpusu:

```
TranslationModel translationModel = new TranslationModel(alignmentModel, corpus);
translationModel.buildTranslationModel();
```

Dochodząc do ostatniego etapu, ładujemy system tłumaczeń. System może być gotowy do pracy tłumaczenia interaktywnego lub po uruchomieniu przetłumaczyć wskazany plik i zakończyć działanie. Wybierając opcję pracy interaktywnej należy wywołać metodę:

```
Decoder decoder = new Decoder(languageModel, translationModel);
decoder.runConsoleInteractiveModeDecoding();
```

Samo stworzenie obiektu `Decoder` nie uruchamia procesu ładowania systemu tłumaczeń. Dokonuje się to podczas pracy metody `runConsoleInteractiveModeDecoding` jak powyżej lub `runFileDecoding` w przypadku potrzeby tłumaczenia całego pliku.

Wszystkie wyżej wołane konstruktory nie powodują uruchomienia jakichkolwiek obliczeń. Obliczenia kolejnych etapów wykonywane są podczas wołania metod tych obiektów. Metody zapisują wyniki obliczeń w podfolderze `/models`, gdzie znajduje się struktura folderów z wynikami kolejnych etapów. Struktura jest tworzona automatycznie, więc nigdy nie trzeba tworzyć żadnego folderu ręcznie. Jednocześnie istnieje możliwość obliczenia wyników jednego z etapów, pozostawiając wyliczenie pozostałych na później (program wykryje tę sytuację i znajdzie istniejące modele oraz utworzy w odpowiednim miejscu brakujące foldery z wynikami podczas późniejszego uruchomienia reszty obliczeń. W przypadku wielokrotnego obliczenia tego samego etapu, istniejące modele będą nadpisywane przez nowe. Chcąc tylko uruchomić już stworzony wcześniej systemu tłumaczeń, należy upewnić się, że podfolder z modelami znajduje się w odpowiednim miejscu, stworzyć obiekty wrappera oraz wywołać pożądaną metodę klasy `Decoder`. Do metod kolejnych etapów można wstrzykiwać własne parametry. Więcej na ten temat w punkcie 10.

9. Bibliografia oraz pozostałe źródła informacji

[1] https://nlp.stanford.edu/phrasal/

[2] https://nlp.stanford.edu/wiki/Software/Phrasal2

[3] **Phrasal: A Toolkit for New Directions in Statistical Machine Translation**
Spence Green, Daniel Cer and Christopher D. Manning, Computer Science Department, Stanford University

[4] **Phrasal: A Toolkit for Statistical Machine Translation with Facilities for Extraction and Incorporation of Arbitrary Model Features**
Daniel Cer, Michel Galley, Daniel Jurafsky and Christopher D. Manning, Stanford University, Stanford, CA 94305, USA

[5] https://kheafield.com/code/kenlm/

[6] https://kheafield.com/code/kenlm/structures/

[7] **Scalable Modified Kneser-Ney Language Model Estimation**
Kenneth Heafield, Ivan Pouzyrevsky, Jonathan H. Clark, and Philipp Koehn. ACL, Sofia, Bulgaria, 4—9 August, 2013

[8] **KenLM: Faster and Smaller Language Model Queries**
Kenneth Heafield. WMT at EMNLP, Edinburgh, Scotland, United Kingdom, 30—31 July, 2011

[9] http://www1.icsi.berkeley.edu/Speech/docs/HTKBook3.2/node213_mn.html

[10] http://www.speech.sri.com/projects/srilm/manpages/ngram-format.5.html

[11] https://cmusphinx.github.io/wiki/arpaformat/

[12] http://www.statmt.org/moses/

[13] http://matrix.statmt.org/resources/matrix?set=euro_all

[14] https://github.com/machinalis/yalign

[15] http://rali.iro.umontreal.ca/rali/?q=en/yasa

[16] https://www.microsoft.com/en-us/download/details.aspx?id=52608

[17] http://statmt.org/europarl/v7/tools.tgz

[18] http://statmt.org/europarl/

[19] http://matrix.statmt.org/resources/matrix?set=eu_official

10. Możliwości konfiguracji działania poszczególnych etapów

Każdy z etapów budowania systemu tłumaczeń wymaga podania wielu parametrów konfiguracyjnych. Po przeanalizowaniu wybrałem zestaw wartości domyślnych dla każdego z nich. W przypadku potrzeby innej konfiguracji któregoś z etapów, wrapper daje możliwość nadpisania całej domyślnej konfiguracji (nie pojedynczych opcji) własnym zestawem. Możliwość jest dostępna dla niżej wymienionych etapów przygotowania systemu SMT.

- Etap 3) Wyrównanie na poziomie wyrazów (ang. word alignment)
- Etap 5) Budowanie modelu tłumaczeń
- Etap 6) Ładowanie systemu tłumaczeń

Aby nadpisać konfigurację należy przekazać tablicę własnych parametrów do metod implementujących powyższe etapy, identycznie jak jest to zrealizowane w funkcji main zwykłego programu w Javie.

Poniżej znajdują się tabele dostępnych parametrów dla każdych z etapów wraz z wyjaśnieniem dostarczonym przez twórców narzędzi.

Etap 3) Wyrównanie na poziomie wyrazów (ang. word alignment)

Nazwa parametru <typ wartości>	Opis [Domyślna wartość]
log.maxIndLevel < int>	Maximum indent level. [10]
log.msPerLine < int>	Maximum number of milliseconds between consecutive lines of output. [1000]
log.file < str>	File to write log. []
log.stdout < bool>	Whether to output to the console. [true]
log.note < str>	Dummy placeholder for a comment []
log.forcePrint < bool>	Force printing from logs* [false]
log.maxPrintErrors < int>	Maximum number of errors (via error()) to print [10000]
EMWordAligner.nullProb < dbl>	How to assign null-word probabilities (=1 means 1/n) [1.0E-6]
EMWordAligner.mergeConsiderNull < bool>	When merging expected sufficient statistics, take into account the NULL (fix). [false]
EMWordAligner.handleUnknownWords < bool>	Don't crash with unknown words (better to train on test set). [false]
EMWordAligner.priorFraction < dbl>	Fraction of a count to add for links in dictionary prior (1 works well). [1.0]
EMWordAligner.numThreads < int>	Number of concurrent threads to use during E-step (set to number of processors). [1]
EMWordAligner.safeConcurrency < bool>	Safe concurrency (gets rid of concurrency warnings at the expense of speed) [false]
EMWordAligner.evaluateDuringTraining < bool>	Whether to evaluate the model after each training iteration (slower, more memory). [false]
TreeWalkModel.usePushProbabilities < bool>	Separate parameters for moving and pushing. [true]
TreeWalkModel.conditionOnTag < bool>	Whether to condition distortion on the tag types. [true]
TreeWalkModel.cacheTreePaths < bool>	Whether to cache paths through trees (uses lots of memory; faster). [false]
Data.trainSources < str*>	Directories or files containing training files. [example/train]
Data.testSources < str*>	Directory or file containing testing files. [example/test]
Data.sentences < int>	Maximum number of the training sentences to use [2147483647]
Data.offsetTrainingSentences < int>	Skip this number of the first training sentences [0]
Data.maxTrainingLength < int>	Maximum length (in words) of a training sentence [200]
Data.maxTestSentences < int>	Maximum number of the test sentences to use [2147483647]
Data.offsetTestSentences < int>	Skip this number of the first test sentences [0]
Data.foreignSuffix < str>	Foreign language file suffix [f]
Data.englishSuffix < str>	English language file suffix [e]
Data.reverseAlignments < bool>	Reverse test set alignments (i.e., foreign to english) [false]
Data.lowercaseWords < bool>	Convert all words to lowercase [false]
Data.leaveTrainingOnDisk < bool>	Don't load and store the training set upfront (slower, but less memory) [false]
Data.saveRejects < bool>	Save rejected sentence pairs [false]
Data.addTestToTrain < bool>	Train on the test set (recommended) [true]
Data.dictionary < str>	Bilingual dictionary file (e.g., en-ch.dict) [example/en-ch.dict]
Data.splitDefinitions < bool>	Breaks up multi-word definitions and enters each word into the dictionary map [false]
Evaluator.searchForThreshold < bool>	Evaluate using line search [false]
Evaluator.thresholdIntervals < int>	Sets the number of intervals for posterior threshold line search [20]
Evaluator.saveAlignmentObjects < bool>	Save object files for proposed alignments (large files) [false]
Evaluator.usePosteriorDecoding < bool>	Use posterior decoding (recommended for best performance). [true]
Evaluator.posteriorDecodingThreshold < dbl>	Threshold in [0,1] for deciding whether an alignment should exist. [0.5]
Evaluator.writePosteriors < bool>	Produce posterior alignment weight file when aligning training (lots of disk space) [false]
Evaluator.writePosteriorThreshold < dbl>	Minimum posterior value to be written (others assumed to be 0) [1.0E-4]
Evaluator.writeGIZA < bool>	Produce GIZA++-style alignment files [false]
Evaluator.alignInputsSeparately < bool>	Output alignments for each input file in a separate .align file [false]
Main.forwardModels <enum*>	Which word alignment model to use in the forward direction. [MODEL1 HMM]
Main.reverseModels <enum*>	Which word alignment model to use in the backward direction. [MODEL1 HMM]
Main.iters < int*>	Number of iterations to run the model. [5 5]
Main.mode <enum*>	Whether to train the two models jointly or independently. [JOINT JOINT]
Main.trainingCacheMaxSize < int>	Max sentence length for caching the HMM trellis (efficiency only). [30]
Main.loadParamsDir < str>	Directory to load parameters from. []
Main.loadLexicalModelOnly < bool>	When true, the lexical model is loaded, but the distortion model is not. [true]
Main.saveParams < bool>	Whether to save parameters. [true]
Main.saveAlignOutput < bool>	Whether to save test alignments produced by the system. [true]
Main.alignTraining < bool>	Produce two GIZA files and a Pharaoh file for translation [false]
Main.saveLexicalWeights < bool>	Produce two lexical translation tables for lexical weighting (unsupported) [false]
Main.competitiveThresholding < bool>	Use competitive thresholding to eliminate distributed many-to-one alignments [false]
Main.evaluateDirectionalModels < bool>	Evaluate directional models alone [false]
Main.evaluateHardCombination < bool>	Evaluate hard alignment combinations [false]
Main.evaluateSoftCombination < bool>	Evaluate soft alignment combinations [false]
Main.rantOutput < bool>	Output a lot of junk (largely unsupported) [false]
exec.create < bool>	Whether to create a directory for this run; if not, don't generate output files [false]
exec.monitor < bool>	Whether to create a thread to monitor the status. [false]
exec.execDir < str>	Directory to put all output files; if blank, use execPoolDir. []
exec.execPoolDir < str>	Directory which contains all the executions (or symlinks). []
exec.actualExecPoolDir < str>	Directory which actually holds the executions. []
exec.overwriteExecDir < bool>	Overwrite the contents of the execDir if it doesn't exist (e.g., when running a thunk). [false]
exec.useStandardExecPoolDirStrategy < bool>	Assume in the run directory, automatically set execPoolDir and actualExecPoolDir [false]
exec.printOptionsAndExit < bool>	Simply print options and exit. [false]
exec.miscOptions < str*>	Miscellaneous options (written to options.map and output.map, displayed in servlet); exam

Nazwa parametru <typ wartości>	Opis [Domyślna wartość]
exec.addToView < str*>	Name of the view to add this execution to in the servlet []
exec.recordPath < str>	Record file to write to []
exec.charEncoding < str>	Character encoding []
exec.jarFiles < str*>	Name of jar files to load prior to execution []
exec.dontInitializeJars < bool>	Skip initialization of jars [false]
exec.initializeJarsAfterDirCreation < bool>	Initialize from jars after copying them to a newly created execDir [false]
exec.makeThunk < bool>	Make a thunk (a delayed computation). [false]
exec.thunkAutoQueue < bool>	A note to the servlet to automatically run the thunk when it sees it [false]
exec.thunkPriority < int>	Priority of the thunk. [0]
exec.thunkMainClassName < str>	Launch this class []
exec.thunkJavaOpts < str>	Java options to pass to Java when later running the thunk []
exec.thunkUseScala < bool>	Use Scala to run rather than Java [false]
exec.thunkReqMemory < int>	Use Scala to run rather than Java (in MB) [1024]
exec.dontCatchExceptions < bool>	Whether to catch exceptions (ignored when making a thunk) [false]

Etap 5) Budowanie modelu tłumaczeń

Nazwa parametru	Opis parametru
Sets of mandatory arguments (user must select either set 1, 2, or 3)	
Set 1	
-fCorpus <file>	source-language corpus
-eCorpus <file>	target-language corpus
-align <file>	alignment file (Moses format)
Set 2	
-fCorpus <file>	source-language corpus
-eCorpus <file>	target-language corpus
-feAlign <file>	f-e alignment file (GIZA format)
-efAlign <file>	e-f alignment file (GIZA format)
Set 3	
-inputDir <directory>	alignment directory created by Berkeley aligner v2.1
Optional arguments	
-symmetrization <type>	alignment symmetrization heuristic (expects -feAlign and -efAlign)
-extractors <class1>[:<class2>:...:<classN>]	feature extractors
-fFilterCorpus <file>	filter against a specific dev/test set
-fFilterList <file>	phrase extraction restricted to this list
-split <N>	split filter list into N chunks
(divides memory usage by N, but multiplies running time by N)	
-refFile <file>	check features against a Moses phrase table
-maxLen <n>	max phrase length
-maxLenF <n>	max phrase length (source-language)
-maxLenE <n>	max phrase length (target-language)
-numLines <n>	number of lines to process (<0 : all)
-startAtLine <n>	start at line <n> (<0 : all)
-endAtLine <n>	end at line <n> (<0 : all)
-noAlign	do not specify alignment in phrase table
-verbose	enable verbose mode
-minCount <n>	Retain only phrases that occur >= n times
-outputDir path	Output files to <path>
-addSentenceBoundaryMarkers	Add <s> and </s> tokens at the beginning and end of the sentence

Etap 6) Ładowanie systemu tłumaczeń

Nazwa parametru	Opis parametru
-text file	Filename of file to decode
-ttable-file filename	Translation model file. Multiple models can be specified by separating filenames with colons.
-lmodel-file filename	Language model file. For KenLM, prefix filename with 'kenlm:'
-ttable-limit num	Translation option limit.
-n-best-list num	n-best list size.
-distinct-n-best-list boolean	Generate distinct n-best lists (default: false)
-force-decode filename [filename]	Force decode to reference file(s).
-prefix-align-compounds boolean	Apply heuristic compound word alignmen for prefix decoding? Affects cube pruning decoder only. (default: false)
-stack num	Stack/beam size.
-search-algorithm [cube|multibeam]	Inference algorithm (default:cube)
-reordering-model type filename [options]	Lexicalized re-ordering model where type is [classic|hierarchical]. Multiple models can be separating filenames with colons.
-weights-file filename	Load all model weights from file.
-max-sentence-length num	Maximum input sentence length.
-min-sentence-length num	Minimum input sentence length.
-distortion-limit num [cost]	Hard distortion limit and delay cost (default cost: 0.0).
-additional-featurizers class [class]	List of additional feature functions.
-disabled-featurizers class [class]	List of baseline featurizers to disable.
-threads num	Number of decoding threads (default: 1)
-use-itg-constraints boolean	Use ITG constraints for decoding (multibeam search only)
-recombination-mode name	Recombination mode [pharoah,exact,dtu](default: exact)
-drop-unknown-words boolean	Drop unknown source words from the output (default: false)
-independent-phrase-tables filename [filename]	Phrase tables that cannot have associated reordering models. Optionally supports custom per-table prefixes for features (e.g., pref:filename).
-alignment-output-file filename	Output word-word alignments to file for each translation.
-preprocessor-filter language [opts]	Pre-processor to apply to source input.
-postprocessor-filter language [opts]	Post-processor to apply to target output.
-source-class-map filename	Feature API: Line-delimited source word->class mapping (TSV format).
-target-class-map filename	Feature API: Line-delimited target word->class mapping (TSV format).
-gaps options	DTU: Enable Galley and Manning (2010) gappy decoding.
-max-pending-phrases num	DTU: Max number of pending phrases for decoding.
-gaps-in-future-cost boolean	DTU: Allow gaps in future cost estimate (default: true)
-linear-distortion-options type	DTU: linear distortion type (default: standard)
-print-model-scores boolean	Output model scores with translations(default: false)
-input-properties file	File specifying properties of each source input.
-feature-augmentation mode	Feature augmentation mode [all|dense|extended].
-wrap-boundary boolean	Add boundary tokens around each input sentence (default: false).
-ksr_nbest_size int	size of n-best list for KSR computation(default: 0, i.e. no KSR computation).
-wpa_nbest_size int	size of n-best list for word prediction accuracy computation (default: 0, i.e. no WPA computation).
-reference String	reference file for KSR/WPA computation.